A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling

Raphaël Mannadiar

Supervised by: Hans Vangheluwe

School of Computer Science
McGill University
Montreal, Quebec, Canada

April, 13th, 2012

A thesis submitted to McGill University in partial fulfilment of the requirements of the degree of Doctor of Philosophy in Computer Science

Acknowledgments

Thank you Professor Vangheluwe. Thank you for introducing me to the fascinating field that is Domain-Specific Modelling. Thank you for all the opportunities you have given me. Thank you for sharing your knowledge, insights, advice and time with me. Thank you for your support, encouragements and criticisms. I am grateful that our paths crossed.

Thank you to my family, my friends and (last but not least!) my dog for your invaluable contributions. While the vast majority of these were admittedly scientifically modest, neither myself nor this thesis would be quite as complete as we are today without them.

I must also thank the Natural Sciences and Engineering Research Council of Canada for funding my research for the last two years.

Abstract

The complexity of software systems has been steadily increasing over the past few decades. As a result, numerous means of shielding developers from unnecessarily low-level details, or accidental complexity, have been adopted. Today, Model-Driven Engineering (MDE) is considered to be the state-of-the-art of such means. Models describe complex systems from various viewpoints and at various levels of abstraction. To reduce accidental complexity, Multi-Paradigm Modelling (MPM) promotes modelling every part of a system, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s).

At the intersection of MDE and MPM principles is Domain-Specific Modelling (DSM), which promotes modelling using constructs familiar to domain-experts. Documented industrial applications of DSM have demonstrated increases in productivity of up to an order of magnitude, as well as an elevation in the level of abstraction of first class development artifacts. Due to a number of obstacles, DSM has yet to be widely embraced and recognized as a worthwhile and competitive discipline. On the one hand, means to perform essential tasks such as Domain-Specific model (DSm) differencing (e.g., for version control) and debugging are still lacking or incomplete. On the other hand, an enormous strain is placed on DSM experts to develop tools, formalisms and compilers that elegantly hide and encapsulate the complexities of whole families of systems. As such, these experts are often faced with problems and challenges that are several “meta-levels” more complex than the increasingly complex systems whose creation they facilitate.

This thesis contributes to the removal of both the aforementioned obstacles with a strong focus on the last. First, the long-standing Separation of Concerns principles, that have guided much of the improvements in computer-assisted development, are explored in the light of MPM to produce a new and more structured approach to artifact (e.g., executable code, configuration files) synthesis from DSms. Whereas traditional artifact synthesis techniques favour brute force and transform entire models to artifacts via a single and often complex generator, we propose the modular isolation, compilation and later re-combination of each concern within DSms. Second, a side-effect of this layered approach is a much increased ease of examining and manipulating intermediate data and concept representations between DSms and artifacts. This leads to the introduction of a set of guidelines, caveats and examples, that together form a blueprint for future DSM debuggers. Third, the proposed approach to artifact synthesis from DSms is re-examined in the context of Domain-Specific Modelling

Language (DSML) engineering, while remaining mindful of DSM and MPM principles. The result is the extraction of a collection of concepts that form the domain of DSML design and specification, and the introduction of a technique for composing these concepts to create new DSMLs while dramatically reducing the complexity of this notoriously difficult task.

Finally, AToMPM, a new tool for MPM, is presented. In addition to several noteworthy technical innovations and improvements, its main scientific interest lies in its theoretically sound approach towards the specification of modelling language syntax and semantics and of model transformation rule pattern languages, and in its implementation and integration of recent research work by ourselves and others.

Abrégé

Au cours des dernières décennies, la complexité des systèmes logiciels n'a cessé de croître. En conséquence, de nombreuses techniques visant à isoler les développeurs de détail de bas niveau, aussi appellé complexité accidentelle, ont été adoptées. La palme de celles-ci est l'Ingénierie à Base de Modèles (IBM), qui préconise la description de systèmes complexes au moyen de modèles de cible et de niveau d'abstraction variés. Afin de réduire la complexité accidentelle, la Modélisation à Paradigmes Multiples (MPM) prône la modélisation de chaque partie d'un système au niveau d'abstraction le plus approprié, usant des langages les plus appropriés.

La Modélisation Spécifique au Domaine (MSD) se situe au croisement des principes de l'IBM et de la MPM. Elle promeut l'usage de notions familières aux experts des domaines concernés. Des études en milieu industriel ont révélé que la DSM pouvait mener à une augmentation de la productivité allant jusqu'à un ordre de magnitude, tout en élevant le niveau d'abstraction des artefacts de développement. En raison d'un certain nombre d'obstacles, la MSD ne jouit toujours que d'une adoption et d'une renommée limitées. D'une part, les moyens et outils requis pour effectuer des tâches essentielles telles que la comparaison de modèles Spécifiques au Domaine (mSD) (e.g., pour le contrôle de version) et leur débogage demeurent absents ou incomplets. D'autre part, un fardeau énorme est placé sur les épaules d'experts en MSD, à qui incombent les tâches de créer des outils, des langages et des compilateurs qui masquent élégamment la complexité de familles de systèmes entières. Ces experts font souvent face à des épreuves et des problèmes qui se situent à plusieurs "meta-niveaux" de complexité et difficulté au-dessus de ceux des systèmes, eux-mêmes de plus en plus complexes, dont ils facilitent la création.

Cette thèse aborde les deux obstacles mentionnés ci-haut tout en insistant sur le second. Tout d'abord, les principes aguerris de la Séparation des Préoccupations, qui ont guidé bon nombre des améliorations passées dans le développement informatique, sont explorés dans le contexte de la MPM pour produire une nouvelle approche, plus structurée, à la synthèse d'artefacts (e.g., code exécutable, fichiers de configurations) à partir de mSD. Cette approche se distingue des méthodes traditionelles, qui privilégient la synthèse d'artefacts via un unique et souvent très complexe compilateur, en optant plutôt pour une synthèse par phase, qui isole, compile et recombine chacune des préoccupations présentes dans un mSD. Un des effets secondaires de cette approche par phase est qu'il devient largement plus aisé d'examiner, de manipuler et de représenter les données et notions se situant conceptuellement entre les mSD et les artefacts. Exploitant cet avantage, nous proposons une série de recommendations, d'avertissements et d'exemples qui forment une marche à suivre pour le développement de débogeurs pour la MSD. Ensuite, nous réexaminons notre approche de synthèse d'artefacts dans le contexte de la conception de Langages de Modélisation Spécifiques au Domaine (LMSD), en demeurant toujours fidèles aux principes fondateurs de la MSD et de la MPM. Les résultats sont l'identification d'un ensemble de concepts qui forment le domaine de la spécification et du design de LMSD, et l'introduction d'une technique qui permet la création de nouveaux LMSD au moyen de l'agencement de ces concepts. Notre technique réduit drastiquement la difficulté associée à la création de LMSD, une tâche dont la grande complexité est notoire.

En dernier lieu, AToMPM, un nouvel outil de MPM, est présenté. En plus d'un nombre important d'innovations et d'améliorations techniques, ses attraits principaux, d'un point de vue purement scientifique, sont son approche élégante à la spécification de la syntaxe et de la sémantique de langages de modélisation et de langages de motifs, et son intégration de techniques récentes d'auteurs variés, dont nous-mêmes.

Contents

List of Figures

List of Tables

Introduction

Context

The past few decades have seen an explosive increase in the complexity of developed software. This has been addressed by means of increasingly higher-level programming languages and supporting tools (i.e., Integrated Development Environments). These were motivated by the principles of problem abstraction [Dah02], problem decomposition [CLR00] and *Separation of Concerns* [Dij82]. Not due to any fault of their driving principles, programming language improvements eventually proved to be insufficient. Their main shortcoming is the inability to significantly lessen the conceptual gap between problems to solve and their implementation. As a result, model-driven approaches emerged as a means to shift systems development away from code and towards higher-level and more abstract (possibly graphical) models, closer to the problem domain.

At the forefront of model-assisted development, more formally known as Model-Driven Engineering (MDE) [Sch06], are widely accepted standards, most notably the *Unified Modeling Language* (UML) [Obj09]. Unfortunately, in recent years, it has become apparent that existing standards merely offer more appealing views of the systems to-be-built while remaining far too conceptually close to the final implementations, and thus to the solution domain. Hence, their success in truly raising the level of abstraction and overall productivity of software development endeavours has been limited.

An emerging branch of MDE is Domain-Specific Modelling (DSM) [GTK$^+$07]. Its primary aim is to address the aforementioned shortcomings of more traditional modelling efforts and raise the level of abstraction of development tasks sufficiently high for domain experts – with possibly little to no understanding of traditional programming or even modelling – to play first-class roles in solution design and implementation. DSM achieves this by promoting otherwise abstract domain concepts to full-fledged modelling constructs with explicit mappings to artifacts in the solution domain (e.g., Java code, HTML documentation, Petri Net models). Although DSM has been applied successfully in several industrial efforts, it has yet to be widely embraced and recognized as a worthwhile and competitive discipline. While the shortcomings of past and current, more code-centric MDE approaches undoubtedly deserve a share of the blame for this situation, greater obstacles still stand in the way of DSM's widespread adoption. On the one hand, means to perform essential tasks such as model differencing (e.g, for version control) and debugging are still lacking or incomplete. On the other hand, an enormous strain is placed on DSM experts to develop tools, formalisms and compilers that elegantly hide and encapsulate the complexities of whole families of systems. As such, these experts are often faced with problems and challenges that are several "meta-levels" more complex than the increasingly complex systems whose creation they facilitate.

The bulk of current research in the field focuses on enabling DSM projects with tooling comparable to that of the still dominant code-centric development world. These include facilities to differentiate models (e.g., for version control) and debug them. Other studied facilities aim at hiding or automating processes that common programmers are oblivious to but that domain-specific modellers are frequently exposed to, such as language evolution and compiler debugging. Finally, some research has focused on alleviating the aforementioned strain placed on DSM experts to ensure that their limited numbers and required expertise do not become a bottleneck for DSM's adoption.

Problem Statement and Thesis Overview

Domain-specific modelling makes a list of ambitious claims at the top of which is that it can significantly shorten and often completely close the notoriously large conceptual gap between problems and their realizations in the solution domain, a feat that no other computer-assisted development technique has achieved. While these claims have been substantiated in several documented industrial and academic efforts, more focus is often given to the benefits of DSM than to the lower-level implementation details of how exactly the aforementioned conceptual gap is tackled, or to the supporting tooling (or lack thereof). These unsung implementation details often comprise the specification of complex code-generators that make little use of reusable modules and that adhere to no standardized development guidelines. Unfortunately, no such modules or guidelines even exist, an inconvenient fact that holds true in every facet of Domain-Specific Modelling Language (DSML) engineering. As for tooling, while a number of effective tools exist for creating modelling languages, models and model transformations, facilities for version control, debugging and evolution (or maintenance) are still rare or non-existent.

This thesis contributes on both fronts, with a strong bias towards the former. It is guided by Multi-Paradigm Modelling (MPM) principles [MV04], which promote modelling every part of a system, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s). The means by which Domain-Specifc models (DSms[1]) are transformed into artifacts are explored and a novel and more structured approach is proposed that raises the level of abstraction of artifact generators and boasts other beneficial side-effects. One of these pertains to the ease of maintaining traceability links between models, artifacts, and the often very relevant intermediate steps that result from artifact generation. The enabling effects

[1]Note that we refer to Domain-Specific Modelling as DSM and to a Domain-Specific model as a DSm.

of such traceability links are described in the context of model and model transformation debuggers, and a blueprint for future DSM debuggers is proposed. Next, a step back is taken and the means by which DSMLs are specified are explored. The organizing approach introduced to specify artifact generators, otherwise known as DSML semantics, is adapted to the entire process of DSML design. A new technique for specifying DSML syntax *and* semantics is introduced that enables the fully-automated synthesis of otherwise ad-hoc and error-prone syntax and semantics specifications through the composition of reusable DSML "building blocks".

Contributions

This thesis incorporates four core contributions:

1. A *structured approach to artifact synthesis* from DSms that addresses the numerous flaws of widely adopted (within the DSM community) artifact synthesis techniques through the application of MPM principles. Layered model transformations are used to modularly isolate, compile and re-combine various concerns within DSms, while maintaining traceability links between corresponding constructs at different levels of abstraction. A thorough study of the approach reveals a number of its wide-ranging benefits, one of which is its simplifying effect on the notoriously difficult problem of addressing non-functional requirements (e.g., timing and resource utilization constraints). The approach is demonstrated through the synthesis of fully functional Google Android applications from DSms of mobile phone applications.

2. A mapping between common *debugging* concepts (e.g., breakpoints, assertions) from the code-centric development realm to the DSM realm. The meaning of these concepts is explored from both the *modeller*'s and the *DSM expert*'s points of view, where the tasks and debugging requirements of the former are akin to those of programmers, while those of the latter are akin to those of compiler builders. Guidelines, caveats and examples are provided, many of which are implemented and demonstrated, as blueprints for future DSM debuggers. They also serve to demystify the amount of effort required to produce DSM debuggers.

3. A *DSML engineering approach* that automates much of the complex tasks traditionally associated to the specification of DSML syntax and semantics. The basic "domain constructs" of the domain of DSML engineering are identified as being portions of lower-level modelling formalisms that capture commonly occurring syntactic and semantic concepts and structures in DSMLs. Then, a template-based approach for composing these DSML building blocks into new DSMLs is proposed. The approach is demonstrated on two very different DSMLs and studied to clearly identify its benefits and limitations.

4. *AToMPM, A Tool for Multi-Paradigm Modelling.* In addition to several technical innovations and improvements, its main scientific interest lies in its theoretically sound approach towards the specification of modelling language syntax and semantics and of model transformation rules and pre- and post-condition pattern languages, and in its implementation and integration of recent research work by ourselves and others. The features, implementation and usage of the tool are explored in detail.

Outline

This thesis is divided into five chapters. Chapter 1 provides a comprehensive introduction to DSM. Chapters 2 to 5 each introduce one of the aforementioned contributions. Finally, we provide concluding remarks, a summary of the thesis and suggested directions for future research.

Chapter 1

Introduction to Domain-Specific Modelling

The purpose of this chapter is to review important Model-Driven Engineering concepts, as well as the motivations, benefits, limitations and challenges of Domain-Specific Modelling. First, the emergence of Domain-Specific Modelling is explained through an exploration of the history of computer-assisted software development. Then, essential terminology, and the underlying concepts and techniques behind DSM are introduced. Finally, past and current relevant research areas are presented.

1.1 A Brief History on Raising Abstraction: From 0s and 1s to UML, and beyond

The discipline of software engineering has existed for several decades now [NR69]. Initially, it consisted of the ad-hoc mental translation of a system's requirements to a series of punched holes on a paper card. The conceptual gap between problem and solution domains was immense. These early programs were expressed at the lowest possible level of abstraction: their instructions could be interpreted by the underlying computer architecture with virtually no preprocessing. As the demand for more complex systems increased, the need to easily read, understand, debug, maintain and evolve programs grew too, and high(er) level programming languages were adopted to address them. Rather than intended for machine reading, these languages were meant for human reading and manipulation. At first, they were at such a low level of abstraction that nearly each of their statements had a one-to-one mapping to a machine instruction or concept. Later, they evolved and began to abstract away machine concepts enabling software development using constructs closer to the mental model humans have of problems and especially of their (algorithmic) solutions. Programs written in these languages needed to be transformed or *compiled* to machine code to be executable. Initial detractors claimed that not only would such languages be less expressive, their compilers could never produce code as efficient as human-optimized machine code. Although both these concerns may still hold some truth to this day, the tremendous increases in productivity (due to faster development and easier maintenance, for example) quickly proved to compensate for these inconveniences and very few modern applications still rely on coding in low-level machine languages.

Over the following decades, languages at increasingly higher levels of abstraction were adopted in a slow but steady motion away from machine concepts and towards problem concepts to address the increasing scale and complexity of software projects. The evolution of programming languages boomed with the advent of Object-Oriented Programming (OOP) [DMN70, Cox87, Dah02] which promotes the representation of abstract concepts as instantiable *classes* that encapsulate concept-relevant properties and behaviour. Current improvements to OOP languages revolve around addressing their limitations with respect to *cross-cutting concerns* (i.e., features that can not be elegantly modularized within the traditional class-association structure) [KLM^{+}97] and improving their syntax and efficiency [Wikb, AK08]. However, efforts to further raise the level of abstraction of modern OOP languages above algorithmic and code-centric notions appear non-existent. As the plethora of modern General Purpose Languages (GPLs), a superset of OOP languages, have proven themselves effective for the representation of a very wide spectrum of algorithmic solutions, continued efforts to raise the level of abstraction of development efforts have turned to the use of models, and increasingly regard GPLs as targets for automated artifact synthesis.

Visual modelling languages were first introduced as means of representing systems at higher levels of abstraction where distracting implementation details, also known as accidental complexity [Bro87], were hidden. Not only were these abstract models ideal blueprints to guide developers in their implementation, they were also considerably more accessible than code to less technically inclined stakeholders such as clients and managers, who required only moderate training to understand them. These blueprint models are now at the heart of modern software development efforts, providing structure, and promoting modularity and best practices to what were previously ad-hoc development efforts. Today, the most widely known and adopted family of modelling languages is the Object Management Group's (OMG)[1] *Unified Modelling Language* (UML), which proposes (visual) means of representing object-oriented system structure and behaviour. For structure, UML *Class Diagrams* provide notations for the aforementioned object-oriented classes as well as for several types of inter-class *Associations* (e.g., *specialization*, *dependency*). Support for specifying names, types, parameters and visibility of class properties and methods is also provided. For behaviour, UML *Statechart* and *Sequence* diagrams enable the modelling of a system's responses to relevant *events*, and of how objects in a system interact. To enable code-generation, these behaviour diagrams refer to relevant class properties and methods via complex code snippets, commonly expressed in the OMG's *Object Constraint Language* (OCL) [Obj10].

Although UML diagrams have proven to be an effective means of representing desired system facets while hiding undesired facets and implementation details, several researchers have noted the fact that the difference in the level of abstraction between the said diagrams and modern GPLs such as Java and C++ is smaller than that between GPLs and their compiled machine code forms [B0́5, KT08]. The wide conceptual gap between GPLs and assembly code is believed to go hand in hand with the monumental increase in productivity that accompanied the adoption of GPLs in favor of machine code decades ago, and while UML has certainly helped raise abstraction and structure development efforts, the resulting increases in performance are marginal in comparison. Indeed, a close re-examination of the above description of UML reveals that it is profoundly rooted in the object-oriented and programming mindsets. Modellers are still required to be intently aware of code-centric notions like types, attribute visibility and methods. They are expected to be proficient with one or more programming language. They must still mentally translate problem-domain requirements into class-association architectures and state and flow diagrams. They are even required to understand and very often manipulate code generated from UML specifications. Thus, despite its benefits, UML fails to shield developers from numerous implementation technicalities and, worst of all, fails to solve the most difficult aspect of software engineering: the abstract, ad-hoc, complex and possibly counter-intuitive burden of translating problems from their domains to the often far away solution domain – be it low-level C or higher-level UML diagrams – remains.

Though many extensions, or *profiles*, to UML have been proposed to increase its expres-

[1]The Object Management Group is a consortium of multi-national companies responsible for standards such as the *Common Object Request Broker Architecture* (CORBA) and the *XML Metadata Interchange* (XMI).

siveness and flexibility, it remains undeniably inseparable from its OOP roots. As a prime candidate to solve this problem, DSM is believed to be the next step forward that will enable the first great leap in productivity in decades. Already, several real-world experiments using DSM for small to medium scale development efforts have reported decreases of up to an order of magnitude in development time [Bro04, Saf07, KT08].

1.2 Essential Concepts and Terminology

Before proceeding with a review of DSM and how it addresses the limitations of UML-centric development approaches, a number of fundamental concepts must first be defined.

1.2.1 Models

At the core of MDE activities are *models*. Regardless of its representation (i.e., visual or textual), a model should display three high-level properties: it should represent a real system (*mapping feature*), it should abstract away parts of that system (*reduction feature*), and it should be possible to make use of it in place of the system it models for some application (*pragmatic feature*) [K0̈6]. Consequently, the use of models is often motivated by the fact that they are often cheaper, safer and quicker to build, reason about, test and modify than the systems they represent.

Models can be divided into two large categories: *token models* and *type models* [Fav06, K0̈6]. The former offer an abstract view of a single system (e.g., a 5"x5" map of Montreal's subway network as a to-the-point, portable abstraction of the actual network). The latter, which are often referred to as *classification* or *schema models*, are meant to model many systems (e.g., a UML Class Diagram that relates `SubwayStop` and `SubwayLine` constructs as a conceptual model of all existing and possible subway networks).

1.2.2 Meta-models and Meta-modelling

While a model may *represent* a system, it is said to *conform* to a *meta-model* [B0́5]. Meta-models define a possibly infinite set of syntactically legal models – similarly to how the grammar of a textual programming language defines a possibly infinite set of syntactically conforming programs. They achieve this by defining certain sets and conditions. First, a set of valid, possibly attributed, modelling constructs. Second, a possibly empty set of valid relationships between modelling constructs. Last, a set of validity constraints. As an example, consider the abridged description of UML Class Diagrams from the previous section. The meta-model of UML Class Diagrams would specify that `Class` constructs exist, that they have `properties` and `methods` attributes, and that they may be connected to each other via `Specialization` and `Dependency` relationships. A sensible validity constraint might verify that no class transitively specializes itself (i.e., specialization should not be cyclic).

A strong point of consensus within the wider MDE community is that meta-modelling is the key to empowering scalable model-based techniques. In essence, it is not sufficient for systems to be represented by models and for models to conform to meta-models. For operations such as model comparison, model transformation and model merging to be feasible, all meta-models also need to themselves conform to a meta-model, referred to as a *meta-meta-model* [Bro04, B05́]. Popular meta-meta-models are the OMG's *Meta-Object Facility* (MOF) [Obj06], the *Entity-Relationship Model* [sC76] and the OMG's UML Class Diagrams, which are each expressive enough to specify any meta-model, including themselves. In practice, meta-meta-models are at the heart of virtually every modern modelling tool. As such, higher-order operations that take meta-models as parameters become possible and the full and automated synthesis of tools that provide domain-specific modelling, debugging and simulation environments is enabled. Developing such environments for each new domain with conventional software development approaches would be infeasible due to time, cost and complexity constraints.

1.2.3 Modelling Languages

Modelling languages, domain-specific or otherwise, can be broken down into three core components: *abstract syntax*, *concrete syntax* and *semantics*.

Abstract syntax describes language concepts, relationships between them, and validity constraints. For a number of reasons, not the least of which is that scalable DSM tools rely heavily on meta-models and meta-modelling, MDE dictates that modelling language abstract syntax should be specified via meta-models. From this stems the slightly inaccurate but common simplification that meta-models are models of modelling languages.

Concrete syntax provides graphical and/or textual representations of abstract syntax elements. As an example, consider again the UML Class Diagram language. While its abstract syntax defines the notions of `Class` and `Specialization`, it is its concrete syntax that specifies that instances of the former should be depicted by rectangles that contains concise representations of the `properties` and `methods` attributes, and that instances of the latter should be depicted by dotted-lines ending with larger triangular arrow-heads. Note that though modelling languages may only have one abstract syntax, they may have several different concrete syntaxes.

Abstract and concrete syntax specifications are sufficient to synthesize modelling environments for a given modelling language. However, the created models would be of limited use: they could serve as little more than sketches, blueprints and/or documentation. For models to play a more prominent role in development efforts, the language they are expressed in must have associated semantics, or meaning. Semantics are commonly defined *operationally* or *denotationally*[2].

[2]We use the term loosely and do not mean the use of Scott domains. To avoid confusion, denotational

Operational semantics often encode system behaviour and can be described as a collection of "items", each denoting the transformation from one valid system state to another. For example, consider a very simple modelling language for parking lots. An operational semantics item could implement the "parking" action by looking for a car next to an open parking space (one valid system state) and then moving that car into the (consequently occupied) parking space (the next valid system state). By repeatedly executing the previous item along with other similar items for actions such as "driving" and "leaving", models from the given parking lot language can be simulated [EB04].

Unlike operational semantics, denotational semantics define the meaning (or "denotation") of a modelling language by specifying how its instances should be mapped onto models in other (modelling or programming) languages for which operational or denotational semantics are well defined (e.g., code, Petri Nets). Thus, example denotational semantics specifications for the parking lots modelling language might describe how instance models should be compiled to executable Java programs, or to HTML documentation.

MDE principles dictate that both types of semantics should be implemented as *model transformations* [Bro04].

1.2.4 Model Transformations

Model transformations are at the core of MDE best practices for a wide range of model-related activities. Various techniques for specifying model transformations have been proposed. As the focus of this thesis is not to improve on the state-of-the-art of model transformations, only those techniques that will be used or criticized in upcoming chapters are discussed here.

The most primitive means of specifying model transformations is via programs, written as imperative code, that traverse and modify models via modelling tool APIs (i.e., code interfaces for viewing and manipulating internal representations of model entities). Not only do such approaches contradict the MDE philosophy, they considerably complicate many advanced uses of model transformations due to their (accidental) complexity and the difficulty to maintain them. Indeed, the intent of the transformation can easily be lost in implementation details. It is widely accepted that such approaches should be avoided [Por05].

Most currently adopted model transformation approaches are based on *transformation rules* and consider them as elementary entities. Rules are traditionally parameterized by a Left-Hand Side (LHS) and Right-Hand Side (RHS) pattern, one or more optional Negative Application Condition (NAC) pattern, condition code, and action code. The LHS and NAC patterns respectively describe what sub-graphs should and should not be present in

semantics are often referred to as *translational* semantics.

the source model for the rule to be applicable while the RHS pattern describes how the LHS pattern should be transformed by the rule's application. Further applicability conditions may be specified within the condition code while actions to carry out after successful application of the rule may be specified within the action code. The key advantages of rule-based approaches is that they are modular and deal only with modelling concepts. This, as opposed to the aforementioned coded transformations which manipulate internal representations tools have of models. Furthermore, implementation details (the "how") pertaining to the flow of control from one rule to the next, and to how sub-graph matching and rewriting are performed are commonly hidden from the transformation developer, thereby avoiding that the transformation's intent (the "what") be drowned in lower-level concerns.

A powerful rule-based approach for transforming *visual* models are *graph transformations*. Their main appeal is their theoretical foundations in category theory and that they allow for the very intuitive and accessible *visual* specification of patterns. In the past, graph transformation-based approaches were not taken as seriously as they could have for a number of reasons. One of them is that more code-like approaches to rule specification, such as the *ATLAS Transformation Language* [JK06] or the OMG's *Query/View/Transformation* (QVT) [Obja], are often more appealing to developers with strong programming backgrounds. Most importantly, however, is that early graph transformation tools lacked sufficiently powerful mechanisms for specifying the flow of execution of their rules [Tra05]. Indeed, the conventional approach for executing graph transformations was first inspired by the executable semantics of graph grammars (which is extended from that of textual grammars): any applicable rule may execute until there are no more applicable rules. Despite the elegance of this model of execution, missing native facilities for forcing termination and determinism, and the lack of simple means of imposing arbitrary control flow structures (e.g., loops and conditional executions) make it impractical. Today, a number of graph transformations-enabled tools provide more advanced rule-sequencing options. AToM^3 (A Tool for Multi-formalism and Meta-Modelling) extends rule specifications with priorities [dLV02]. *MoTif* [SV09], *GReAT* (Graph Rewriting And Transformation) [AKN$^+$06] and AToMPM (A Tool for Multi-Paradigm Modelling) each support more complex control flow facilities such as conditions, loops, transactions and rule amalgamation.

The field of model transformation reaches far beyond the above overview. Czarnecki and Helsen provide a detailed and comprehensive feature-based classification of model transformation approaches in [CH06]. Due to their numerous advantages, most notably, the ease of elegantly representing them graphically, the contributions presented in this thesis restrict themselves to the use of rule-based graph transformations.

1.2.5 Multi-Paradigm Modelling

MPM aims at maximally reducing accidental complexity in MDE efforts. It argues that all parts of a system should me modelled at the most appropriate level(s) of abstraction, using only the most appropriate formalism(s). Thus, MPM takes a divide-and-conquer approach to the modelling of increasingly complex systems by encouraging that models of complex, multi-faceted systems be separated into a number of single-facet models. Each of these "sub-models", or views, can then be specified with a modelling language optimally tailored towards the relevant facet, thereby shielding the modeller from temporarily extraneous details. MPM also leads naturally to DSM as DSMLs are, by design, at the most appropriate level of abstraction for the modelling of problems in arbitrary domains.

1.3 Domain-Specific Development

The idea of tailoring development environments to arbitrary domains is not new. Already, FORTRAN and COBOL, two half-century old programming languages, were respectively tailored towards the needs of the scientific and business communities. Over the decades, a number of techniques have taken this idea further, the most prominent of which are described below.

1.3.1 Generative Programming

Generative programming aims to elevate software engineering to the same level of automation as other engineering disciplines. In [CE00], Czarnecki and Eisenecker relate the story of how components of manufactured automobiles went from being hand-crafted to becoming standardized, thus enabling modern automobiles to be automatically assembled on assembly lines. Comparing the evolution of the manufacturing industry to current software engineering practices, the authors observe that modern software engineering uses techniques analogous to those used in manufacturing one century ago. Generative programming is introduced as a means to bridge this one century gap by studying and applying lessons from other disciplines. In essence, whenever many software artifacts are relatively similar, rather than coding them each separately, techniques can be used to produce one configurable piece of code in place of many configured pieces of code. A generator of sorts can then, given a desired configuration, automatically produce the final application by appropriately "instantiating" the configurable piece of code. Despite being a definite step in the direction of DSM, this approach, like all of those presented thus far, remains far away from the problem domain. Proposed techniques for writing configurable code are very code-centric, heavily relying on notions like templates, aspects, polymorphism and static meta-programming. Moreover, although the general ideas of generative programming are appealing in the context of generating full artifacts from higher-level specifications, existing applications are mostly targeted at generating artifacts that remain useful only to programmers (such as highly configurable data structures).

1.3.2 Model-Driven Architecture

Recognizing the aforementioned limitations of UML, the OMG attempted to standardize MDE practices into the *Model-Driven Architecture* (MDA) [Objb]. MDA encapsulates standards for model definition, serialization and transformation. At a glance, it appears to be the standardized equivalent of DSM. However, upon closer examination, certain authors argue that it should instead be considered as an approach somewhere between UML modelling and DSM [Bro04, B0́5, KT08]. The main reason for this is that MDA, both in its philosophy and in its proposed implementations, remains strongly coupled with artifacts that DSM and MPM proponents argue should remain hidden from modellers. MDA proposes to view software development as a series of model refinements where lower- and lower-level models, referred to as *Platform-Specific Models*, are (semi-)automatically generated from higher-level ones, referred to as *Platform-Independent Models*. The issue of concern is that modellers are expected to modify and contribute to generated intermediate models. The implications of this problem are better understood when viewed from the more familiar context of programming: requiring modellers to interact with lower-level artifacts generated from their models is equivalent to requiring programmers to interact with the compiled bytecode form of their programs. Clearly, raises in abstraction and productivity comparable to those that GPLs brought on can not be achieved as long as modellers are expected to interact with lower-level artifacts.

1.3.3 Domain-Specific Modelling

DSM is a branch of MDE that takes the notion of abstraction to the furthest possible extreme: DSms are meant to deal only with concepts belonging to *problem* domains. Thus, the tedious, complex and error-prone translation from problem to solution domains is hidden from developers, now domain-specific modellers, and from any other interested stakeholders (e.g., the client). From DSms, a wide variety of solution-domain artifacts may be generated such as executable code, configuration files for existing applications, and documentation. The most common reservation against the adoption of DSM stems from previous attempts at code generation from UML models that have discredited serious code generation approaches to the software engineering community. Such efforts often led to awkward, inefficient, inflexible and/or incomplete generated artifacts. The main reason for this was that the target domain for the said code generation was too large [KT08]: any application can be modelled in UML and so the target domain of a UML-to-code compiler must be the domain of all possible applications. This is where DSM differs drastically from past and current modelling and artifact synthesis approaches. As indicated by their name, DSms are restricted to some specific domain and hence DSm-to-code compilers need only be able to understand valid models of problems in the given domain and produce valid applications in the context of that same domain. The source and target domains for code (or any other type of artifact) synthesis are thus enormously reduced.

The above not only defines DSM but it also hints as to when and why DSM should be used. DSM has the highest potential for well understood domains, in domains where many relatively similar problems exist, and especially in domains where domain experts are otherwise incapable of developing the solution-domain artifacts they require (e.g., non-programmers in the context of software development). By abstracting away the need to translate problem concepts to solution concepts, DSM indeed enables “non-developers” to actively participate in the development of the solutions they require. Proponents of DSM have repeatedly justified its use by citing increased productivity. The underlying reasons for this increase are diverse. First, the repetitive and difficult craftsmanship that usually characterizes problem-to-solution mental translations and programming tasks is entirely absorbed into DSm-to-artifact compilers. This not only means developers don't waste time on this craftsmanship, the automation of the coding process entirely removes the possibility of many common coding errors. Second, DSM environments can easily be made to guide modellers into creating only valid models by enforcing domain rules encoded as meta-model validity constraints. This makes learning new DSMLs an intuitive and organic process and helps enforce certain invariants that considerably simplify the implementation of DSm compilers. In contrast, UML or Java could hardly be expected to guide developers into creating only sensible and runtime error-free applications. Finally, savings in development time are also induced by the fact that DSms are immediately accessible to interested stakeholders, who do not require any prior training (e.g., about OOP or UML).

DSM is not a silver-bullet, however, and has clear and well understood bounds. Implanting a DSM framework takes time: a DSM tool is required, DSMLs (their syntax and semantics) need to be created and properly tested, developers need to be initiated to their usage, etc. In short, the adoption of DSM is not without considerable overhead. Thus, in contexts where a very small number of DSms are needed, the required investment in time and effort may not be worthwhile. Also, for less well-understood domains that are subject to frequent change or where it is difficult to extract core concepts and relationships, and where key points of variability between domain problems are difficult to identify, it may be premature to attempt to deploy a DSM solution. Finally, and perhaps most importantly, a number of critical problems that limit the feasible scalability of DSM projects to industrial needs still require more research. These are explored in the next section.

1.4 Relevant Research Areas

This section briefly introduces relevant research problems within the MDE and DSM communities. More in-depth and thorough surveys are provided within the contribution chapters.

1.4.1 Model and Model Transformation Debugging

The need to debug software systems is as old as software systems themselves. Empirical studies of what causes programs to fail and how programmers eradicate bugs have revealed that error tracking and reproduction are key activities in the debugging process [Eis97, Zel09]. Facilities to pause execution in pre-determined states, to step over or into compound expressions, and to examine internal state variables emerge as indispensable tools for software debugging. An admittedly small number of researchers from the modelling community have explored how these facilities translate to the DSM world and how they might be implemented [WGM08, MV11a], asking and proposing answers to questions like "What does it mean to pause the execution of a model?" Furthermore, a facet of debugging which is mostly ignored by the vast majority of modern programmers is brought to the forefront of debugging in the context of DSM: compiler debugging. Whereas compiler development is a rare activity in the mainstream code-centric development world, every single DSM project involves the development of possibly many facilities akin to compilers (i.e., DSML denotational semantics). In this sense, the debugging challenge in DSM extends past the debugging of models, and onto the debugging of artifact generators, possibly and ideally specified as model transformations.

1.4.2 Domain-Specific Modelling Language Engineering

A number of popular approaches currently exist for the specification of DSML abstract syntax and semantics. However, several researchers have begun to notice that virtually no reuse mechanisms or guidelines exist to organize the design and creation of new DSMLs. For instance, a definition of the *Coloured Petri Net* language might benefit from the reuse (or referencing) of parts or all of the definitions of the concrete syntax, abstract syntax and semantics of the basic Place/Transition Petri Net formalism. Moreover, the definition of a DSML for any domain where notions of state and transition are significant might benefit from the reuse of parts or all of the definitions of the concrete syntax, abstract syntax and semantics of the Statechart formalism. Current work on the topic of DSML engineering focuses on how to enable such reuse, be it by enhancing the structure and modularity of semantics specifications, or by providing alternate design interfaces where reusable components are clearly identified and accessible [CSN05, ES06, MV10, MV11b].

Chapter 2

Structuring the Synthesis of Artifacts from Domain-Specific Models

This chapter introduces a novel and structured approach to artifact generation where layered model transformations are used to modularly isolate, compile and re-combine various concerns within DSms, while maintaining traceability links between corresponding constructs at different levels of abstraction. The proposed technique is explained with the help of a running example that demonstrates how fully functional Google Android applications can be synthesized from DSms of mobile phone applications. Then, a study of how the approach simplifies addressing non-functional requirements (e.g., timing and resource utilization constraints) of modern embedded systems is provided. This simplification is demonstrated by synthesizing performance models from DSms, and then synthesizing performance predictions, simulations and measurement facilities from the performance models.

2.1 Problem Statement and Outline

Due to the very central role played by automatic artifact synthesis in DSM, structuring how DSms are transformed into artifacts is both beneficial and necessary. To this day, the prevalent approach to artifact synthesis from DSms is to programmatically manipulate internal model representations and generate text – often code. Notwithstanding the fact that this approach contradicts MDE principles, it is also riddled with flaws. Firstly, the resulting generators are often conceptually very distant from the models they "compile". This causes their development and their maintenance (e.g., as a result of meta-model evolution) to be tedious, complex and error-prone. Furthermore, implementing "advanced" features which require one- or two-way communication between model and artifact (e.g., model animation as a result of artifact execution) adds considerable accidental complexity to the generators, which in turn worsens their maintainability. Another inconvenience of these hand-coded generators is that, due to their low-level nature and structure, they are difficult to study and analyse. Artifact generators should be anything but difficult to understand as they encode no less than the *semantics* (or meaning) of the DSms themselves. As such, ensuring their correctness and efficiently communicating their inner workings (e.g., among project team members) are high priorities. In short, the traditional approach makes the very crucial semantics of models difficult to specify, debug, maintain and understand.

To address the aforementioned issues of poor maintainability, poor extensibility and low abstraction, we propose that artifact synthesis from DSms be carried out via several visual, rule-based graph transformations, that iteratively isolate and project tangled concerns within DSms onto appropriate lower-level modelling formalisms as an intermediate step to final artifact generation. The first of our two main contributions lies in this novel approach to artifact synthesis. The second lies in the study and demonstration of how the approach can be used to elegantly and modularly replicate numerous assessment activities such as performance analysis, simulation and testing.

The rest of this chapter is structured as follows. In Section 2.2, we survey relevant research work. In Section 2.3, we introduce our approach to artifact synthesis. In Section 2.4, we study and demonstrate how our approach simplifies the addressing of the characteristic non-functional requirements (e.g., timing and resource utilization constraints) of modern embedded systems. In Section 2.5, we present a non-trivial instance model of a mobile phone application and detail its transformation into a fully functional application running on a Google Android [Goob, CD09] device. We also apply the ideas introduced in Section 2.4 to instrument the Google Android application with performance measurement and reporting facilities that update its associated DSm with live performance information. In Section 2.6, we compare our approach to the relevant works presented in Section 2.2. Finally, in Section 2.7, we discuss future work and provide some closing remarks.

2.2 Survey of Relevant Research Work

The purpose of this section is to provide an in-depth survey of research work relevant to the proposed scientific contributions of this chapter. Our contributions are neither compared here nor are they situated with respect to the reviewed works. Such comparisons are instead provided in Section 2.6.

2.2.1 Compiling Domain-Specific Models

Via Hand-coded Text Generators

MDE principles state that model transformations are the preferred means of synthesizing artifacts (whatever these may be) from models [Bro04]. However, few researchers, other than Levendovszky et al. in [LLMM08], have reported the use of model transformations for artifact synthesis in industrially relevant contexts. Indeed, another approach is favoured in most of the works that have explored the complete DSM development process starting from the design of DSMLs to the synthesis of target platform artifacts from instance DSms. In these endeavours, DSms are systematically transformed to target platform artifacts by means of ad hoc hand-coded text generators [Saf07, KT08, Met]. This technique has numerous shortcomings all of which revolve around the fact that the resulting DSm compilers are at too low a level of abstraction. Compiler designers and maintainers must manually implement complex traversal, filtering and matching algorithms and elegantly bundle them as model transformation facilities. They must interact with internal model representations (via tools APIs) rather than with domain-specific constructs in their concrete syntax. Automating the co-evolution of such compilers, following source or target domain evolution, is infeasible, and performing such co-evolution manually is not only lengthy, but often complex and error-prone. Finally, augmenting such compilers with additional (cross-cutting) concerns, such as the construction of DSm-to-artifact mappings to enable two-way communication between DSm and artifact, is certain to further decrease their modularity and accessibility.

Via Model Transformations

Some authors have suggested that model transformation languages, even graphical rule-based ones, may be too difficult to learn and that this may hamper their adoption, leaving coded generators as the only alternative for DSm compilers. In an effort to make the learning and specification of model transformation rules more accessible, Sun et al. introduced Model Transformation By Demonstration (MTBD) in [SWG09]. In their approach, rather than specify LHS and RHS patterns manually, the modeller "records" his actions as he modifies the DSm within the MTBD-enabled framework. Then, an inference engine generates a model transformation rule, specifically its LHS and RHS patterns, that, when run, re-enacts the actions the modeller had recorded. MTBD can thus enable even model transformation novices to very rapidly synthesize DSm-to-artifact compiler prototypes through very accessible modelling activities.

Finally, some researchers have examined the challenges related to fully specifying DSm compilers via model transformations. In [YCDW10], Yie et al. suggest that for complex DSMLs and DSms, building a single "almighty" model transformation may be too difficult. Instead, they argue that an easier and more maintainable solution is to develop a number of smaller model transformation that each focus only on compiling certain parts of DSms. The challenge then becomes enabling the resulting partial artifacts to communicate with each other. Yie et al. achieve this by enforcing naming conventions throughout all model transformations, and via manual adjustments to the artifacts. This last point along with a lack of specific rules and guidelines to determine how to decompose the "almighty" model transformation are the main limitations of their work.

In [HKGV10], Hemel et al. identify a different challenge pertaining to the use of model transformations to produce coded artifacts. They argue that DSm-to-artifact transformations may be plagued by complexities of the target language. For instance, the need to ensure proper indentation of generated Python code[1] can easily drown a rule's specification in accidental complexity. The authors propose the use of model transformations to translate DSms into models of coded artifacts before producing the coded artifacts. These models conform to meta-models which *loosely* reflect the syntax of the programming language (e.g., Java, Python) targeted for artifact generation. Then, another transformation that encapsulates all of the low-level details of the target language produces the artifacts from their models. The main benefit of this technique is the improved modularity of DSm-to-artifact transformations.

2.2.2 Modelling Performance Requirements

Non-functional requirements, often referred to as performance requirements, pertain not to a system's desired functionality (i.e., its functional requirements) but rather to its ability to deliver correct results given certain performance constraints. These constraints may limit such quantities as the amount of bandwidth, memory, power or time that the system may consume to compute and output its result. In extreme cases, failure to meet performance constraints is as bad or worse than producing an incorrect result.

In [TP08], Tawhid and Petriu review past and current research on the benefits of elevating performance concerns to the early stages of development of Software Product Lines (SPLs) [CE00] and propose means to realize the elevation. Their technique consists in annotating UML models with information that enables their subsequent transformation into *performance models*. These performance models lend themselves to analysis and can be used to produce performance predictions. The reasoning behind integrating arguably lower-level non-functional requirement-related concepts so early on in the development process is that it is best to realize as early as possible that these requirements cannot be met by means of the current design. This reasoning is shared by numerous other authors in the *performance*

[1]Incorrect indentation of a Python program can cause it to crash or may alter its meaning.

engineering community.

In [BKR09], Becker et al. present an approach that makes use of MDE techniques to enable performance predictions in the context of component-based software engineering. The Palladio Component Model is a meta-model that allows for performance-relevant information to be specified on models of component-based systems. For instance, a component's time and/or resource requirements can be parameterized by probability mass functions of the input sizes (e.g., number of entries, number of kilobytes) of its arguments. From a network of parameterized components, their framework can produce performance predictions with associated probabilities. Their framework also makes use of model transformations to synthesize basic simulations which issue synthetic demands on resources.

In [KR10, KGHR10], Kapova et al. further combine MDE and SPL techniques. In their approach, target platforms are described by *Feature Diagrams* [KCH+90]. Then, selected features in these diagrams are used to configure model transformation rules that refine application models with target platform specifics. These refined application models are in turn transformed into performance models, which are themselves used to obtain performance predictions.

In summary, considerable attention has been paid to the manual and (semi-)automated enhancement of software models with performance-relevant information, and the subsequent automatic synthesis of performance models and predictions. A crucial problem subsists: the level of abstraction of the development process remains fixed at or near the solution domain. Although certain performance- and platform-relevant details are hidden from developers, their tasks are still performed at the level of code and/or models of solutions. This contradicts both the DSM and MPM philosophies. From DSM and MPM standpoints, current attempts at integrating performance concerns within the development process occur at too low a level of abstraction and entangle too many concerns. Techniques for integrating them with DSms and DSMLs are needed.

2.3 Structured Artifact Synthesis

It is not uncommon for various concerns (e.g., user interface layout, behaviour, performance) to be tangled within a problem domain. Consequently, DSMLs, whose core purpose is to enable the specification of models at the problem domain level, will often reflect this entanglement. The Separation of Concerns (SoC) principle dictates that modularity (and its numerous derived benefits) can be achieved by minimizing the entanglement of concerns. Although the problem domain (and thus its model, the DSML) should arguably not be altered and/or polluted by accidental complexities in the name of this principle, the specification of DSm-to-artifact compilers can indeed be made more modular by its application.

The SoC principle is at the core of the approach presented in this section. We propose to isolate and project the various concerns that make up a domain (and by extension its associated DSML and DSms) onto appropriate lower-level formalisms as an intermediate step to target platform artifact generation. Thus, the complete transformation of DSms into artifacts is composed of numerous modular sub-transformations, each focusing on a single concern. At a high level, our approach for DSm-to-artifact compiler design may be summarized as follows:

1. Identification of the set of concerns that form the DSML;
2. Implementation of one "projection" transformation that compiles DSms into lower-level single-concern models for each identified concern;
3. Implementation of "merging" transformations that recombine the intermediate models into the desired target platform artifact.

We detail and demonstrate our approach in an example-driven manner, describing how DSms of mobile phone applications are iteratively and modularly transformed into intermediate representations, and eventually into fully functional Google Android applications.

2.3.1 PhoneApp: A Multi-Concern DSML

A typical mobile phone application combines three core concerns. The first is its visual interface, which is essentially described by the placement of widgets on the various screens the user must interact with. The second is its behaviour, which is described by the timed (e.g., a welcome screen that disappears after two seconds) and user-prompted (e.g., the click of a button) transitions between the aforementioned screens. The third (and perhaps most domain-specific) encompasses features and functions specific to mobile phone applications (e.g., sending text messages). A DSML for mobile phone applications should capture these three concerns at an appropriate level of abstraction, as does the *PhoneApp* DSML[2]. Its meta-model is shown in Figure 2.1. Essentially, timed and user-prompted transitions describe the flow of control between `Screens` – that can contain various `VisualElements` (i.e., widgets) – and `Actions` – mobile phone device specific features (e.g., sending text messages, dialing numbers).

Without a means to transform DSms into target platform artifacts, DSms can only serve as blueprints and documentation. The traditional approach to artifact synthesis – which is also the one chosen by Kelly and Tolvanen to produce artifacts from DSms for their version of the PhoneApp DSML – to achieve this transformation is via hand-coded text generators. In our approach, however, a series of rule-based graph transformations "compile" PhoneApp models into increasingly lower-level (i.e., closer to the target platform) formalisms until a fully functional Google Android application is synthesized. Figure 2.2 depicts the

[2]The PhoneApp meta-model is heavily inspired by the meta-model for modelling mobile phone applications presented by Kelly and Tolvanen in [Met, KT08]. Our contribution lies in the means used to produce artifacts from PhoneApp models rather than in the definition of the language's meta-model.

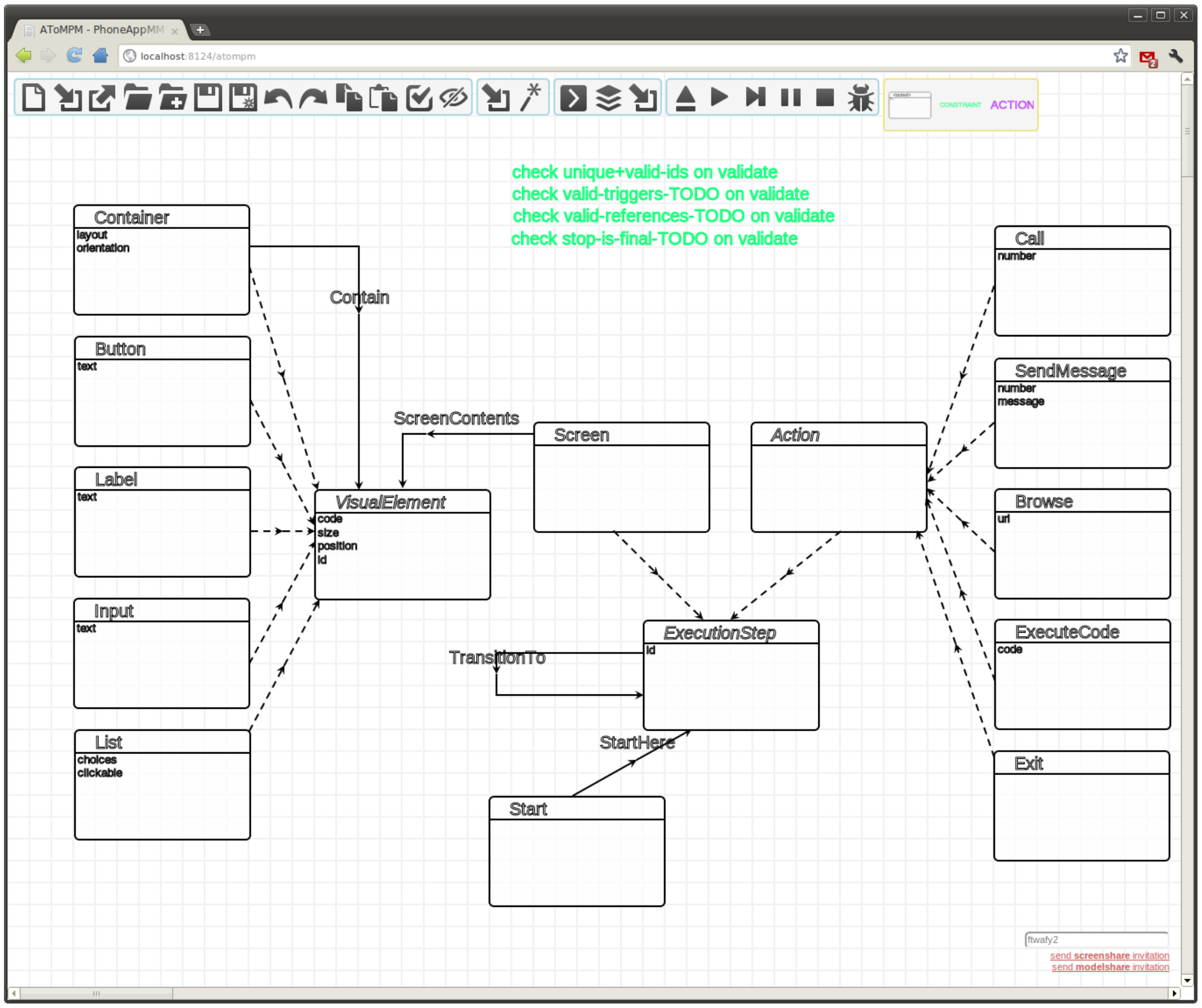


Figure 2.1: The *PhoneApp* meta-model.

relationships between the involved formalisms, with arrows between them designating model transformations. First, PhoneApp models are projected onto three intermediate models that each capture one of three domain concerns. Then, these lower-level models are merged together to form the target platform artifact. The distinction between the *FileSystem* and *Disk* formalisms will be explored later.

As depicted in Figure 2.2, projecting the three constituting concerns tangled within PhoneApp models produces disjoint instances of different formalisms, and as such, the order in which these projections take place is of no consequence. In practice, however, an order may be fixed to facilitate implementation and debugging, or due to tool limitations. Figure 2.3 depicts *T_PhoneApp2Disk(Android)*, the top-level model transformation that compiles PhoneApp models into Google Android applications, specified in AToMPM. Note that an order has indeed been imposed for the projection and merging transformations. The for-

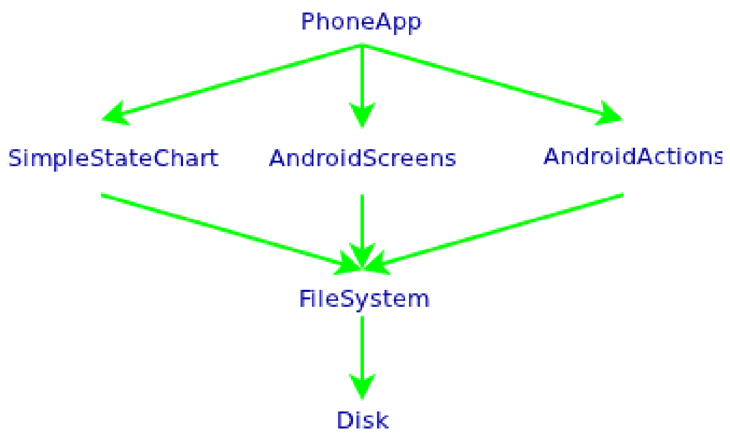


Figure 2.2: Formalism Transformation Graph [dLVA04] for PhoneApp.

malisms and transformations introduced in Figures 2.2 and 2.3 are described in the following sub-sections.

Note that the PhoneApp DSML is one of many possible DSMLs for modelling mobile phone applications. Some alternatives may include provisions for using touchscreen gesture, gyroscope and camera data via new modelling constructs. Others may be tailored to very specific application domains (e.g., health care, social computing) and include higher-level abstractions[3]. Finally, others may require lower-level concepts such as communication protocols to be exposed. In either case, new model transformation rules and/or entire model transformations may be required to isolate and compile added concepts and concerns.

2.3.2 Isolating Behaviour

The behaviour of a mobile phone application is inherently state-based: control flows between disjoint application screens. In the PhoneApp DSML, these states are modelled as `ExecutionSteps`, and the flow between them is fully described by event- and timeout-triggered transitions. Thus, a natural mapping exists between the behaviour described by a PhoneApp model and a Statechart model[4] [Har87]. The "first" step in the synthesis of executable applications from PhoneApp models is the isolation of their behavioural components and their subsequent projection onto a behaviourally equivalent Statechart model.

The isolation and projection tasks are accomplished by the *PhoneApp-to-Statechart* model transformation, or *T_PhoneApp2SimpleStateChart* as it is referred to in Figure 2.3,

[3]We demonstrated this in [DLL+09] where we built *SecureApps*, a DSML for modelling privacy preserving applications. We later transformed SecureApps models into PhoneApp models (which were thus at a lower-level of abstraction) as an intermediate step to artifact generation.

[4]Note that the current range of possible behaviours of PhoneApp models requires only the expressiveness of timed automata. Future work might extend the formalism with notions of hierarchy and concurrency such that more powerful Statechart features (e.g., orthogonality, nesting) become required.

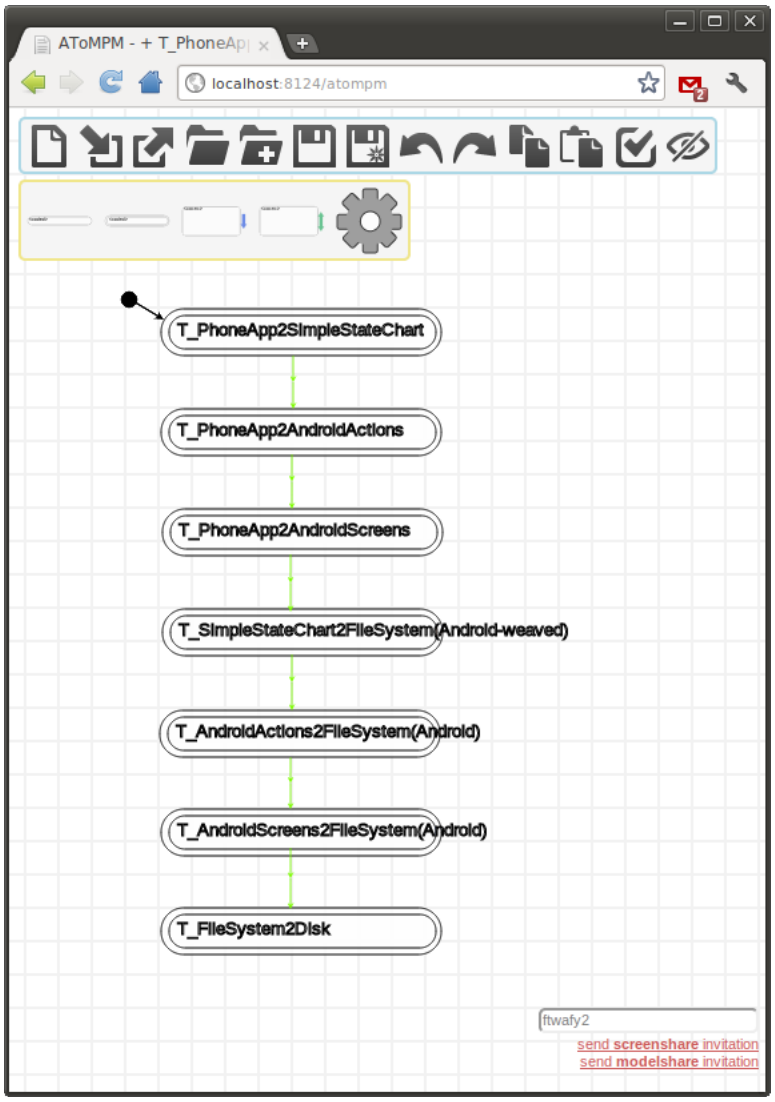

ATоMPM - + T_PhoneAp
localhost:8124/atompm
T_PhoneApp2SimpleStateChart
T_PhoneApp2AndroidActions
T_PhoneApp2AndroidScreens
T_SimpleStateChart2FileSystem(Android-weaved)
T_AndroidActions2FileSystem(Android)
T_AndroidScreens2FileSystem(Android)
T_FileSystem2Disk
ftwafy2
send screenshare invitation
send modelshare invitation

Figure 2.3: The *T_PhoneApp2Disk(Android)* transformation.

depicted in Figure 2.4. Three rules are sequenced, with the first and second both being repeatedly reapplied until they are no longer applicable before moving on to the next. This "looping" is captured by the green and grey inter-rule links, which respectively depict the flow of control after a successful rule application and after a failure to meet a rule's pre-condition. Figure 2.5 shows these rules in more detail. First, the *R_ExecutionStep2State* rule creates a Statechart `State` for every PhoneApp `ExecutionStep`[5] (i.e., for every `Screen` and `Action`).

[5]The "for every" notation is abusive. In reality, the *R_ExecutionStep2State* rule only creates one Statechart `State` for one PhoneApp `ExecutionStep`. It is the recursive loop onto itself, defined by the rule scheduling (or model transformation) model shown in Figure 2.4, that effects the actual re-application of the rule, while the rule's NAC ensures that the rule is never applied more than once on the same `ExecutionStep`, incidentally also ensuring that the rule will eventually cease to be applicable.

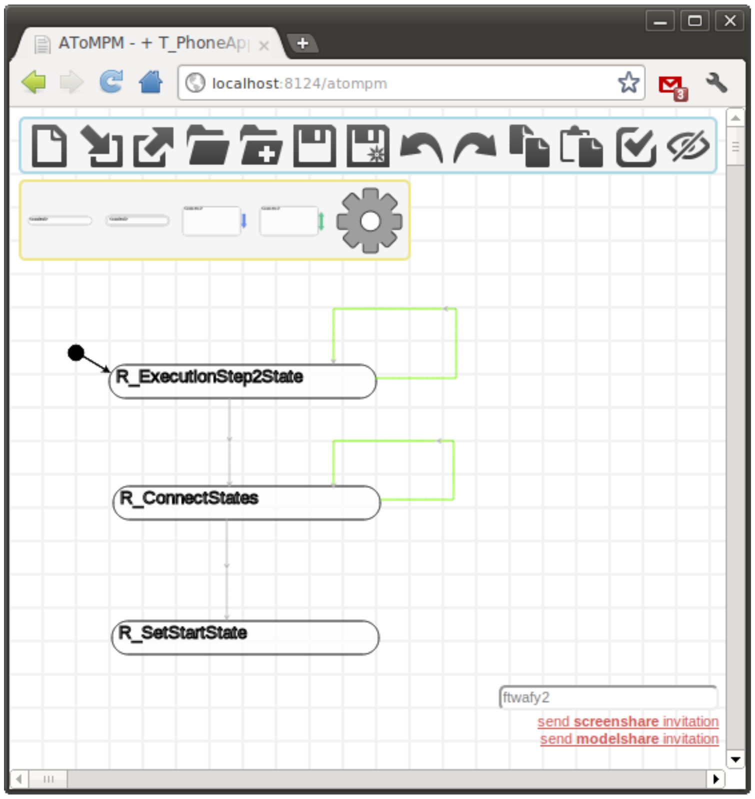
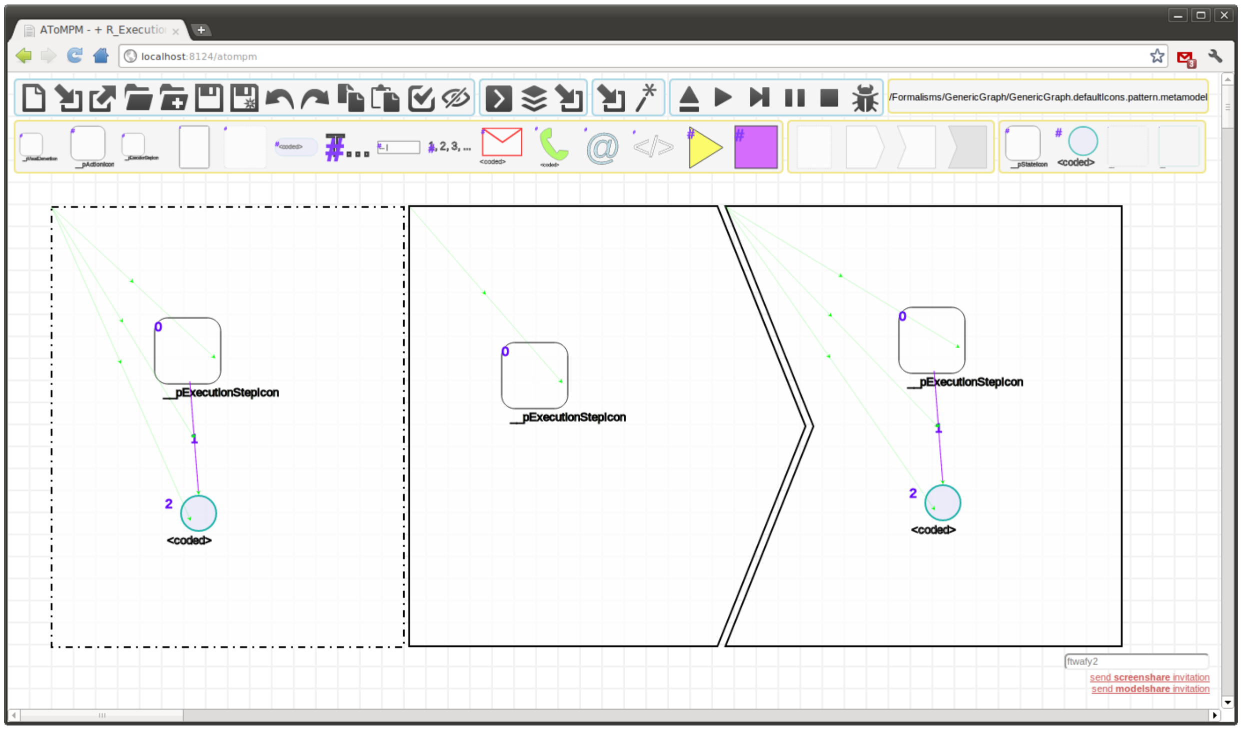


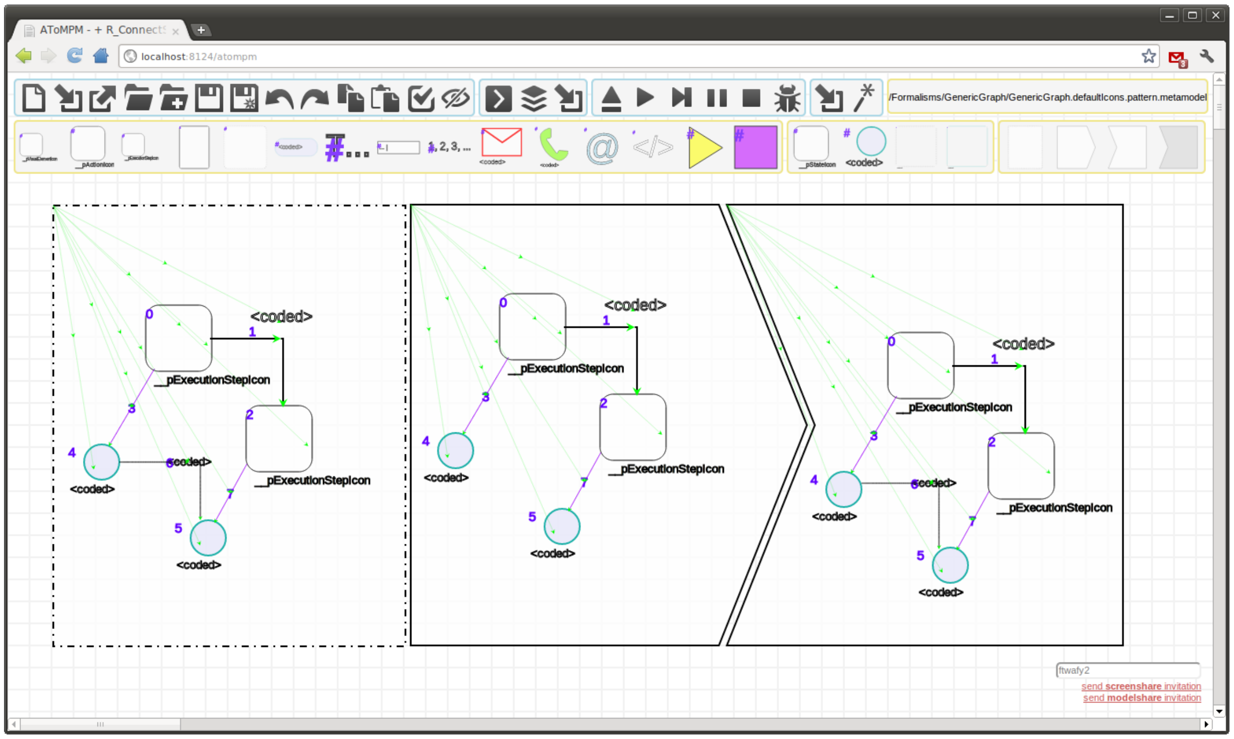
Figure 2.4: The *PhoneApp-to-Statechart* model transformation.

Note the intuitive and non-intrusive use of *generic links* to establish explicit correspondences between constructs from different formalisms and at different levels of abstraction. Second, the *R_ConnectStates* rule connects the newly created `States` via Statechart `Transitions` such that for every pair of connected `ExecutionSteps`, the corresponding pair of `States` are also connected. Note how simple and natural the aforementioned use of generic links makes the specification of cross-formalism correspondences in rule patterns. This, as opposed to the manipulation of some sort of internal, possibly id-based, construct mapping in rule condition and action code. The newly created `Transitions` are parameterized to reflect the properties of the edges connecting their `ExecutionSteps`: events and timeouts specified at the DSm level are appropriately altered to conform to event and timeout syntax from the Statechart formalism. Third and last, the *R_SetStartState* rule matches the starting `ExecutionStep`, identified by an incoming connection from a PhoneApp `Start` construct, and its corresponding `State`, and alters that `State`'s `isStart` attribute.

When the *PhoneApp-to-Statechart* transformation has run its course, every PhoneApp `ExecutionStep` has a corresponding Statechart `State`. These are connected via customized Statechart `Transitions` in a manner that reflects the edges that connect their corresponding `Screens` and `Actions`. Traceability links connect each construct in the generated Statechart model with its corresponding construct in the PhoneApp model. A focal point of interest

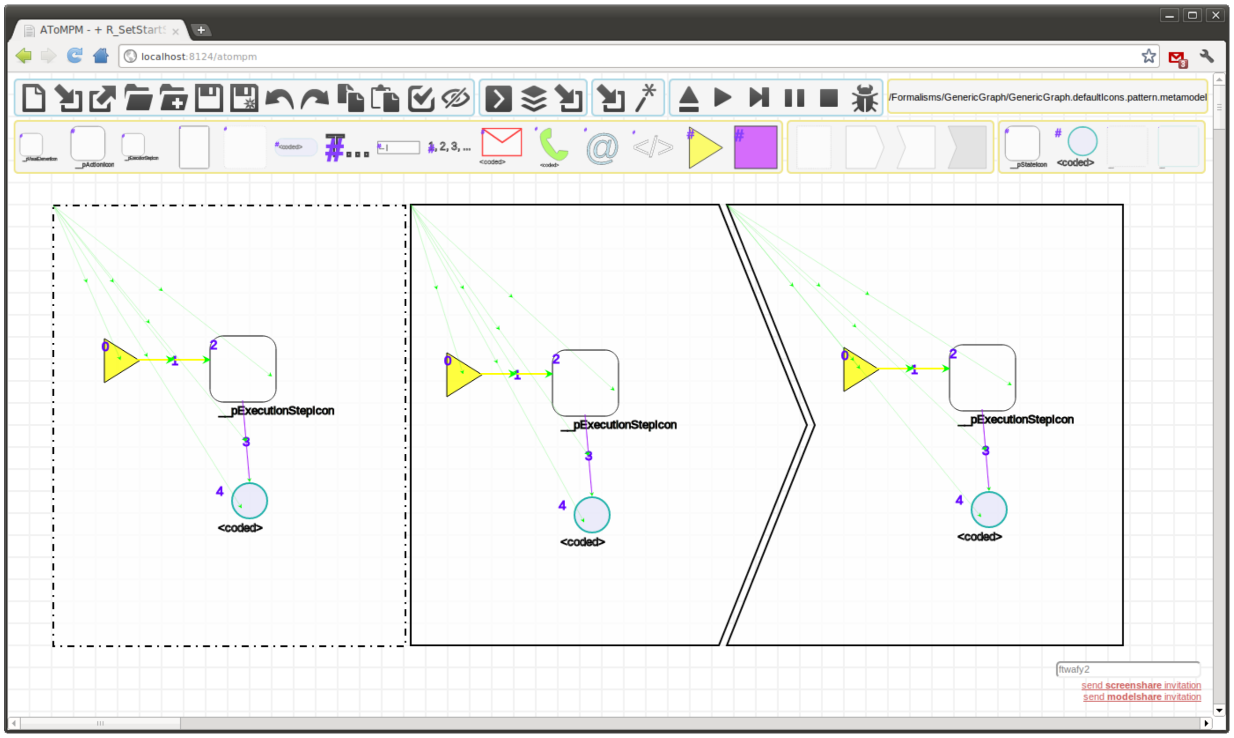

AToMPM - + R_Executio
localhost:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
0
__pExecutionStepIcon
1
2
<coded>
ftwafy2
send screenshare invitation
send modelshare invitation

(a) The *R_ExecutionStep2State* rule.


AToMPM - + R_ConnectS
localhost:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
0
1
2
3
4
5
6
7
<coded>
__pExecutionStepIcon
ftwafy2
send screenshare invitation
send modelshare invitation

(b) The *R_ConnectStates* rule.

Figure 2.5: The rules that compose the *PhoneApp-to-Statechart* model transformation *(continued)*.


ATоMPM - + R_SetStartS
localhost:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
__pExecutionStepIcon
<coded>
0
1
2
3
4
send screenshare invitation
send modelshare invitation

(c) The *R_SetStartState* rule.

Figure 2.5: The rules that compose the *PhoneApp-to-Statechart* model transformation.

here is that no information pertaining to the nature and placement of widgets within the `Screens` or to `Action` parameters appears in the synthesized Statechart representation: the *PhoneApp-to-Statechart* transformation indeed truly projects *only* the behavioural concerns of the DSm.

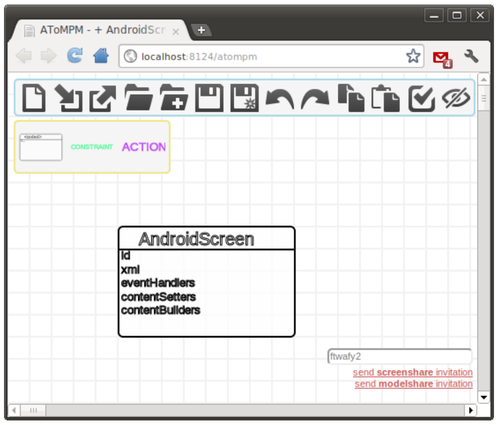
On a more general note, although the proven and studied Statechart formalism is indeed an appropriate target formalism for the projection of behaviour in this context (i.e., for the modelling of reactive state-based behaviour), it may not be optimal for or even capable of capturing other systems' behaviour. For instance, for a traffic network DSML describing the non-deterministic flow of vehicles across connected road segments, it may be more appropriate for behaviour to be projected onto a formalism such as Petri Nets [Pet81]. Thus, the proposed approach suggests not to systematically isolate and project behavioural concerns in DSms onto Statechart models, but rather to isolate and project them onto semantically appropriate behaviour models, with little regard for their actual formalism.

2.3.3 Isolating Layout

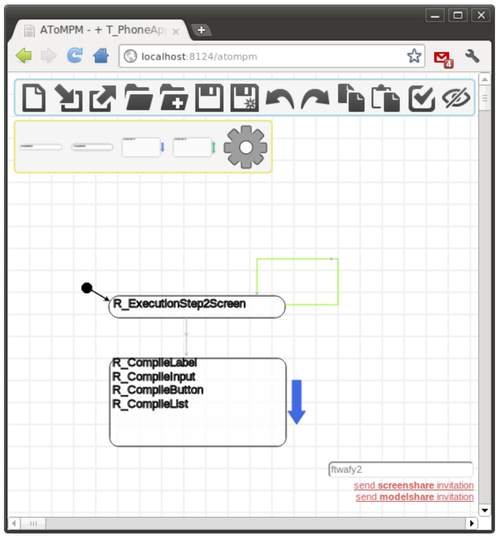
The layout concern of a mobile phone application is captured by widget placement within application screens. In the PhoneApp DSML, screens are modelled as `Screens`, and widgets as `VisualElements`. The formalism we introduce as a target for the projection of the layout concern is *AndroidScreens*. Its simple meta-model is shown in Figure 2.6a. A single construct, the `AndroidScreen`, is parameterized by snippets of Google Android-specific code for rendering a screen, configuring and initializing its widgets according to modeller specifications, and properly handling events resulting from end-user interaction with clickable and/or editable widgets.

The layout semantics of PhoneApp models are fully encompassed within the contents and parameters of `Screens`. The isolation and projection of these semantics are accomplished by the *PhoneApp-to-AndroidScreens* model transformation, or *T_PhoneApp2AndroidScreens* as it is referred to in Figure 2.3, depicted in Figure 2.6b. Fives rules are sequenced. The first is repeatedly reapplied until it is no longer applicable. The next four rules are bundled in a scheduling construct that only terminates when all of the bundled rules are no longer applicable[6]. Figure 2.7 shows two of the aforementioned rules. First, the *R_ExecutionStep2Screen* rule creates an AndroidScreens `AndroidScreen` for every PhoneApp `Screen`. The newly created `AndroidScreens` are also initialized with screen rendering boilerplate code (via their `xml` attribute). Note again the use of generic links to enable traceability and facilitate establishing correspondences. Second, the *R_CompileLabel* rule translates a matched PhoneApp `Label` into appropriate alterations to the relevant `AndroidScreen`'s `xml`, `contentSetters` and `contentBuilders` attributes. These respectively encompass widget rendering, static widget initialization (i.e., setting the label text to a modeller specified value) and dynamic

[6]A detailed description of the rule scheduling formalism provided by AToMPM and used in the various depictions of model transformations throughout this work is included in its User's Manual [Man].



(a)

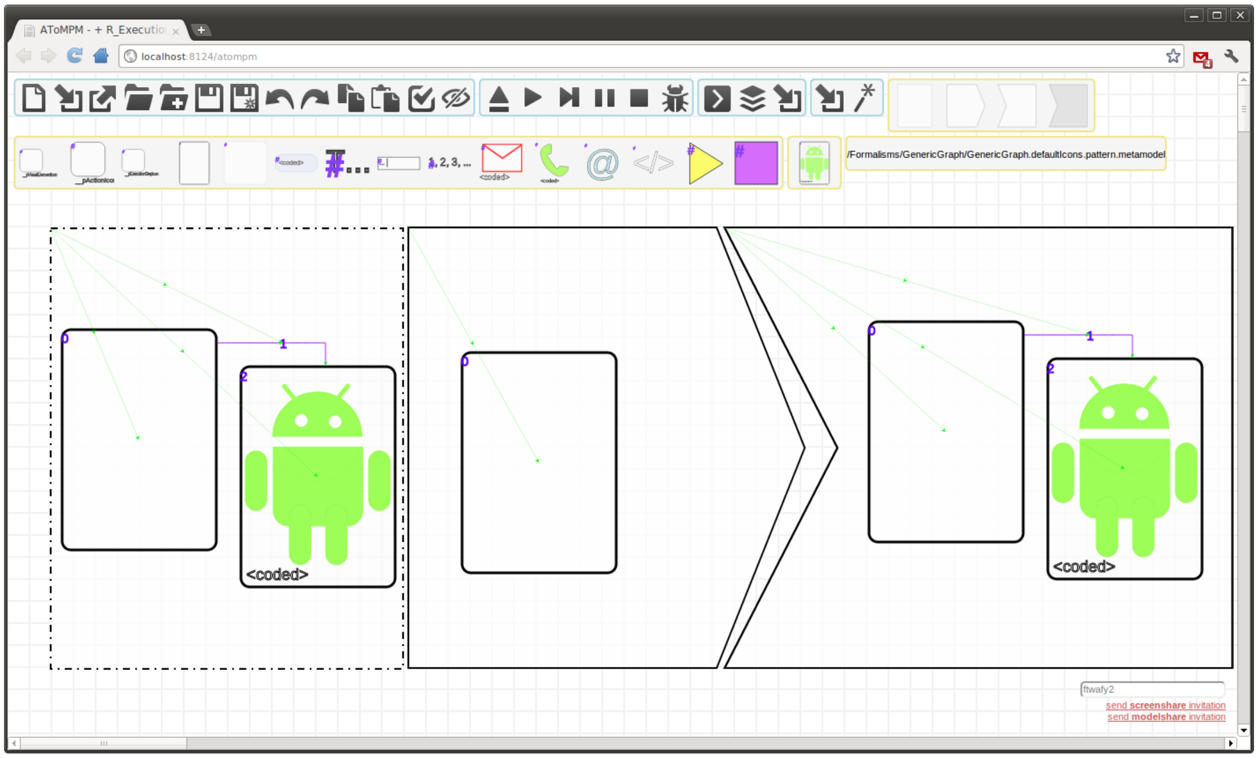


(b)

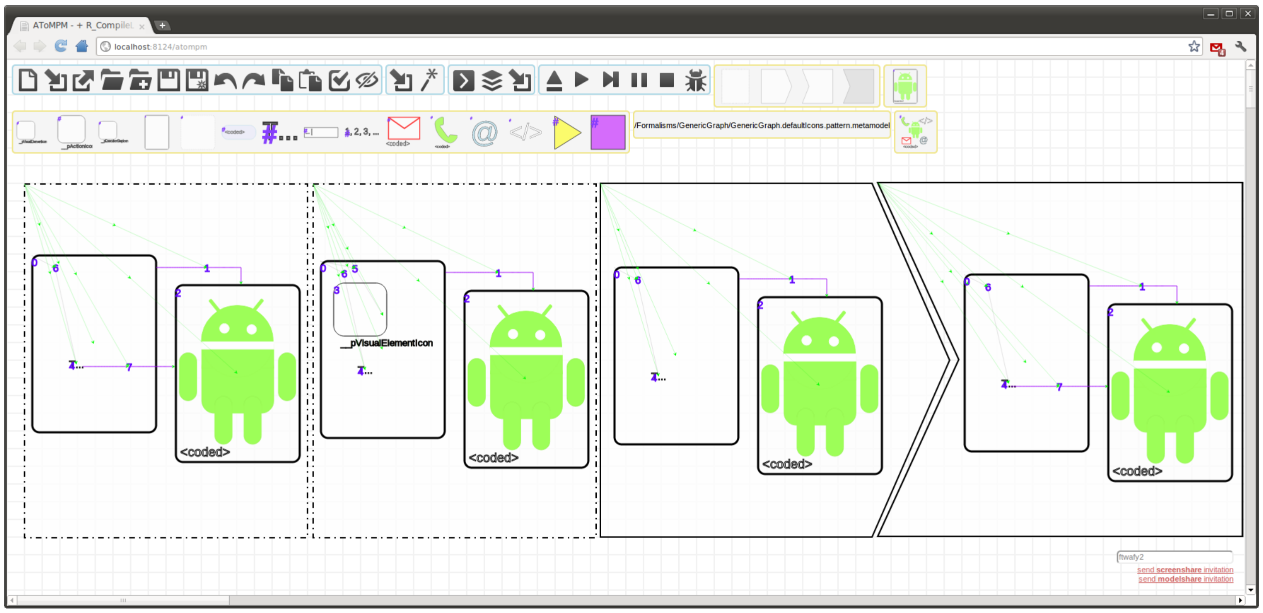
Figure 2.6: (a) The *AndroidScreens* meta-model. (b) The *PhoneApp-to-AndroidScreens* model transformation.

widget configuration (i.e., evaluating modeller-specified action code. if any, to set label text at runtime). Note that the `eventListeners` `AndroidScreen` attribute is left unchanged. This is because labels do not produce events. The *R_CompileLabel* rule has two NACs. The leftmost one is used to ensure the same PhoneApp `Label` is not matched more than once. The other is used to ensure `VisualElements` are matched in an order that reflects their positioning on the `Screen` (e.g., given a vertical layout, `VisualElements` at the top of a `Screen` are matched first). The *R_CompileButton*, *R_CompileInput* and *R_CompileList* rules are not shown due to their strong similarity with the *R_CompileLabel* rule.

When the full *PhoneApp-to-AndroidScreens* transformation has run its course, `Screens` each have corresponding and appropriately parameterized instances of `AndroidScreens`. Traceability links connect each construct in the generated AndroidScreens model with its corresponding construct in the PhoneApp model. Note that no information pertaining to any of the two other concerns appears in the AndroidScreens representation. Everything from the PhoneApp model that pertained to behaviour and mobile phone features has been stripped away. Nevertheless, the generated Statechart model produced by the *PhoneApp-to-Statechart* transformation is not entirely semantically disjoint from the generated AndroidScreens model produced here. They are linked by the fact that PhoneApp `Screens` are mapped to both a Statechart `State` and an AndroidScreens `AndroidScreen`. The reconciliation of these correspondences will be discussed in Section 2.3.5 when the three intermediate representations of PhoneApp models are woven back together into target platform artifacts.


AToMPM - + R_Executio
localhost:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
<coded>
ftwafy2
send screenshare invitation
send modelshare invitation

(a) The *R_ExecutionStep2Screen* rule.

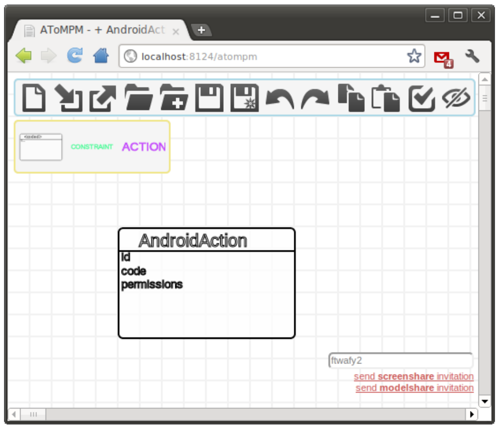

AToMPM - + R_Compile
localhost:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
__pVisualElementIcon
<coded>
ftwafy2
send screenshare invitation
send modelshare invitation

(b) The *R_CompileLabel* rule.

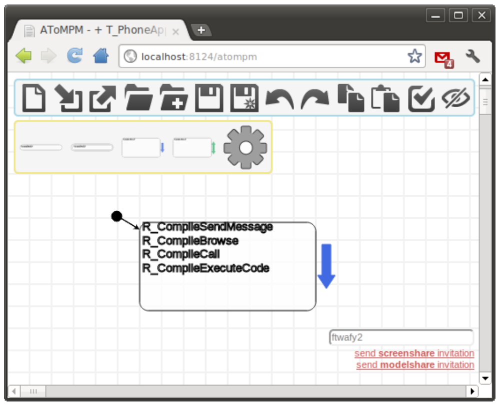
Figure 2.7: Two rules that compose the *PhoneApp-to-AndroidScreens* model transformation.

2.3.4 Isolating Mobile Phone Features

Features specific to mobile phone applications include making phone calls, sending text messages and launching native smartphone applications such as browsers. In the limit, one might also argue that custom code snippets that make use of smartphone APIs also fit into this category. Although there are many more such features, in its current state, the PhoneApp DSML only supports these four. They are respectively modelled by the `Call`, `SendMessage`, `Browse` and `ExecuteCode` constructs. The formalism we introduce as a target for the projection of this final concern is *AndroidActions*. Its trivially simple meta-model is shown in Figure 2.8a. A single construct, the `AndroidAction`, is parameterized by snippets of Google Android-specific code that carry out the desired feature, and by a list of required permissions (e.g., access to the contacts list, access to geographic location).



(a)



(b)

Figure 2.8: (a) The *AndroidActions* meta-model. (b) The *PhoneApp-to-AndroidActions* model transformation.

The mobile phone feature semantics of PhoneApp models are fully encompassed within `Actions`. The isolation and projection of these semantics are accomplished by the *PhoneApp-to-AndroidActions* model transformation, or *T_PhoneApp2AndroidActions* as it is referred to in Figure 2.3, depicted in Figure 2.8b. Four rules are bundled in the same manner as those from Figure 2.6b. Figure 2.9 shows one of them. The *R_CompileBrowse* rule creates an AndroidActions `AndroidAction` for every PhoneApp `Browse` and sets its `code` attribute to a Google Android code snippet that launches a browser and loads the modeller-specified URL. The `permissions` attribute is left empty because no special permissions are required to perform the said operation. The *R_CompileSendMessage*, *R_CompileCall* and *R_CompileExecuteCode* rules are not shown due to their strong similarity with the *R_CompileBrowse* rule.

Once the *PhoneApp-to-AndroidActions* transformation has run its course, each `Action` has an appropriately parameterized associated `AndroidAction`. As was the case for both

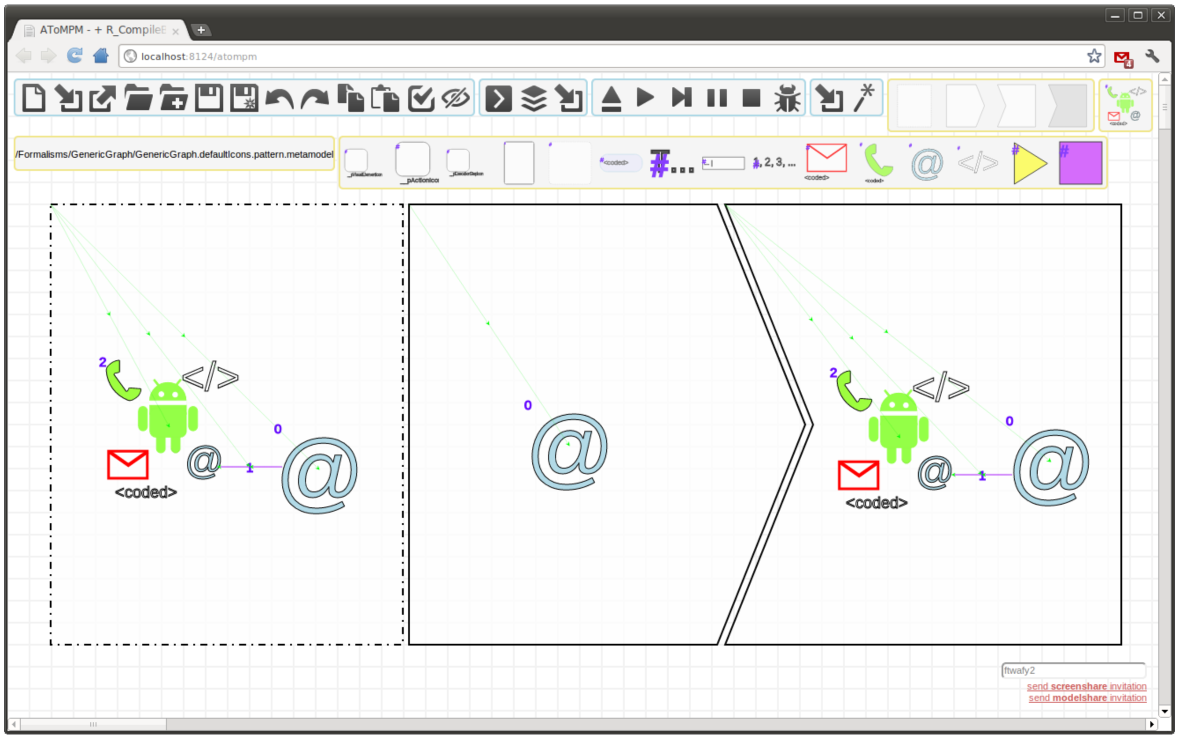

AToMPM - + R_CompileE
localhost:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
<coded>
ftwafy2
send screenshare invitation
send modelshare invitation

Figure 2.9: The *R_CompileBrowse* rule, from the *PhoneApp-to-AndroidActions* model transformation.

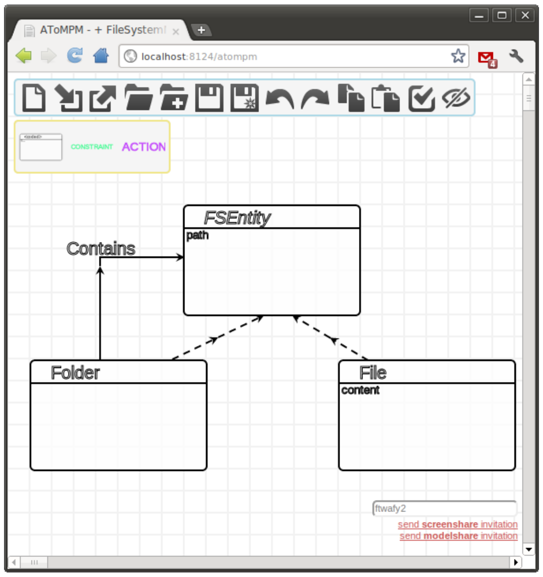


Figure 2.10: The *FileSystem* meta-model.

previously introduced projection transformations, only the desired concern is captured by the synthesized intermediate representation and traceability links are left behind between the DSm and newly generated constructs. Lastly, conceptual correspondences also exist between AndroidAction `AndroidActions` and Statechart `States` that result from the same PhoneApp `Actions`.

2.3.5 Merging intermediate representations

A side-effect of our approach of projecting DSms onto various intermediate representations is that these projections eventually need to be recombined to produce the final target platform artifacts. This is especially relevant when certain constructs in the DSm are mapped to constructs in more than one intermediate representation, as is the case in the running example.

Keeping with the mindsets of modularity and traceability, we introduce a final intermediate formalism, *FileSystem*, as a target for the merging of the generated Statechart, AndroidScreens and AndroidActions models. Its meta-model is shown in Figure 2.10. This formalism serves as a simple abstraction of files and folders on disk. In short, rather than output the results of the merging process directly to files, they are first represented as an instance model of the FileSystem formalism. Beyond the eased maintenance of traceability links between the `Files` and `Folders` that make up the generated FileSystem model and their source constituents (i.e., model entities from the three intermediate representations), an added benefit of this design choice is that the generated contents for each (future) file can be reviewed from within the modelling environment as part of the debugging process. This, as opposed to having to locate generated files on disk, opening them in a separate editor, and possibly having to return to the model editor to perform changes. Another crucial benefit is that `File` contents may be produced out of order, thereby considerably simplifying the intermediate representation merging process. For instance, easily matched "tags" can be inserted into synthesized contents to be replaced by subsequent rule applications. Though such flexibility would also be achievable even if the *FileSystem* formalism were to be bypassed, it would come at a high cost as numerous disk I/O operations – often the slowest operation on modern computers – would be required for each rule. Finally, note that if a finer level of traceability is required, *contents* may be an instance of an explicitly meta-modelled programming language, as proposed by Hemel et al. in [HKGV10].

A sensible and, most of all, generic approach to take for the realisation of the merging process is for the intermediate representation that captures behaviour to become the *main* executable artifact. This artifact should not only implement the desired behaviour, it should also be instrumented to make appropriate use of artifacts that capture other concerns. In the PhoneApp example, both of these aspects are captured by three transformations: *Statechart-to-FileSystem*, *AndroidScreens-to-FileSystem*, and *AndroidActions-to-FileSystem*, or *T_SimpleStateChart2FileSystem(Android-weaved)*, *T_AndroidScreens2FileSystem(Android)* and *T_AndroidActions2FileSystem(Android)* as they are respectively referred to in Figure 2.3. Each of these is shown in Figure 2.11 and described below. Before moving on to a more detailed discussion, notice the first rules of each transformation. If nothing else, that of the first implies the initialization of some sort of a "manifest" while that of the second and third rather imply the verification of the existence of a "manifest". As argued earlier, fixing the order in which transformations occur may lead to possible optimizations, often at the cost of flexibility. In this case, fixing the ordering releases us from having to prepend boilerplate to verify if the "manifest" has been initialized and to do so if not at the start of each transformation. The cost of course is that running the transformations out of order will not produce the desired target platform artifacts.

The first merging transformation, *Statechart-to-FileSystem*, produces four `Files`: *App.java*, *AppLib.java*, *AndroidApp.java* and *AndroidManifest.xml*. The first contains Google Android-aware Java code that captures the behaviour described by the Statechart generated from the PhoneApp model. The second is meant as a repository for code that other artifacts might wish to make accessible to the compiled Statechart. The last two files contain generic boilerplate required for all Google Android applications. Another crucial task performed by this transformation, more specifically by each application of the *R_CompileState* rule, is the aforementioned instrumentation that enables the compiled Statechart to interact with other artifacts. The *entry action* of every compiled `State` is populated with method calls that execute the rendering of a `Screen` (`renderScreen`$<$ *ScreenId* $>$`()`) or the execution of an `Action` (`runAction`$<$ *ActionId* $>$`()`) depending on if that `State` corresponds to a PhoneApp `Screen` or `Action`, with correspondences trivially determined by following generic links. Further implementation details regarding the compilation of Statechart models to Java are left to the reader which we refer to Harel's and Kugler's description of the semantics of (Rhap-

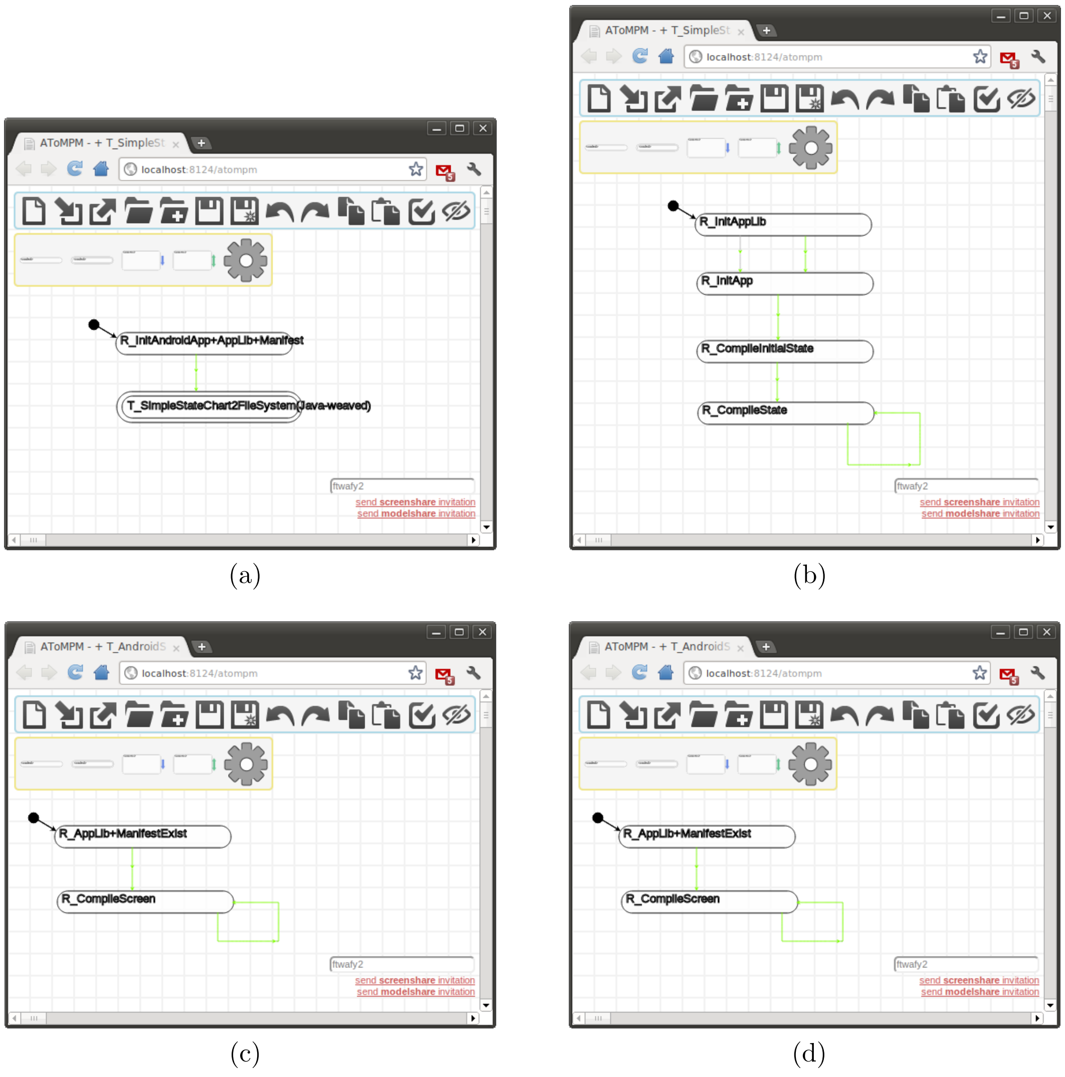


Figure 2.11: The *StateChart-to-FileSystem* (a-b), *AndroidActions-to-FileSystem* (c) and *AndroidScreens-to-FileSystem* (d) model transformations.

sody) Statecharts in [HK04].

Next, the *AndroidScreens-to-FileSystem* and *AndroidActions-to-FileSystem* transformations are executed to produce Statechart-oblivious target platform artifacts. The former transformation is two-fold. First, each `AndroidScreen` triggers the creation of a new `File` containing XML code that embodies widget placement. This XML code is taken literally from the `AndroidScreen`'s `xml` attribute. Second, a Java method, `renderScreen`$<$ *ScreenId* $>$`()`, that carries out widget content and event handler setup is inserted into the *AppLib.java* `File` for each `AndroidScreen`. Its body is constructed from the `AndroidScreen`'s `eventHandlers`, `contentSetters` and `contentBuilders` attributes. Notice that the new method name fol-

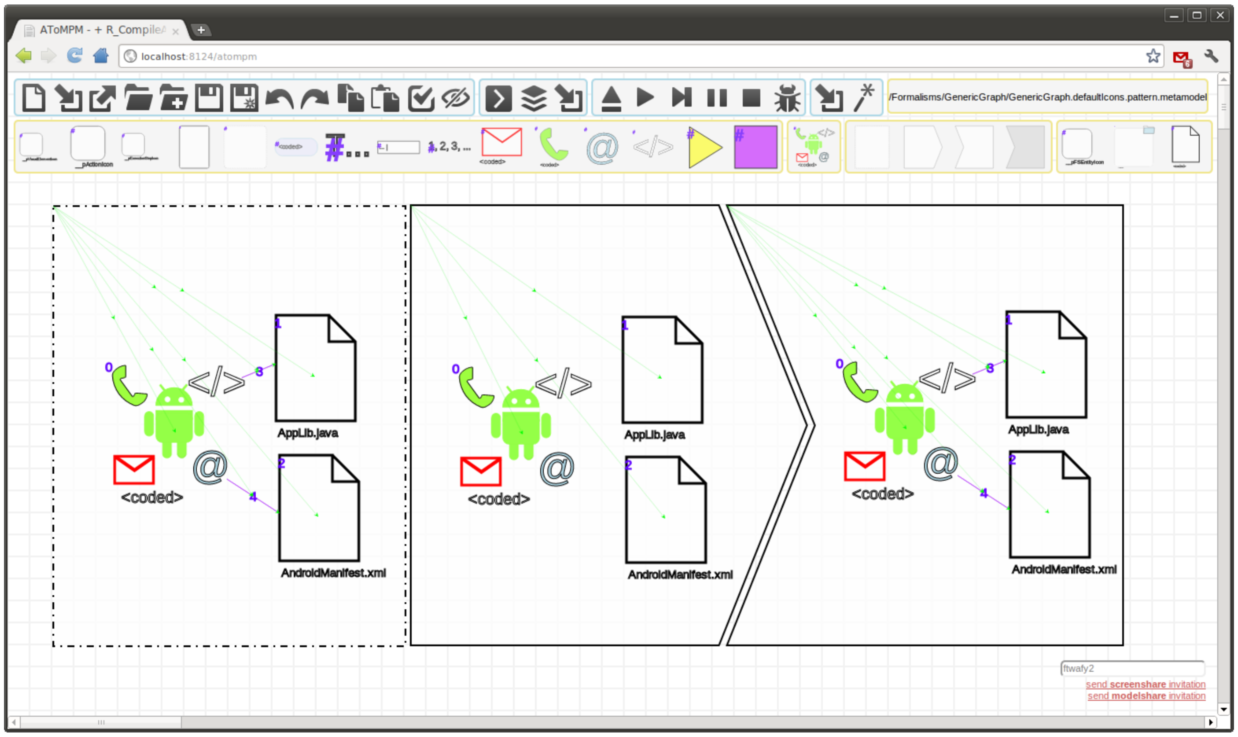

AToMPM - + R_CompileA
localhost:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
0
1
2
3
4
<coded>
AppLib.java
AndroidManifest.xml
ftwafy2
send screenshare invitation
send modelshare invitation

Figure 2.12: The *R_CompileAction* rule, from the *AndroidActions-to-FileSystem* model transformation.

lows the same convention as those that were previously inserted into the compiled Statechart's entry actions. Enabling artifact communication in such a name-based manner is not unlike Yie et al.'s work in [YCDW10]. As for the *AndroidActions-to-FileSystem* transformation, similarly, each `AndroidAction` results in a Java method, `runAction`$< ActionId >$`()`, populated with the contents of that `AndroidAction`'s `code` attribute. These are also inserted into the *AppLib.java* `File`. Additionally, any permissions the action might require, specified in the `AndroidAction`'s `permissions` attribute, are inserted into the *AndroidManifest.xml* `File`. Figure 2.12 shows *R_CompileAction*, the rule that carries out these tasks.

Once the *Statechart-to-FileSystem*, *AndroidScreens-to-FileSystem*, and *AndroidActions-to-FileSystem* transformations have run their course, a number of `Files` have been generated. One of them, *App.java*, contains the compiled Statechart model. Another, *AppLib.java*, contains method definitions for rendering screens and carrying out mobile phone actions. *AndroidApp.java* and *AndroidManifest.xml* contain Google Android boilerplate and permission requirements. The remainder (one for each `AndroidScreen`) contain XML layout code. The final step of the artifact synthesis process is for physical files to be output from these `Files`. This is straightforward as `Files` are parameterized by paths and contents, which is all one needs to create equivalent files on disk. Upon completion, the physical files can be loaded onto a Google Android-enabled device and executed.

The entire process of artifact synthesis from DSms has been introduced. Tangled concerns within DSms are isolated and projected onto lower-level (i.e., closer to the target platform) formalisms in a modular fashion. Then, the generated instances of these intermediate formalisms are woven back together to reflect the intended semantics of the source DSms. Finally, the result is output to disk to form the desired target platform artifacts. Moreover, the many model transformations involved leave behind a network of traceability links between corresponding constructs at different levels of abstraction, from DSms to target platform artifacts (and back). The following sub-section details the benefits of this approach over the traditional text generator approach to artifact synthesis.

2.3.6 Benefits of Modular Artifact Synthesis

The motivations for our approach were the need to address the poor maintainability and extensibility of coded text generators, as well as to raise their low level of abstraction. In the following, we explain how our approach improves on text generators in these three areas.

The most important advantage of our approach is that it raises the level of abstraction and modularity of DSm compilers. Whereas in the traditional approach, their development includes interaction with internal model representations and the writing of complex code, in our approach, the task of implementing an artifact generator is reduced to specifying relatively simple (graphical) model transformation rules that interact with model entities as they are presented to modellers. Furthermore, the layered nature of our approach (i.e., the existence of intermediate representations between DSm and artifacts) enables low-level (e.g., target platform) details to be hidden within lower-level transformations (i.e., intermediate representation to artifact transformations) rather than included in higher-level, DSm-to-artifact transformations. On the one hand, from an MPM perspective, our approach to artifact synthesis seems like both a natural and logical improvement. On the other, it is especially effective at limiting the scope of required maintenance brought on by external evolution. For instance, if a domain evolves, intermediate representation to artifact transformations need not be co-evolved.

Another benefit is that the multiple intermediate layers between DSm and artifact provide a means to observe models from various viewpoints. For instance, in the context of DSms of mobile phone applications, to study only the behavioural aspects of a model, one may study the generated Statechart model in isolation from the DSm and the other generated artifacts. More generally, a developer who wishes to focus his attention on a single concern without being distracted by irrelevant (from the point of that concern) details can easily do so. This would be an arduous task in the traditional approach where no intermediate representations are available.

The remaining benefits of our approach result from the network of traceability links it creates between corresponding constructs at different levels of abstraction. In the past, the generation of such traceability information was included within existing text generators, thereby inevitably reducing their modularity and polluting them with added accidental complexity. In contrast, our approach performs the complex task of maintaining correspondence links between DSms and synthesized artifacts by explicitly connecting higher-level entities to their corresponding lower-level entities in transformation rules via generic links. These edges have minimal impact on the readability of the rules – one could even argue that they improve their readability by clarifying correspondences – and their specification can be automated.

The first of many "advanced" tasks that are made possible by traceability links is DSm animation as a result of artifact execution. Basic commands can be exchanged between synthesized artifacts (as they run) and the model editing tool. These are propagated throughout the network of traceability links to animate the model. Keeping with the running example, entering an application screen (which corresponds to the entry into a compiled Statechart `State`) can be made to emit an appropriately parameterized highlighting command (`highlight:`*StateId*). A single transformation rule suffices to instrument compiled state entry actions with code snippets that send such commands to the model editor. This rule can easily be enabled or disabled and does not affect any of the other rules. Thus, its impact in terms of accidental complexity is minimal. Once the model editor receives the highlighting command, the relevant AndroidScreens `AndroidScreen`, PhoneApp `Screen` and Statechart `State` entities are highlighted, with each correspondence (i.e., from `State` to `Screen` to `AndroidScreen`) resolved by navigating the network of traceability links. We have prototyped this in our implementation of the running example in AToMPM[7] and demonstrate artifact to DSm feedback in Section 2.5.

Another "advanced" task that is greatly simplified by the presence of traceability links is the debugging of DSms, artifacts and model transformations. The numerous reasons for this are explored in detail in Chapter 3. One of them is that the debugging of denotational semantics is facilitated as clear links depict "what is generated from what". Without these, determining which statement of a coded generator produced which faulty bit of output, or which (erroneous) model entity triggered a transformation rule is often far from straightforward.

Last but not least, although in a finished product the inner workings that synthesize artifacts from DSms should be hidden from the modeller, it may often be useful for didactic purposes to see how higher- and lower-level constructs are related. Both our simple and modular transformation rules and the cross-level links they produce make these relationships accessible and explicit.

[7]AToMPM's API for inbound commands from artifacts is fully described in its User's Manual.

2.4 DSms and Performance Concerns

This section is broken into two parts. The first explores how structured artifact synthesis from DSms can improve upon the state-of-the-art in performance modelling in the context of elevating non-functional requirements and performance concerns to the early stages of development. The second builds on the first to examine how widespread performance assessment methods are realized within the proposed approach.

2.4.1 Revisiting Performance Modelling

The approach to artifact synthesis presented in the previous section can also be instrumental in the context of modelling (and synthesizing) embedded software. More specifically, it helps address non-functional requirements. Keeping with the running example, although the Google Android platform abstracts away numerous traditional embedded systems concerns such as task scheduling, PhoneApp models remain models of embedded system applications. Thus, timing and resource utilization information may be relevant and even required by modellers. More generally, there are numerous scenarios where domain-specific modellers may require information regarding the *performance*[8] of artifacts synthesized from their models. A common, more code-centric approach for addressing this need is for model transformations to refine *annotated* software models into *performance models* from which simulations and performance predictions are produced. The application of DSM and MPM principles, and specifically of the proposed approach to artifact synthesis, can improve that technique in a number of ways. Before discussing these, enhancements to the formalism transformation graph and top-level model transformation presented in Figures 2.2 and 2.3 must be made in the context of adding a performance modelling dimension to the PhoneApp running example. These enhancements are presented in Figures 2.13 and 2.14. The added formalisms and transformations are introduced throughout this section.

PerformanceModel is introduced as a new intermediate formalism between PhoneApp models and Google Android applications[9]. Its meta-model is depicted in Figure 2.15. `ResourceConsumer`s are interconnected via `ResourceConsumerConnector`s. These are parameterized with a `probability` attribute that defines the probability that the flow of control moves between the `ResourceConsumer`s they connect. These probabilities can be set to realistic values (as opposed to their uniformly distributed default values) by the modeller such that performance predictions have realistic associated probabilities[10]. In practice, these probabilities reflect the fact that certain scenarios are more probable than others. Finally, non-functional requirements are modelled via `ConsumptionConstraint`s. These may be `ConsumerConsumptionConstraint`s, which are either explicitly associated with ar-

[8]We use the term performance loosely to represent time and other resources "consumed" by an application.

[9]The dotted transformation arrow from PerformanceModel to FileSystem indicates that the generated application does not need to be instrumented with performance measurement facilities for it to function. The dashed arrows into *PerformanceMetrics* indicate three mutually exclusive means of producing them.

[10]Augmenting performance predictions with probabilities is discussed by Becker et al. in [BKR09].

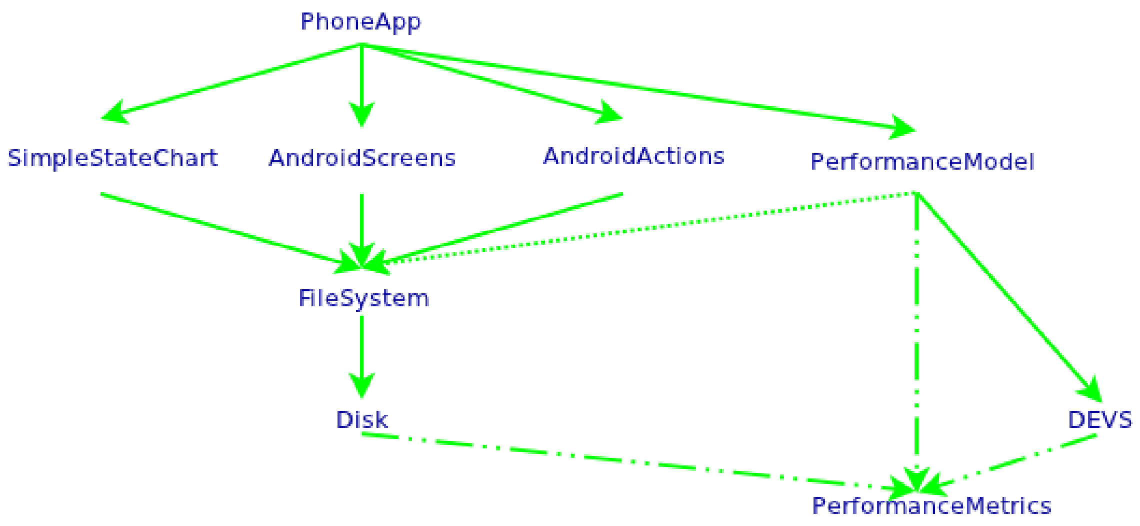


Figure 2.13: An updated Formalism Transformation Graph for PhoneApp, with the added dimension of performance modelling.

bitrary `ResourceConsumers` (via PerformanceModel `appliedTo` links) or implicitly associated with each and every one of them (when they are not connected to any of them), or `ApplicationConsumptionConstraints` which are applied to the system as a whole. This enables the modelling of both local (e.g., maximum resource usage for one component of the system) and global (e.g., maximum resource usage for the full application) requirements. Note that in this example, performance requirements are defined at the level of abstraction of performance models (i.e., as PerformanceModel entities). A more principled approach might be to extend the PhoneApp DSML itself with one or more construct to enable the specification of requirements at the DSm level. This is especially sensible for contexts where performance is a "domain concern": in such cases, it would arguably be non-domain-specific to force modellers to interact with a generated intermediate representation rather than with DSms for performance-related matters. An example enhancement to the PhoneApp DSML could be for `ExecutionSteps` to be augmented with attributes pertaining to their maximum allowed resource consumption. The mapping of this added information onto equivalent `ConsumptionConstraints` would then be carried out by an appropriately altered version of the *PhoneApp-to-PerformanceModel* transformation presented below.

The first benefit of our approach in the context of performance modelling is that application models (i.e., DSms) can be shielded from performance concerns, if and when it makes sense to do so. DSms need not be polluted with performance-related annotations to enable performance modelling. Such annotations traditionally define expected resource consumption ranges. Instead, this information can now be *fully* encapsulated in model transformations that automatically produce complete platform-specific performance models from DSms. In the running example, this is demonstrated through the *PhoneApp-to-PerformanceModel* transformation, or *T_PhoneApp2PerformanceModel(Android)* as it is referred to in Fig-

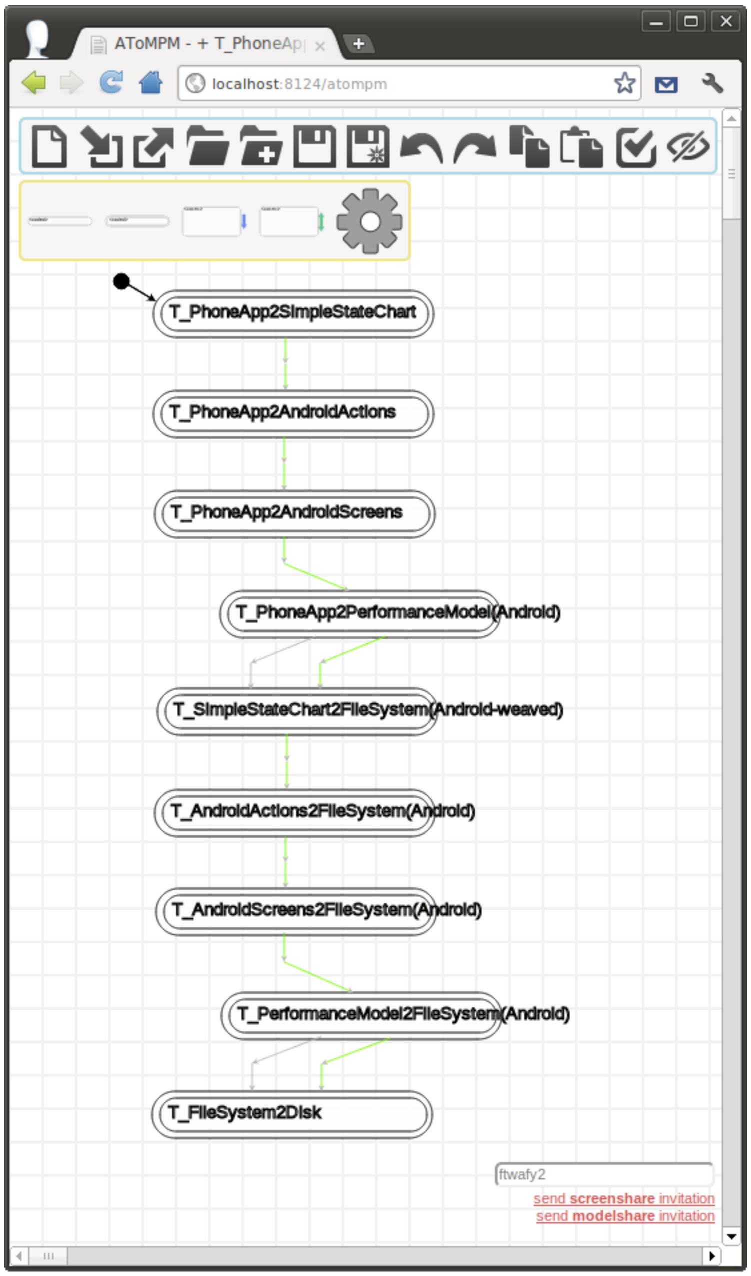

AToMPM - + T_PhoneApp
localhost:8124/atompm
T_PhoneApp2SimpleStateChart
T_PhoneApp2AndroidActions
T_PhoneApp2AndroidScreens
T_PhoneApp2PerformanceModel(Android)
T_SimpleStateChart2FileSystem(Android-weaved)
T_AndroidActions2FileSystem(Android)
T_AndroidScreens2FileSystem(Android)
T_PerformanceModel2FileSystem(Android)
T_FileSystem2Disk
ftwafy2
send screenshare invitation
send modelshare invitation

Figure 2.14: The *PhoneApp2Disk(Android)* transformation, with the added dimension of performance modelling.

ure 2.14, depicted in Figure 2.16. First, the *R_IsPerformanceMonitoringEnabled* rule verifies that performance monitoring is enabled. If it is not, the transformation immediately stops (i.e., no intermediate PerformanceModel model is produced). Otherwise, the flow of control moves on to the *R_ExecutionStep2ResourceConsumer* rule, shown in Figure 2.17a, which implements the projection of all PhoneApp `ExecutionSteps` onto PerformanceModel `ResourceConsumers`. Unseen on the provided depiction is the rule's action code, which pop-

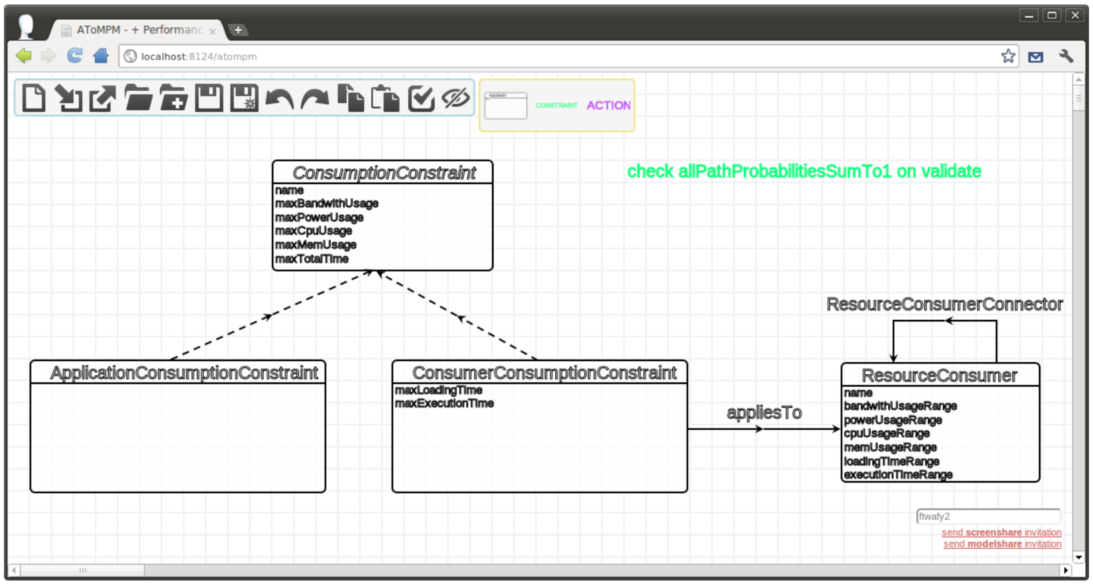


Figure 2.15: The *PerformanceModel* meta-model.

ulates the various consumption range attributes of the generated `ResourceConsumer`. This is accomplished by making use of target platform information pertaining to time and resource consumption of various activities, such as that shown in Table 2.1[11]. For instance, the `loadingTimeRange` attribute of a `ResourceConsumer` corresponding to a `Screen` will reflect the time required for a Google Android device to load and initialize however many widgets there are on that `Screen`. Encapsulating target platform performance specifications in model transformations in this manner is thus an effective means of enabling performance modelling activities without polluting application models. The third and last rule, *R_ConnectResourceConsumers*, shown in Figure 2.17b, connects `ResourceConsumers` if their corresponding `ExecutionSteps` are connected, with rule action code ensuring that the `probability` attribute of generated `ResourceConsumerConnectors` indicates that all paths out of a given `ResourceConsumer` have equal probability (and that these sum to 100%).

When the *PhoneApp-to-PerformanceModel* transformation has run its course (and performance modelling is enabled), each PhoneApp `ExecutionStep` has an associated `ResourceConsumer`. These are connected via `ResourceConsumerConnectors` in a manner that reflects the transitions between corresponding `ExecutionSteps`. Moreover, as in previously introduced projection transformations, only concern-relevant information is projected onto generated performance models. This, as opposed to traditional performance models, which entangle performance concerns and business logic.

[11]Note that Table 2.1 makes the unrealistic simplification that all Google Android devices are identical. Means to overcome this simplification and produce valid device- and platform-specific performance specifications are discussed at the end of this section.

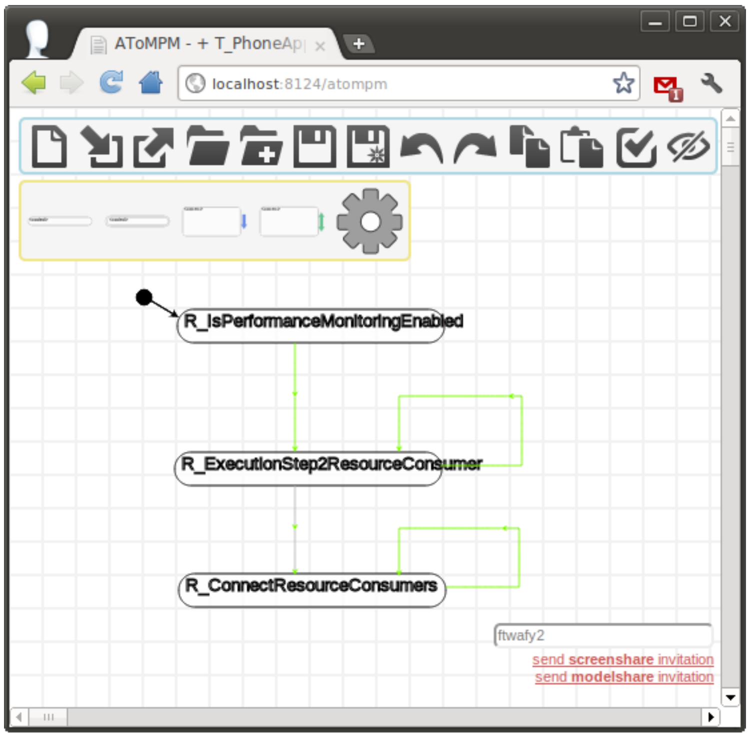
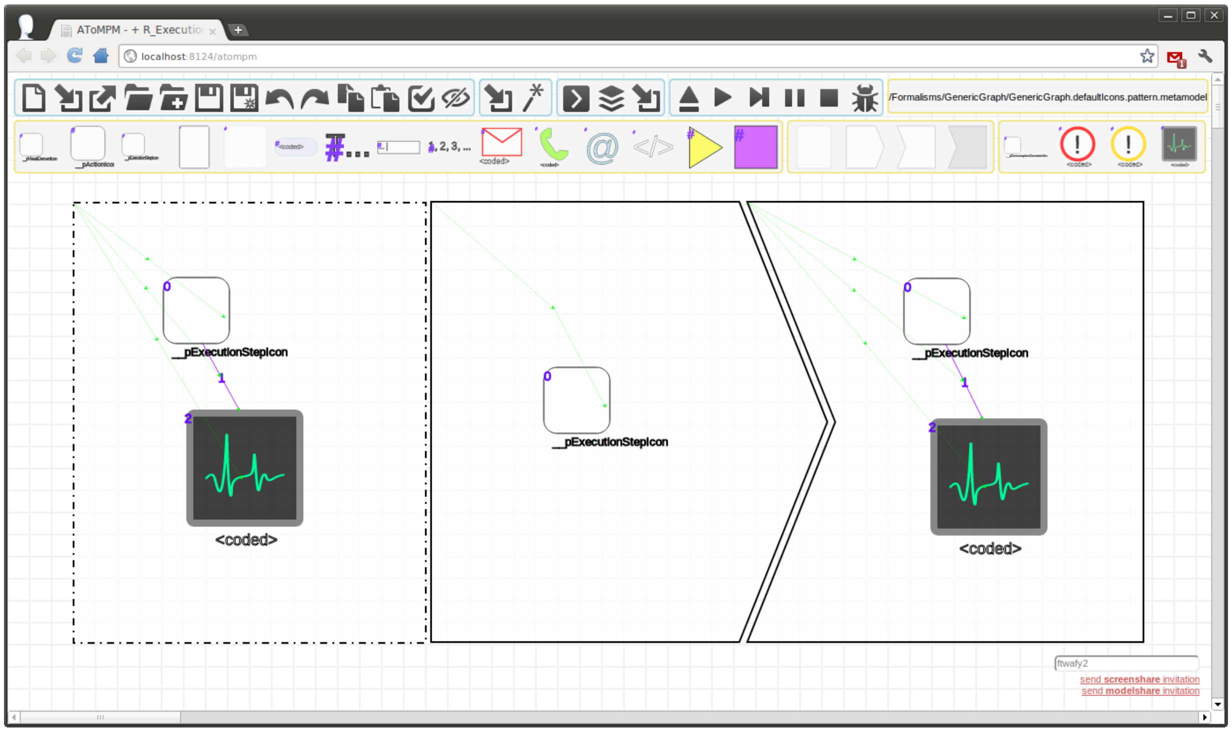

AToMPM - + T_PhoneAp
localhost:8124/atompm
R_IsPerformanceMonitoringEnabled
R_ExecutionStep2ResourceConsumer
R_ConnectResourceConsumers
ftwafy2
send screenshare invitation
send modelshare invitation

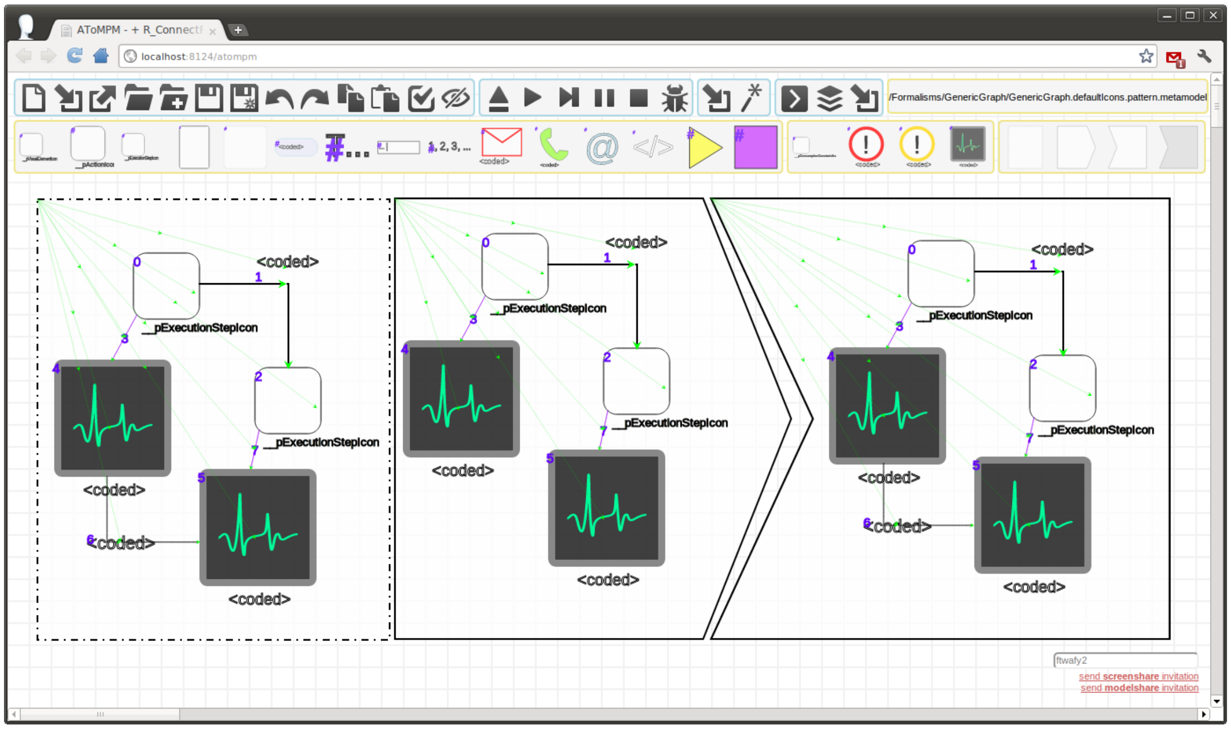
Figure 2.16: The *PhoneApp-to-PerformanceModel* transformation.

Function	**Execution Time Range (s)**	**Battery Usage Range (%)**
Tap Touch Screen	$[0.001, 0.003]$	$[0.0001, 0.0003]$
Load Screen	$[0.05, 0.07] * nb\_widgets$	$[0.001, 0.003] * nb\_widgets$
Send SMS	$[0.1, 0.3] + [0.1, 0.2] * \lceil sms.length \div 120 \rceil$	$[0.01, 0.03] * \lceil sms.length \div 120 \rceil$
Send Email	$[0.05, 0.1] * \lceil msg.length \div 1024 \rceil$	$[0.05, 0.07] * \lceil msg.length \div 1024 \rceil$
Load Web Data	$data.size \div 200 \frac{Kb}{s}$	$data.size \div 50 \frac{Mb}{\%}$
Load Local Data	$data.size \div 5 \frac{Mb}{s}$	$data.size \div 500 \frac{Mb}{\%}$

Table 2.1: Synthetic performance specifications for Google Android devices.

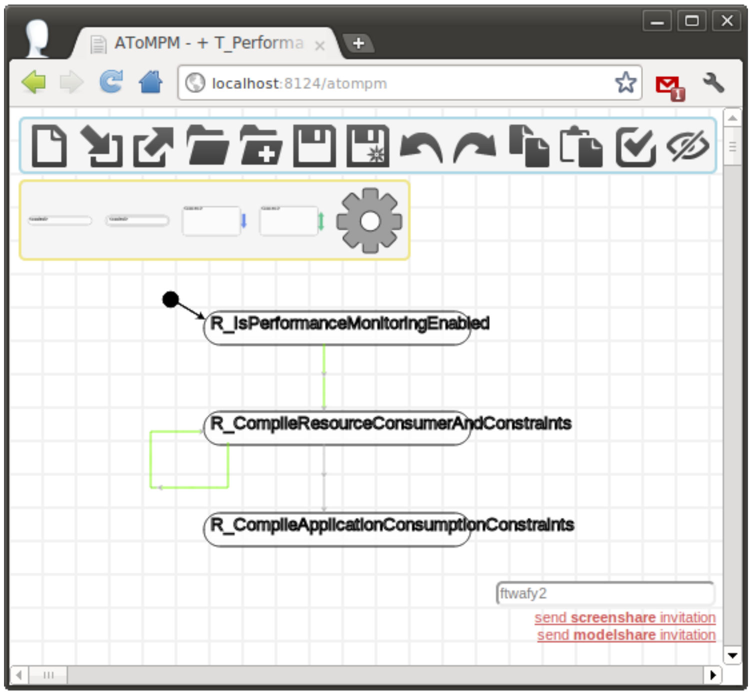

AToMPM - + R_Executio
localhost:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
0
__pExecutionStepIcon
1
2
<coded>
ftwafy2
send screenshare invitation
send modelshare invitation

(a) The *R_ExecutionStep2ResourceConsumer* rule.


AToMPM - + R_Connecti
localhost:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
0
__pExecutionStepIcon
1
<coded>
2
3
4
5
6
7
ftwafy2
send screenshare invitation
send modelshare invitation

(b) The *R_ConnectResourceConsumers* rule.

Figure 2.17: The rules that compose the *PhoneApp-to-PerformanceModel* model transformation.



AToMPM - + T_Performa
localhost:8124/atompm
R_IsPerformanceMonitoringEnabled
R_CompileResourceConsumerAndConstraints
R_CompileApplicationConsumptionConstraints
ftwafy2
send screenshare invitation
send modelshare invitation

Figure 2.18: The *PerformanceModel-to-FileSystem* transformation.

The second benefit of our approach in the context of performance modelling is a consequence of its ties to DSM. A key difference between DSM and more traditional UML-centric modelling efforts is that the former enables full code generation (as opposed to code skeletons). Hence, performance models generated from DSms can not only be used to produce simulations and performance predictions, they can also be further integrated into the artifact generation process to instrument target platform artifacts with performance measurement and reporting facilities. This integration and instrumentation is analogous to the merging of intermediate formalisms discussed in Section 2.3.5. Revisiting the running example, entry and exit actions of compiled Statechart `State`s can be augmented with performance measurement facilities to compute the time and resources consumed by the system in each `State`. Exit actions can be further instrumented with facilities to communicate their measurements back to the model editor (and the modeller). Indeed, the process of performing DSm animation by propagating commands up along the network of traceability links described in Section 2.3.6 can be reused to propagate performance measurements back to any of the intermediate representations and to the DSm[12] during artifact execution. All of the above is prototyped within the *PerformanceModel-to-FileSystem* transformation, or *T_PerformanceModel2FileSystem(Android)* as it is referred to in Figure 2.14, depicted in Figure 2.18. Compiled entry actions measure the current levels of monitored resources (e.g., time, battery), while compiled exit actions repeat these mea-

[12]Conceptually, it may be ambiguous or confusing for performance-related information to be displayed in any representation other than the generated performance model.

surements to establish consumption and transmit their measurements to the modelling tool via tagging commands (`tag:`$< text >, < color >, ResourceConsumerId$). A key point of interest is that `ResourceConsumptionConstraints` also participate in exit action instrumentation: if a constraint described by a modelled requirement is not satisfied, the infringing `ResourceConsumer` entity will be tagged with a red performance metric rather than a green one. Thus, a third benefit of our DSM-based approach is that it enables models of non-functional requirements to be meaningfully included in the artifact synthesis process such that their satisfaction can be intuitively reported to modellers in real-time. Note that global-only performance measurements may be compared to global-and-local measurements to ascertain the performance footprint of the generated measurement and reporting facilities.

Lastly, the aforementioned instrumentation of artifacts may also assist in the arduous task of *model calibration*, a prerequisite to the synthesis of platform- and/or device-specific performance models. This task consists in determining a platform or target device's performance parameters (i.e., the time and resource consumption associated with various activities) such that a realistic *platform model* may be produced. A synthetic platform model is given in Table 2.1. Target-platform artifacts augmented with performance measurement and reporting facilities can be used to measure the performance of arbitrary benchmarking tasks (e.g., the loading of a screen with x widgets). Hence, under our approach, model calibration is reduced to the creation of "benchmark" DSms where tasks of interest are modelled, and to the collection of the performance metrics reported by automatically synthesized and instrumented target-platform artifacts.

2.4.2 Revisiting Performance Assessment

From PerformanceModel models, any one of the three general performance assessment methods may be carried out, namely, *analysis*, *simulation* and *testing*[13].

Performance *analysis* statically computes metrics from performance models. In Figure 2.13, this is captured by the *PerformanceModel-to-PerformanceMetrics* transformation. Such a transformation must first extract every possible path between the `ResourceConsumers` corresponding to the initial and finishing application states (here PhoneApp `ExecutionSteps`). Then, for each path, resource consumption metrics and probabilities are computed by summing the resource requirement ranges of each `ResourceConsumer`, and multiplying the probabilities of all involved `ResourceConsumerConnectors`. This cumulative approach to estimating a path's resource consumption is not unlike that presented in [CCG+09, MSMG10], where general equations are introduced to compute performance metrics of various component compositions. In the end, a set of probability-weighted execution paths have been computed. These are at the level of abstraction of the desired target domain, i.e., performance metrics. Although performance analysis in general is powerful due to its exhaustive nature, it can become impractical (and even infeasible) as the number of possible usage sce-

[13]A thorough comparison of the pros and cons of these different methods is provided in [BKR09].

narios (or paths) of a system grows. Our approach does not (and can not) escape this reality, and as such, as the numbers of `ResourceConsumer`s and `ResourceConsumerConnector` increase, the number of execution paths may become intractable.

Simulations enable arbitrary execution paths to be observed and others to be ignored. In modern UML-centric modelling efforts, simulations are commonly generated from software models refined with performance annotations, with missing business logic captured by synthetic workloads. To capture a system's behaviour at an appropriate level of abstraction, with a focus on time and resource-usage, the DEVS (Discrete EVent system Specification) [ZKP00] formalism is often appropriate. For the purpose of this discussion, DEVS is similar to Statecharts, with a different kind of modularity, tailored for simulation. The key benefits of producing DEVS *models* rather than *coded* approximations of a system are essentially the same as those discussed in Section 2.3.6. In Figure 2.13, the synthesis of DEVS models for simulation is captured by the *PerformanceModel-to-DEVS* transformation, while the execution of the simulated DEVS model, which produces performance metrics, is captured by the *DEVS-to-PerformanceMetrics* transformation. Further descriptions are omitted and left to the reader due to their similarity with the *PhoneApp-to-Statechart* and *Statechart-to-FileSystem* transformations, respectively.

The third and last performance assessment method is the *testing* of (nearly) completed products. Their performance is often measured through code instrumentation with measurement facilities. In the running example, this *weaving*[14] of measurement facilities is captured by the *PerformanceModel-to-FileSystem* transformation, which was reviewed in the previous sub-section.

Three common performance assessment methods, each of which is instrumental in the development of modern embedded systems applications, have been revisited from the perspective of the introduced approach to artifact synthesis. For the latter two in particular, our DSM-based approach improves upon its state-of-the-art siblings in the UML-centric development world by shielding modellers and DSms from a number performance-related tasks, and by producing performance models that are not polluted by business logic. The approach is now further evaluated by reviewing how performance analyses, simulations and measurement facilities would be produced using the traditional coded text generator approach to artifact synthesis from DSms. The task of generating performance metrics from DSms (i.e., carrying out performance analysis) is analogous to that of generating any other artifact. Thus, the traditional coded generator approach would programmatically iterate over model entities to produce desired output, conceivably with a coded version of the traversal algorithm described above. Augmenting existing coded generators to produce such performance metrics would increase their accidental complexity and further reduce their modularity. Additional instrumentation to add performance measurement and reporting facilities into target

[14]The term "weaving" is borrowed from the Aspect-Oriented Programming [KLM+97] world due to certain similarities between aspect weaving and our instrumentation of compiled Statechart models.

platform artifacts, or to produce coded simulations or DEVS models would either result in considerable code duplication, or in further loss of modularity. Thus, although coded generators can replicate the three performance assessment methods – after all, anything can be programmed –, doing so would considerably hamper their modularity, accessibility and maintainability.

2.5 Case study: Conference Registration in PhoneApp

This section provides a concrete overview of the synthesis of a fully functional Google Android application from a PhoneApp model of a conference registration system[15]. All intermediate representations are depicted, as well as highlighting and tagging (of performance measurements) at the DSm level.

Figure 2.19 shows a conference registration system represented as a domain-specific PhoneApp model. The system has three main use cases: registering, viewing the program schedule and cancelling a registration. The former is explored below:

1. The user sees the *Welcome* screen for 2 seconds and is taken to the *ActionChoice* screen.
2. The user clicks on “Register” and is taken to the *NameEntry* screen.
3. The user enters his name, clicks “Next” and is taken to the *PaymentMethodChoice* screen.
4. The user clicks on a payment method. A text message containing the user’s name and chosen payment method is sent to a modeller-specified phone number after which the user is taken to the *RegistrationCompleted* screen.
5. The user sees the *RegistrationCompleted* screen for 2 seconds and the application exits.
6. *The mobile device’s operating system restores the device to its state prior to the launch of the conference registration application.*

The introduced approach dictates that to produce the desired artifact, in this case a Google Android application, the DSm must be decomposed into a number of single-concern lower-level models, which must in turn be merged back together. Figures 2.20, 2.21, 2.22 and 2.23 each display intermediate representations between the DSm and artifact levels. Figure 2.20 shows the behavioural concerns of the conference registration application projected onto a Statechart model. Figure 2.21 shows the layout and mobile phone feature concerns projected onto an AndroidScreens model and an AndroidActions model, which are shown as a single multi-formalism model. Figure 2.22 shows a Google Android-specific performance

[15]The conference registration example, like the PhoneApp meta-model, is heavily inspired by the work of Kelly and Tolvanen in [Met, KT08].

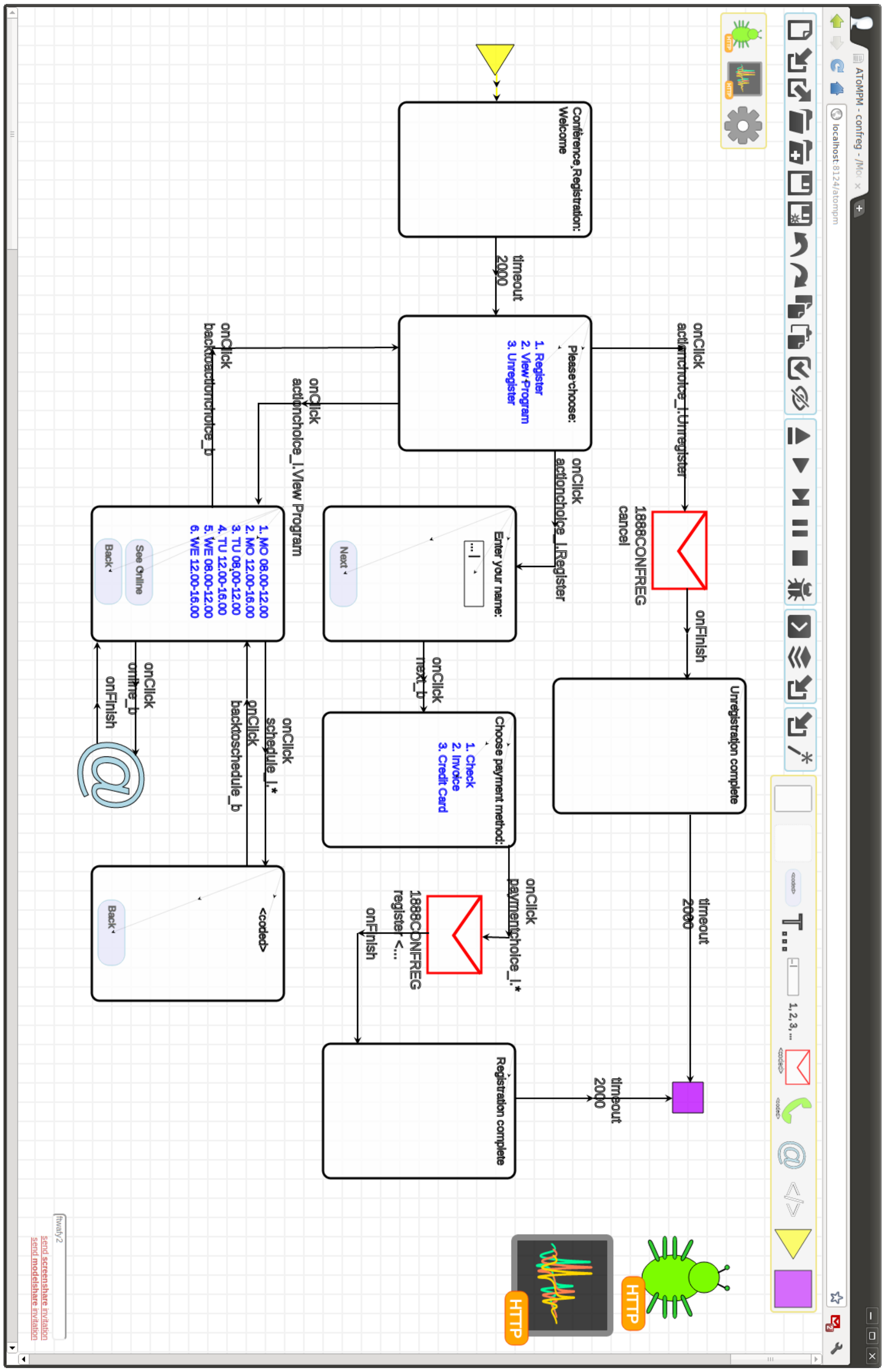


Figure 2.19: A conference registration application, as a PhoneApp model.

model, as a PerformanceModel model. Finally, Figure 2.23 shows a FileSystem model of the files that hold the merged form of all of the above models.

Much of the discussion surrounding the benefits of the proposed technique to artifact synthesis pertain to the traceability links that it leaves behind. Though they have been presented in isolation, models from every formalism at every level of abstraction are linked by an intricate web of interconnections between corresponding entities at different levels of abstraction. This web is partially shown in Figure 2.24. Note that it is by no means meant for human use in its current form, and is displayed here merely to illustrate an otherwise overly abstract concept. In the future, *slicing* this web of artifacts and links may provide useful insights to modellers.

Last but not least, Figure 2.25 demonstrates the aforementioned advanced functionalities enabled by traceability links, namely, the highlighting of model entities during artifact execution (on a physical Google Android device) and their tagging with performance measurements. The conference registration DSm and its associated performance model are overlaid. Note the `ConsumerConsumptionConstraint` in the top-left corner that regulates loading and execution time. As the synthesized application is executed on a Google Android-enabled device, the current PhoneApp `ExecutionStep` is highlighted in blue, entities corresponding to it from other formalisms (here, a PerformanceModel `ResourceConsumer`) are highlighted in yellow, and performance measurements for the regulated resources (here, loading and execution time) are tagged to the relevant `ResourceConsumer`s in a color that reflects their satisfaction of modelled requirements (i.e., green for success, red for failure). Present but not shown is the fact that state information is also propagated from artifact to DSm such that attributes at the DSm level take on values during artifact execution. For instance, the `Label` from the *NameEntry* screen is updated to reflect user input at the artifact level (i.e., user input on the physical device running the synthesized application).

2.6 Comparison with Related Work

This section reviews and situates the introduced approach with respect to relevant work by others surveyed in Section 2.2.

Benefits of the proposed approach with respect to traditional coded artifact generators were discussed in Section 2.3.6. These position the approach as a solution to numerous of its alternative's pitfalls, namely, overly low levels of abstraction that make their development, maintenance and enhancement very complex. As for more intimately related techniques, Yie et al. also propose the decomposition of "almighty" model transformations into a number of sub-transformations. However, their approach is limited with respect to ours in that no guidelines are provided as to which criteria to use to perform the decomposition (e.g., decompose along concern lines) *and* that manual intervention is required to enable inter-artifact communication. Finally, our work also has a likeness to Hemel et al.'s work on representing

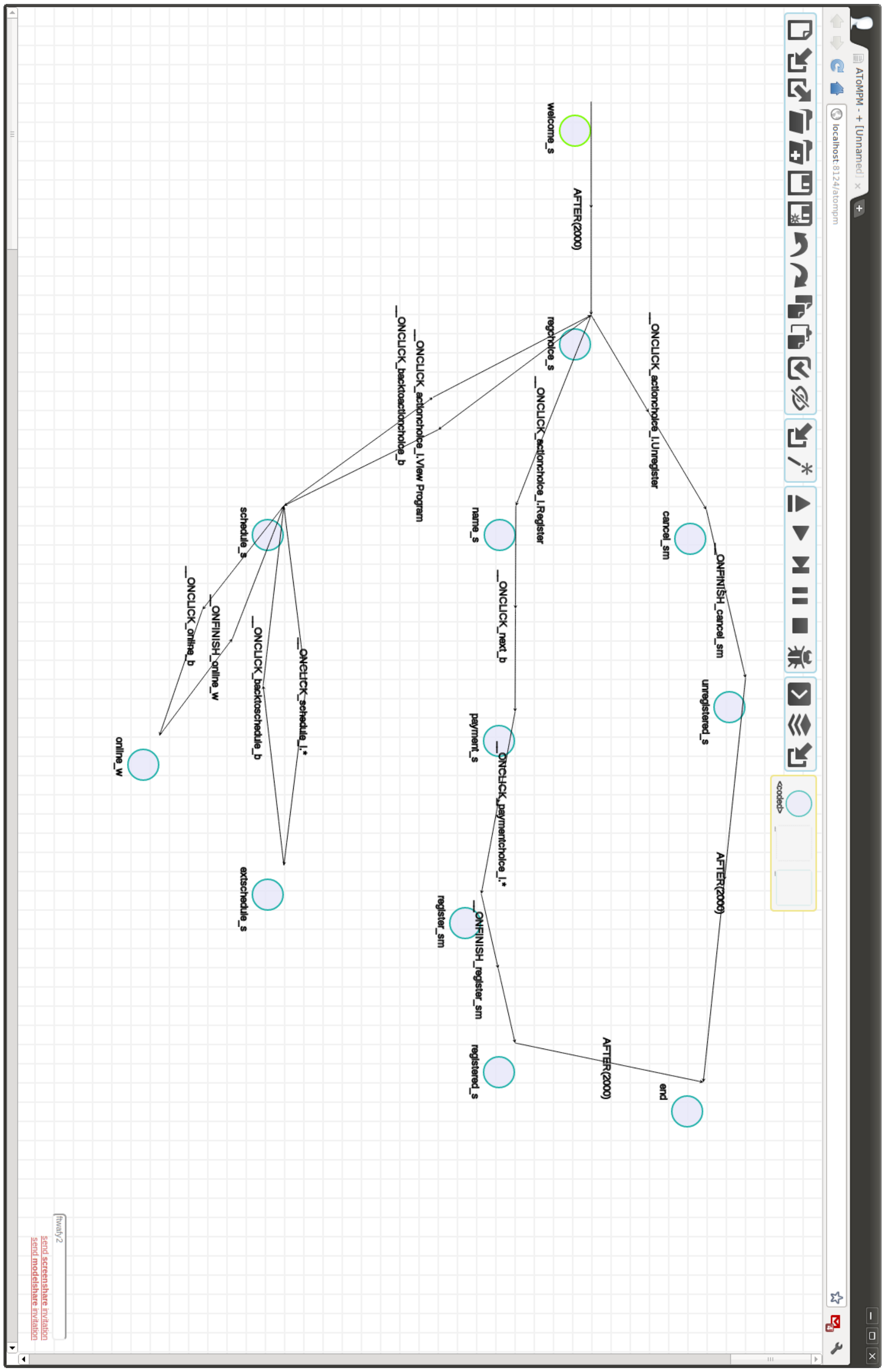


Figure 2.20: The behavioural concerns of the conference registration application projected onto a Statechart model.

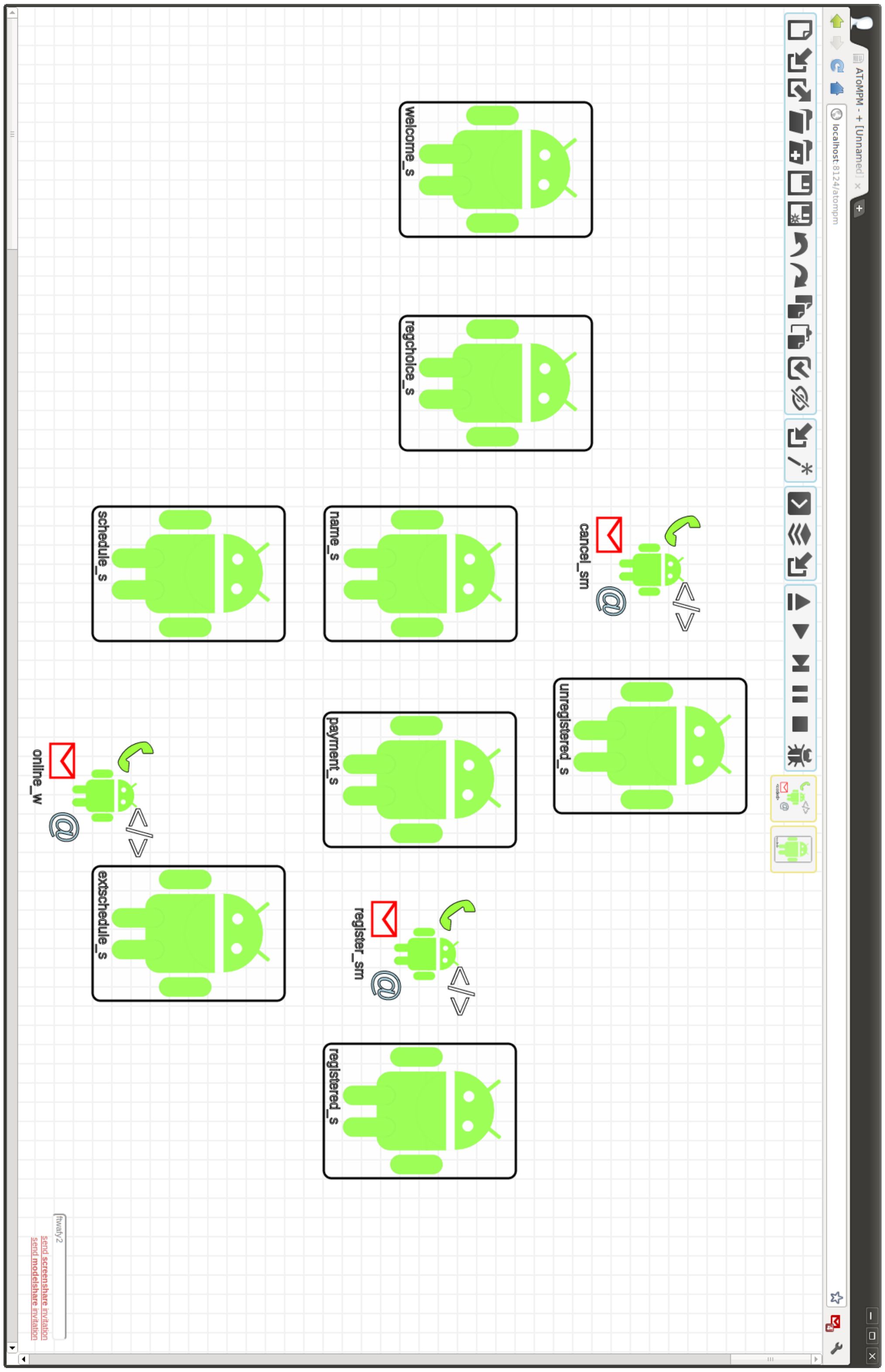


Figure 2.21: The layout and mobile phone feature concerns of the conference registration application projected onto an AndroidScreens model and onto an AndroidActions model, respectively.

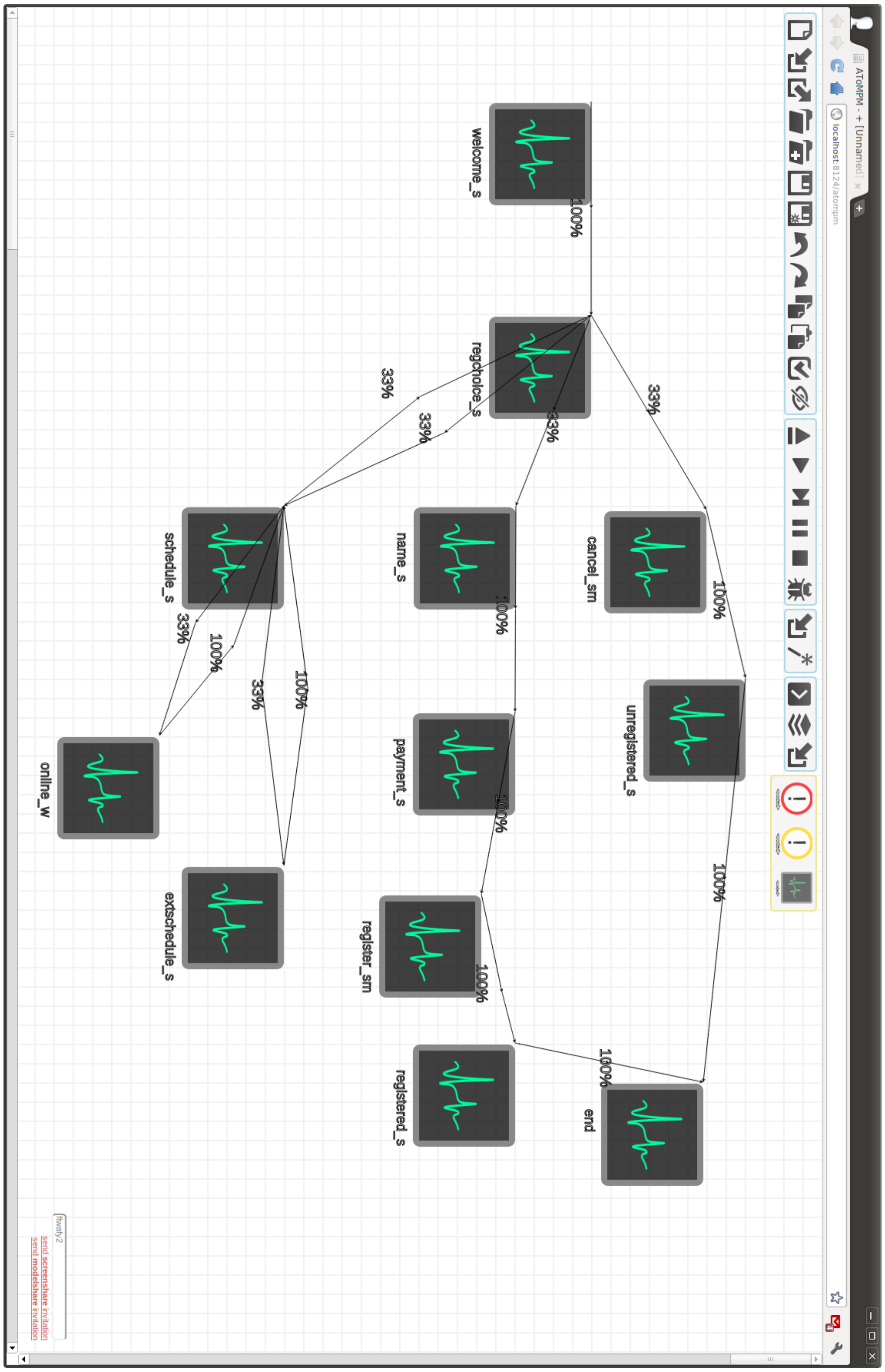


Figure 2.22: A Google Android-specific performance model generated from the conference registration application model, as a PerformanceModel model.

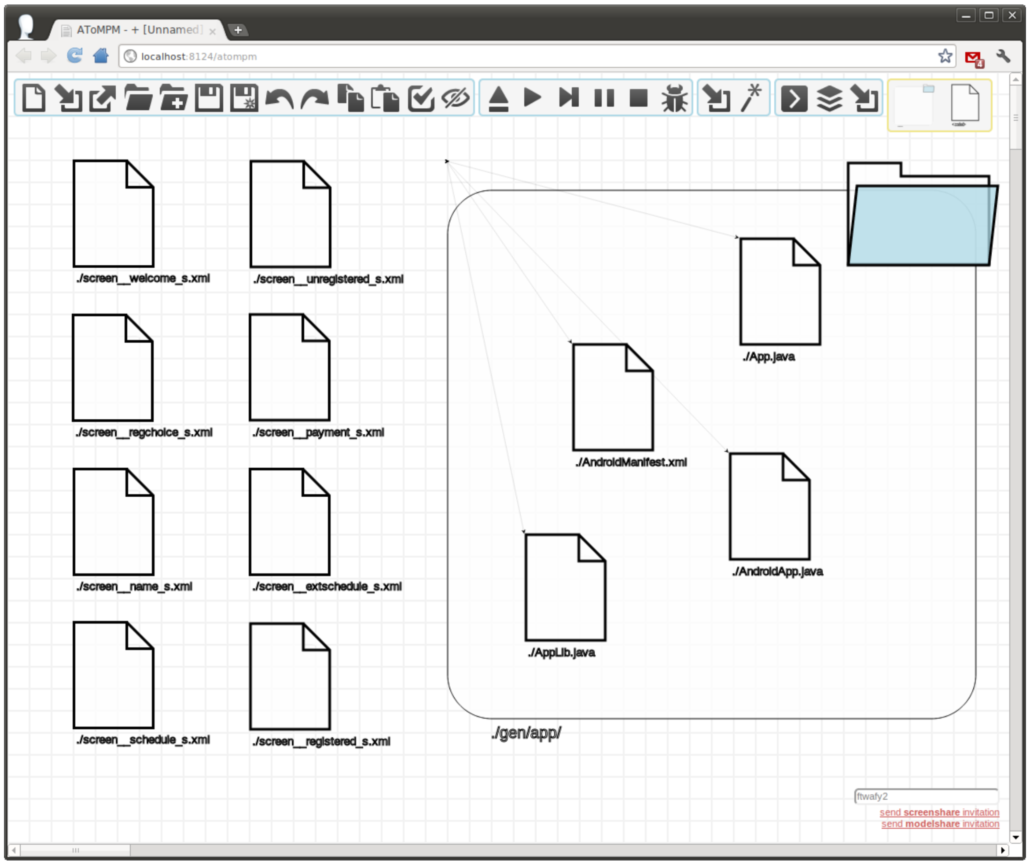


Figure 2.23: A FileSystem model of the files containing the merging of all intermediate models.

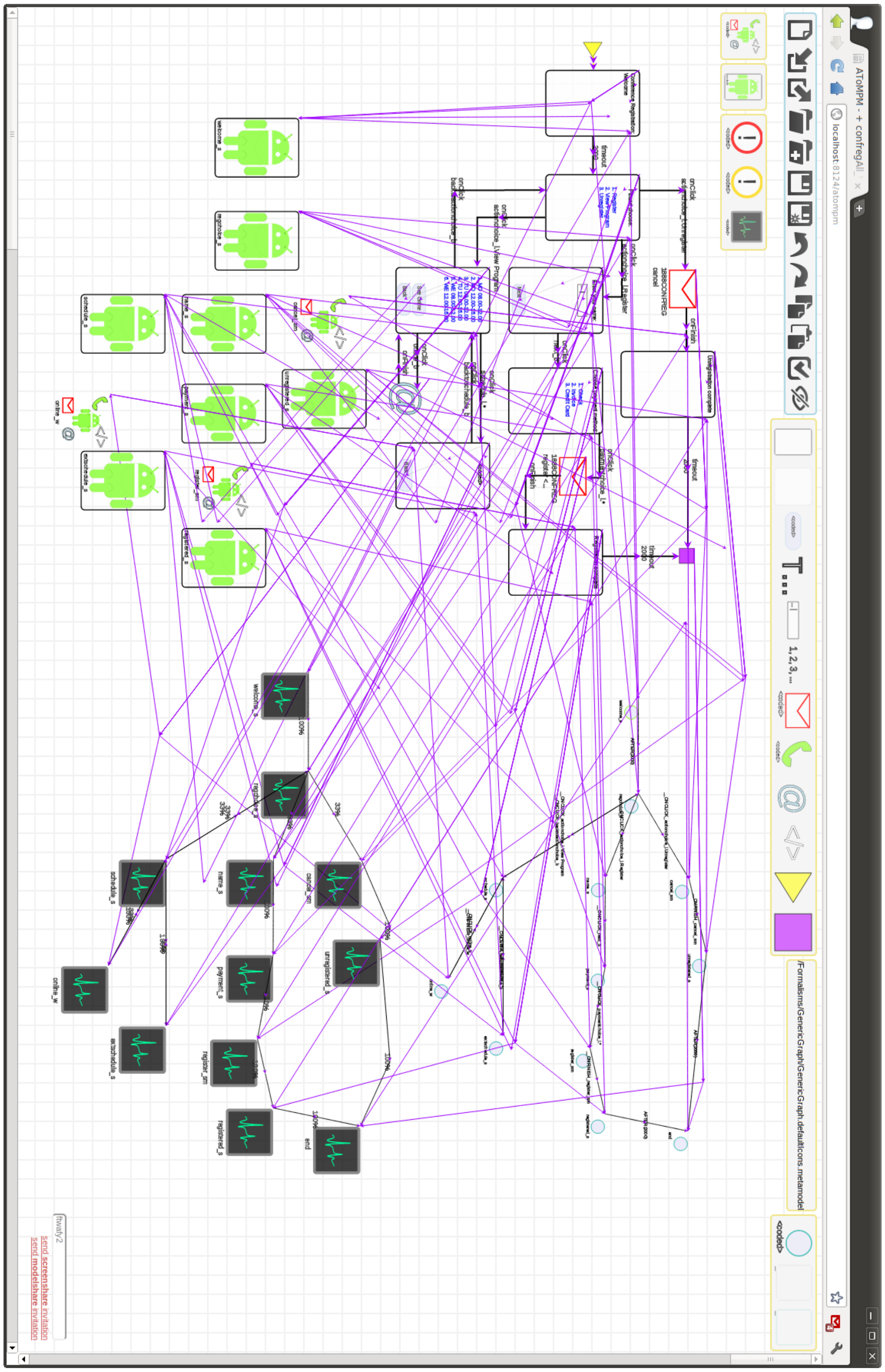


Figure 2.24: The complete web of traceability links left behind by the artifact synthesis process.

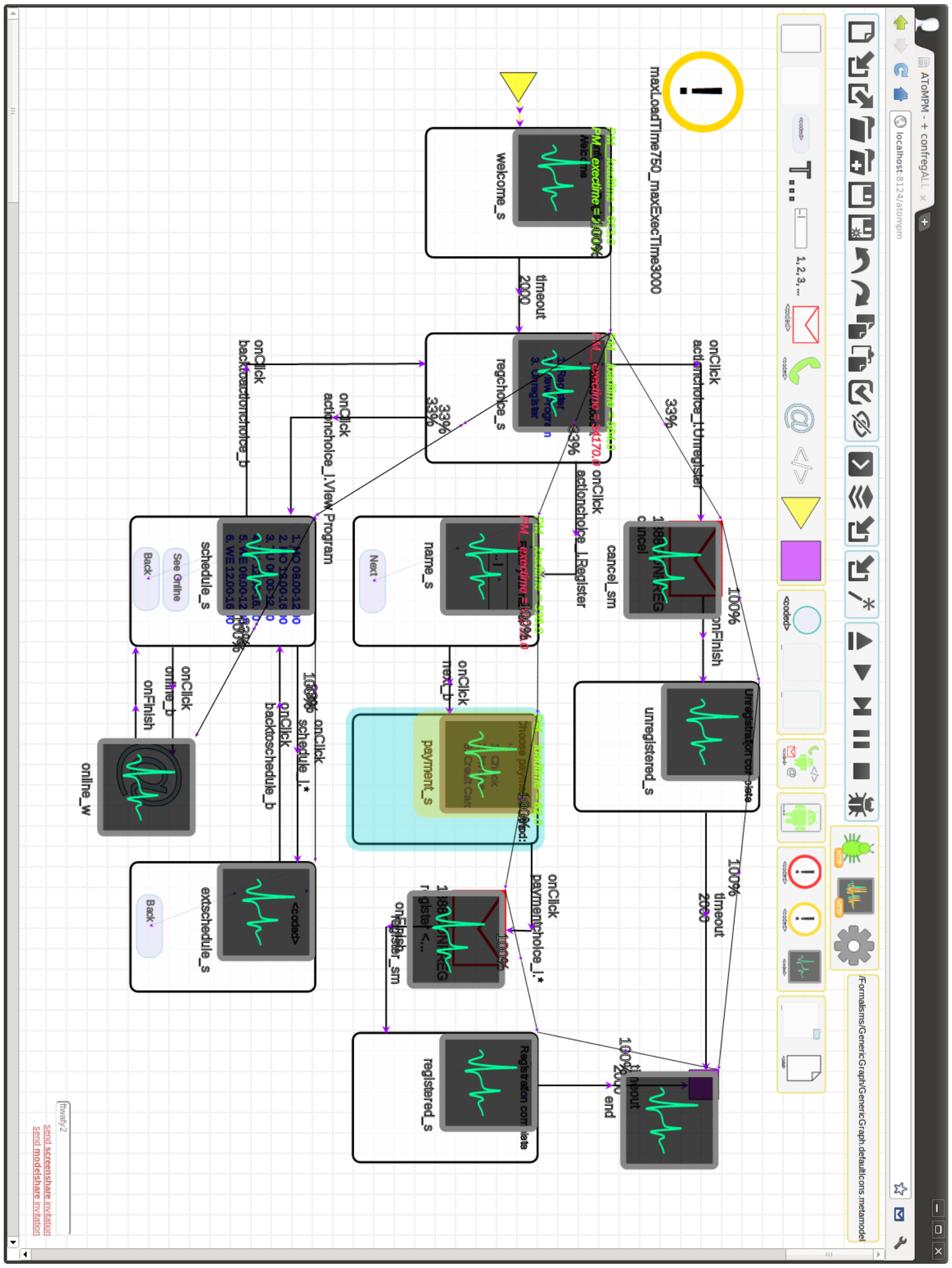


Figure 2.25: Highlighting and tagging model entities during artifact execution.

coded programs as models that conform to meta-models that capture the syntax of GPLs. The use of FileSystem as an intermediate representation shares much of the motivations and benefits of their approach, although it does not go as far.

Many of our approach's advantages were attributed to the web of traceability links it produces. Wu et al. also recognized the usefulness of DSm-to-artifact mappings (in the context of debugging). However, their work is limited with respect to our own in a number of ways, not including the restrictions they impose on DSms (textual only) and tools (Eclipse only). Indeed, graver limitations are the facts that the mapping construction inevitably introduces considerable accidental complexity into already complex coded artifact generators, and that the mapping is not readily presentable to the modeller. The latter, in particular, implies that traceability links can not be used for didactic purposes or to assist in debugging model transformation rules (e.g., by making explicit what is generated from what).

As for performance modelling, while the proposed approach borrows ideas from traditional UML-centric methods, such as the early integration of non-functional requirements into the development process and the parameterization of control flow paths with probabilities, it improves upon them in a variety of ways. Accidental complexity is maximally reduced by shielding DSms from performance concerns (when applicable) and performance models from business logic. However, the most noteworthy improvement is undoubtedly the ability to integrate non-functional requirements seamlessly into the performance testing process through their transformation into instrumented code that verifies their satisfaction at runtime. This is a form of *runtime monitoring* [VK05] which is gaining popularity as a complement to testing.

Thus, the approach can be summarized as a structured and MPM take on traditional and even state-of-the-art artifact generation and performance modelling techniques.

2.7 Conclusion and Future Work

The work presented in this chapter was motivated by the numerous shortcomings of the traditional approach to artifact synthesis from DSms. The programmatic manipulation of internal model representations to produce target platform artifacts is at too low a level of abstraction. This makes them difficult to reason about, maintain (e.g., as a result of meta-model evolution) and extend.

The introduced approach addresses these limitations. Artifact synthesis from DSms is carried out via visual rule-based graph transformations that isolate and project tangled concerns within DSms onto appropriate lower-level modelling formalisms as an intermediate step to final artifact generation. This approach has numerous benefits, including a considerable raise in the level of abstraction of artifact generators, which increases their accessibility and eases their maintenance and extensibility. It also greatly facilitates the maintenance of traceability information between corresponding constructs at different levels of abstraction. This information is instrumental in enabling "advanced" tasks such as DSm and model transformation debugging, and DSm animation (as a result of artifact execution). Additionally, the approach contributes to the area of embedded system applications modelling (and synthesizing) and, more specifically, in the addressing of their characteristic non-functional requirements. The discussed benefits of structuring artifact synthesis are not restricted to coded application synthesis. They also apply to the generation of performance models from DSms, and of performance predictions, simulations and measurement facilities from performance models.

The proposed technique was demonstrated by detailing the synthesis of fully functional Google Android applications from DSms, optionally instrumented with performance measurement and reporting facilities. Note however that although the case study is bound to the Google Android platform, the approach is not. Despite the fact that some of the presented model transformations would need to be refactored if the targeted platform were to change (e.g., to Apple iOS [Appa]), their essence and purpose would be left intact, with each one isolating (or merging) a single concern onto lower- and lower-level representations.

Finally, despite its advantages, our technique still has the unfortunate drawback that it requires a considerable amount of (non-trivial) manual work: the semantic mapping of DSms to artifacts still needs to be specified manually (by DSML designers). This implies that DSML designers must manually identify which portions of their languages to project onto which lower-level formalisms, how to carry out the said projections, and how to merge their results back into coherent artifacts. The work presented in Chapter 4 addresses these limitations and attempts to alleviate and simplify the burden placed on DSML designers.

Chapter 3

Debugging in Domain-Specific Modelling

This chapter introduces a mapping between debugging concepts (e.g., *breakpoints*, *assertions*) in the software and DSM realms. The meaning of these concepts is explored from the very different perspectives of DSML *designers*, who develop and must debug model compilers (ideally specified as model transformations), and of domain-specific *modellers*, who develop and must debug DSms while remaining unaware of artifacts and their generators. A set of guidelines, caveats and examples are proposed with the aim of providing blueprints for future DSM debuggers.

3.1 Problem Statement and Outline

Ideally, the typical workflow of a DSM project consists of the specification by DSM experts of one or more DSMLs and of the model transformations that define their semantics. Subsequently, DSms may be created by arbitrary domain experts[1]. In practice, models, model transformations and synthesized artifacts may all require debugging. A significant obstacle to the wide-spread adoption of model-driven development approaches in industry is the lack of proper debugging facilities. Software debugging support is provided by a combination of language and IDE features which enable the monitoring and altering of a running program's state at the same level of abstraction as that at which the program is written. How these language and tool features translate to the DSM realm is still misunderstood, and is a topic that few researchers have explored in depth.

To address this, we first propose a complete mapping of all common debugging facilities from the software realm to the DSM realm. This mapping accounts for the fact that DSML *designers* and their end-users, or *modellers*, have very different workflows, and thus very different debugging needs. Then, in the context of debugging DSML specifications, we explore the debugging of semantics specified via model transformations. The debugging of coded model manipulators (e.g., coded artifact generators) is also briefly discussed. Next, the debugging of DSms and artifacts (always at the DSm level) is targeted. For these in particular, the approach to artifact synthesis presented in the previous chapter emerges as a truly enabling technique for debugging. This is due in part to the two-way communication enabled by the traceability links it leaves behind which tremendously facilitates the exchange of various debugging commands from DSm to artifact and back.

The rest of this chapter is structured as follows. In Section 3.2, we survey relevant research work. In Section 3.3, we review common debugging concepts such as *breakpoints* and *assertions* from the programming world. In Section 3.4, we proceed to mapping these concepts onto the DSM realm in the context of debugging from the DSML designer perspective. In Section 3.5, this mapping is repeated in the context of debugging DSms and synthesized artifacts. For both scenarios, insights, suggestions and demonstrations are provided. In Section 3.6, we compare these to relevant facilities provided by existing tools or suggested by other researchers. Finally, in Section 3.7 we discuss future work and provide some closing remarks.

3.2 Survey of Relevant Research Work

The purpose of this section is to provide an in-depth survey of research work relevant to the proposed scientific contributions of this chapter. Our contributions are neither compared here nor are they situated with respect to the reviewed works. Such comparisons are instead

[1]Note that for the purpose of this discussion, the fact that the provided description of the DSM process is over-simplified, leaving out DSML evolution for instance, is of no consequence.

provided in Section 3.6.

Very little attention has been paid to debugging by the DSM and MDE communities. Many reports of modern industrial applications of DSM admit that the debugging of models, of their associated ad hoc compilers, and of synthesized artifacts is accomplished without any tool support [Saf07, KT08]. Far more alarmingly, debugging is invariably performed at the code and synthesized artifact levels of abstraction rather than at that of DSms. Debugging deals exclusively with synthesized artifacts and with coded compilers that make use of modelling tool APIs. In the code-centric software development realm, this would be equivalent to instrumenting compilers and compiled bytecode with print statements and stepping through the bytecode to find and resolve issues in a program written in a GPL. This approach is quite obviously in blatant contradiction with the founding goals and principles of DSM: abstraction can not truly be raised and the resulting benefits can not be accurately measured and fully reaped as long as domain-specific modellers are required to manipulate non-domain-specific concepts.

3.2.1 Domain-Specific Model Debugging

Some researchers have recognized the aforementioned contradiction and attempted to provide debugging means at the DSm level. The most advanced published DSm debugger to-date is that presented by Wu et al. in [WGM08]. Their approach allows for the re-use of existing, tried and familiar code debugging facilities at the DSm level of abstraction. First, during artifact synthesis, entirely hidden from the modeller, a detailed and direct mapping between statements in textual DSms and resulting statements within synthesized coded artifacts is constructed. Second, the Eclipse [EFa] Integrated Development Environment's built-in Debugging Perspective is enhanced such that breakpoints may be set on DSm statements and that these may be stepped through. In practice, while executing DSms[2] in debugging mode, "break" and "step" events at the DSm level are internally translated to "break" and "step" commands at the artifact level using the constructed mapping to properly parametrize the said commands. Lastly, during DSm compilation, artifacts are also instrumented such that variable values at the DSm level are appropriately updated to reflect state changes of executing artifacts. Wu et al.'s approach has many clear benefits. On the one hand, domain-specific modellers are given means to set breakpoints within domain-specific code, to pause/resume execution and to review the changing state of DSms as underlying artifacts are executed without requiring any prior knowledge about the synthesized code or the code generator. On the other hand, from a tool implementer's point of view, rather than require the implementation of a whole new debugger for every DSML, their approach allows textual DSML designers to fully reuse the powerful and proven built-in debugging facilities of a popular

[2]This is abusive notation. In reality, artifacts, not DSms are executed. Appropriate feedback mechanisms may be used to create the illusion that DSms are being executed though. This is analogous to the impression, created by IDE debuggers, that GPL programs are executed, when in fact only their compiled form is ever executed.

Integrated Development Environment[3] (IDE). As for the approach's limitations, it is currently limited to textual DSMLs, restricts the modeller to the Eclipse tool, and assumes that generated artifacts are code. Each of these can render the approach unusable in a number of contexts where DSM might be applied. Furthermore, little is said about the inevitable increase in complexity to the already complex task of DSm compiler development incurred by the added requirements of producing DSm-to-artifact mappings and instrumenting the said artifacts with means to synchronize DSm variables with artifact variables during execution. The last and possibly most noteworthy shortcoming of the proposed technique is that is does not consider the debugging of the compiler, be it implemented via code or model transformations, that produces artifacts from models. Thus, it focuses solely on debugging from the domain-specific modeller's perspective, but not at all from the DSM expert (and DSML designer) perspective.

Kos et al's Ladybird DSm debugger is more advanced than Wu et al.'s in some respects [KKMK11]. On the one hand, its obvious limitation is that it is tailored for a single DSML and bound to the single tool that supports it, giving it a somewhat anecdotal character. On the other hand, it is based on and implements many of the recommendations presented in this chapter[4] and provides a real world example of debugging constructs such as breakpoints and runtime variable I/O at the DSm level.

Finally, in [Sun11], Sun also recognizes the importance of providing specification and debugging means at the same level of abstraction. He does so in the context of MTBD, where model transformation rules are implicitly specified through the recording of developer actions. The resulting recordings can be seen as DSms of rules themselves. As such, Sun stresses the importance of providing and demonstrates debugging capabilities that deal with recorded developer actions, as opposed to synthesized "LHS-RHS" rules.

3.2.2 Model Transformation Debugging

There also exists a body of research that addresses the debugging of model transformations. Incidentally, when DSm-to-artifact compilers are defined, as prescribed by MDE principles, via model transformations, proposed techniques for their debugging can address the last of the aforementioned limitations of Wu et al.'s work. Certain modern model transformation-enabled DSM tools (e.g., AToM^3, AToMPM) support basic model transformation debugging by providing a number of enabling facilities. These include step-by-step, as opposed to continuous, rule execution. Instead of having transformations always run to completion, rules can be executed one at a time, users may manually select which rule to execute in cases where multiple rules are simultaneously applicable, or even which of many LHS pattern instances matched in the source graph to actually transform. Furthermore, modification of

[3]Note that although IDEs for developing models exist, every mention of "IDEs" in this work refers exclusively to coded program IDEs.

[4]More specifically, Kos et al. essentially followed the blueprint for DSM debuggers published in [MV11a].

the source model between rule applications is permitted. This enables rule designers to observe the effects of each rule in isolation, to correct erroneous effects during execution, and even to somewhat steer the transformation in desired directions. Still, despite the ability to perform these non-trivial transformation debugging tasks, more advanced functionality such as "pausing a transformation *when* rule R_i is encountered" or "*when* a pattern P appears in the model" are not natively supported.

In [SKV10a, SKV10b], Syriani et al. propose the modelling of exceptions and their handlers at the level of abstraction of Model Transformation Languages (MTLs). They first suggest a categorization of model transformation exceptions where errors in rule action and condition code are distinguished from invalid rule sequencing errors and from (unexpected) internal errors. They then describe how a control flow environment for rule execution can be extended with provisions for capturing and handling exceptions. Essentially, whereas traditionally such environments only allow rules to be sequenced, and for this sequencing to occur on one of two events, "rule applied" and "rule not applicable", the proposed transformation scheduling environment introduces *event handler* constructs which can be sequenced with rules on a third, new event: "rule failure". Finally, the authors also discuss how the *assertion* debugging primitive can be simulated within their framework. First, a rule R is created with its LHS pattern set to some violating sub-graph and its RHS post-application action code instrumented to produce a failed assertion exception. Second, the new rule is appropriately inserted within the existing rule sequencing. Then, when the transformation is executed, if there is at least one instance of the violating sub-graph in the source model when R is executed, the transformation will fail and report the failed assertion. Syriani et al. pave the way towards more modular, fault-tolerant and elegant rule and transformation scheduling designs. However, numerous debugging primitives such as *breakpoints*, *print statements* and *stepping* are left unexplored. Furthermore, a more detailed examination of their proposal for the simulation of assertions reveals that rule designers are exposed to a number of redundancies and awkward low-level details which suggests that higher-level constructs should be introduced.

Finally, in [KSWR09], Kusel et al. clearly enunciate the abstraction mismatch between model transformation design and debugging tasks as a factor in QVT's limited adoption. They fail to provide a convincing alternative however, proposing that QVT relations (i.e., QVT's equivalent to transformation rules) be manually mapped onto graphical Petri Net-like models that they argue can be (more) easily debugged with the naked eye. Whether or not this is indeed the case, a considerable shortcoming of their approach, other than its requirement for an undoubtedly lengthy, repetitive and error-prone manual translation step, is that bugs identified in the Petri Net-like representation must be corrected in the original, more complex QVT specification. In sum, their approach to model transformation rule debugging appears to be more of a commentary on QVT's accessibility than a practical, scalable and widely applicable solution to the problematic abstraction mismatch that motivated their work.

3.3 Debugging Code

Debugging forms a central part of any programming effort. Several authors have looked into common sources of bugs, into what makes certain bugs more insidious than others, and into popular debugging activities [Eis97, Zel09]. It turns out that observing information about program state as well as hand-simulation (i.e., interrupting execution to run part or all of a program one statement at a time) are often used for tracking down and resolving bugs. Means to carry out these activities are thus commonly provided by modern programming languages and their IDEs. Below, a brief overview is given of the most common debugging facilities featured in modern GPLs and popular IDEs.

3.3.1 Language Primitives

The following concepts are commonly used to create a "poor man's" debugger when a full-fledged debugger is not available.

Print Statements

Print statements are the central part of every developer's[5] first "Hello World!" program. Print statements (in debugging) are commonly used to output variable contents and to verify that other statements or code blocks are executed, or more generally, to trace the flow of execution.

Assertions

A slightly more advanced feature than print statements, assertions enable programmers to verify that arbitrary conditions are satisfied at a given point in the program's execution. Assertions traditionally have two particularities. The first is that they cause the execution to be aborted when their condition fails. The second is that the source language or its compiler usually provide means to enable or disable assertions (e.g., Java's compiler has a flag to enable assertions, compiling C++ in "release" mode rather than "debug" mode disables assertions). This implies that developers need not manually remove or comment out assertions in order to avoid undesired output and/or computation in deliverables, as is the case with print statements.

Exceptions

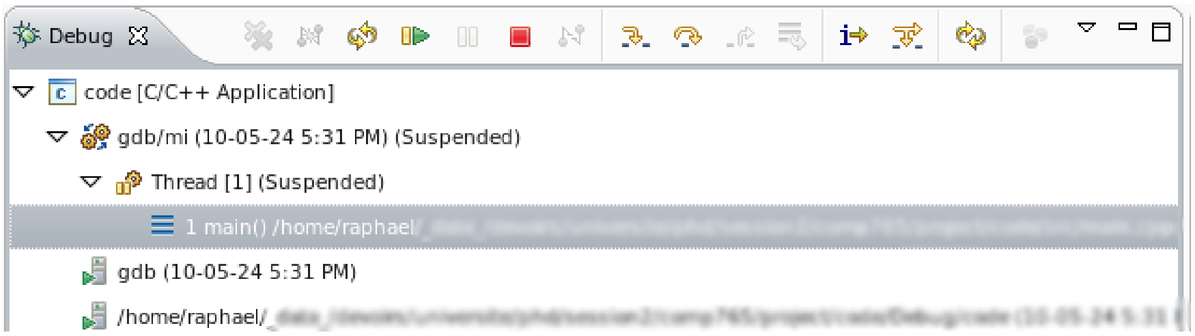
The third and most advanced enabler for debugging commonly found in programming languages, exceptions are *thrown* at runtime to indicate the system is in a problematic state.

[5]In the following, the terms *programmer* and *developer* are used interchangeably to describe both the code's implementer and the person debugging it.

An exception encapsulates information about the system's state as well as about the problem that occurred. Numerous exceptions are built into languages to report events such as I/O and arithmetic problems and null-pointer dereferencing. By default, exceptions halt the execution of a program. However, programmers can define exception handlers to *catch* exceptions and take appropriate action. Typically, such handlers either recover from the identified problem and continue (normal) execution, forward the caught exception higher up the call stack, or gracefully terminate. Lastly, provisions for defining new types of exceptions and their handlers are usually made available to developers to capture application-specific exceptional situations.

3.3.2 Debugger Primitives

The above primitives enable programmers to carry out many crucial debugging tasks. However, modern IDEs have recognized that print statements are an inefficient, time-consuming and, for lack of better word, messy means of observing state and monitoring execution flow. Commonly provided IDE facilities for carrying out these tasks are presented below. Figure 3.1 shows the debugging pane in the Eclipse IDE.



```
code [C/C++ Application]
  gdb/mi (10-05-24 5:31 PM) (Suspended)
    Thread [1] (Suspended)
      1 main() /home/raphael/[redacted]
  gdb (10-05-24 5:31 PM)
  /home/raphael/[redacted]
```

Figure 3.1: The Eclipse IDE debugging pane.

Execution Modes

Modern IDE debuggers support continuous (i.e., until either termination or user interruption) and line-by-line program execution, as well as terminating and non-terminating interruption via *play*, *step*, *stop* and *pause* commands respectively. Furthermore, programmers can often run their code in *release* or *debug* modes. In the former mode, only playing and stopping functionality is available.

Steps

There are usually three line-by-line, or step, commands. These are *step over*, *step into* and *step out* (or *step return*). The first executes the current statement as an atomic block. The second executes one sub-statement "contained" within the current statement (e.g., for a function call statement, contained sub-statements are those that form its definition), if any, thus effecting a change in scope. Advanced debuggers support stepping into the lower-level representations, if any, of seemingly atomic constructs. For instance, it is possible in Eclipse to step through compiled Java bytecode while executing a Java program. Of course, the available level of granularity (or abstraction level) is dependent on tool support and programming language implementation. Finally, stepping out causes continuous execution until the statement that was initially stepped into has returned.

Runtime Variable I/O

IDE debuggers usually provide means to read (and change) global and local variables when the program's execution is paused. This feature effectively removes the need to insert debugging-related print statements into the code. To our knowledge, no popular debugger imposes any constraints (beyond basic type constraints) on the values one can assign to variables when the execution is paused. Hence, it is possible to accidentally or purposefully place an executing program in an otherwise unreachable invalid state. In practice, observing variable values is far more common that modifying them. However, dynamic modifications can be instrumental to steer the execution, particularly when variable values are incorrect (e.g., due to a missing or faulty implementation).

Breakpoints

Breakpoints allow developers to specify *when* the debugger should interrupt normal program execution. This avoids having to step through the code line by line to reach the desired point. Breakpoints are commonly set on statements indicating that the execution should be paused before the debugger executes the given statement. Thus, they are essentially time-saving wrappers around the stepping commands, and can effectively replace print statements for the task of determining whether or not a given line is executed. Most debuggers also support associating *hit counts* or boolean conditions with breakpoints. These respectively enable the programmer to specify that a breakpoint should only halt the execution when its statement has been executed some given number of times or when a given condition over global and local variables is satisfied.

Stack Traces

Stack traces allow the programmer to see which function calls led the program into its current state. Stack traces become visible when the execution is paused. Most debuggers support navigating from the current context to that of any higher level in the stack trace (corresponding to the calling context) for variable viewing and editing purposes.

The concepts above are by no means an exhaustive list of all debugging facilities available to modern programmers. However, together they form an effective basis for writing debuggable code and debugging it.

3.4 Debugging in DSM: Debugging Model Transformations

The development process in DSM has two important facets: developing models and developing formalisms. The latter includes the specification of syntax and of operational or denotational semantics. Only semantics, however, require traditional debugging activities, as the verification of the correctness of a syntactic specification merely involves ensuring valid (i.e., conforming to the language's meta-model) models can be created while invalid ones can not, with inconsistencies easily traced back to their source. Both development facets introduce a crucial difference between the programming and DSM worlds. First, in the latter realm, artifacts to debug are no longer restricted to code and include model transformations, synthesized and hand-crafted models, and other arbitrary non-code artifacts. Second, the code realm counterparts of the very *common* DSM activities of designing and debugging model transformations, which are respectively designing and debugging GPL compilers/interpreters, are both specialized and relatively infrequent activities. Indeed, one can easily argue that the number of programs to debug (and their domains) is orders of magnitude larger than the number of existing compilers and interpreters for the programming languages these programs are implemented in. This is a side-effect of the nature of GPLs which are designed to enable programmers to develop applications for an unbounded set of domains. DSMLs, however, target restricted domains and as such the number of DSMLs, and by extension the number of DSML semantics specifications, is much more closely related to the number of DSms. A complete discussion about the debugging process in the context of DSM must thus consider the debugging of model transformations as a first class concern.

Regardless of the type of semantics (i.e., operational or denotational) a rule-based model transformation defines, it describes a scheduling of rules. Thus, the task of model transformation debugging involves the verification of the correctness of this scheduling and of the rules it refers to. Below, we explore how the concepts from the previous section translate to the debugging of formalism semantics specified via model transformations. In each case, it is assumed that the abstract purpose of the concept remains the same (e.g., print statements are still used to output variable state and verify if control flows through a certain part of the system), with focus placed on how to elegantly achieve the said purpose within the new context.

Print Statements

A contrived means of reproducing print statements for model transformations is to create rules with their action code set to appropriately parametrized calls to console[6] output functions in a supported action language (e.g., Python). In tools where scheduling is grammar-based (i.e., any applicable rule may run) and the user is free to choose between several

[6]The console may take many forms depending on tool support. For instance, in AToM^3 and AToMPM, a built-in console displays text printed from within action code.

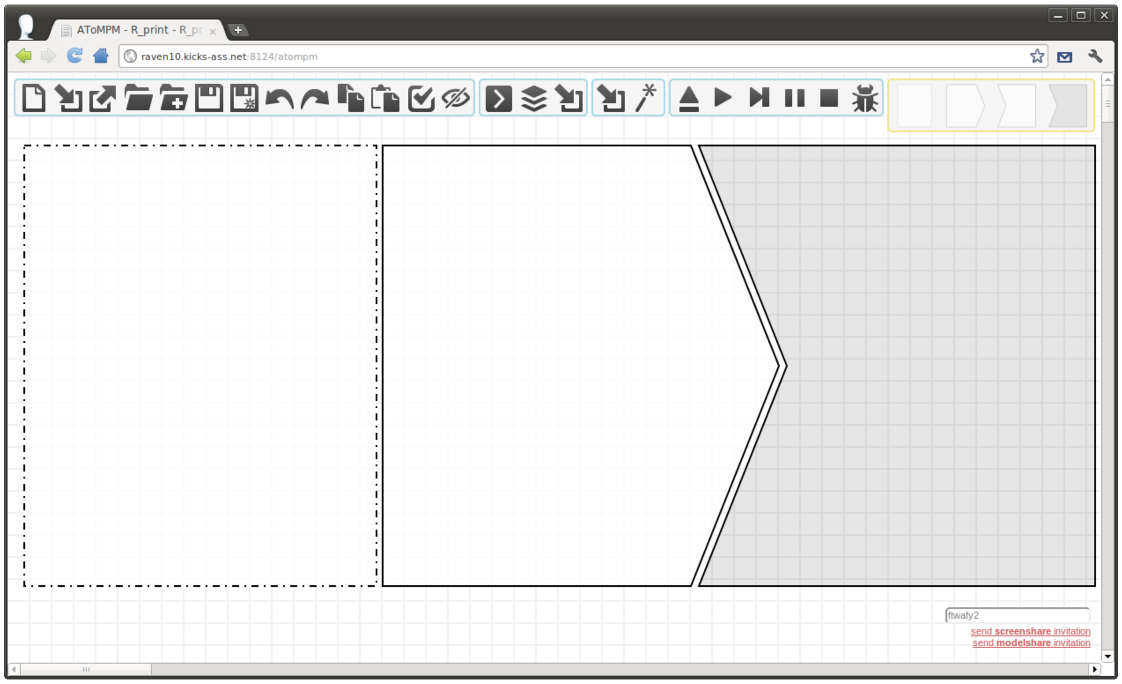
applicable rules, such as AToM[3], such a contrived rule could either be permanently enabled, allowing it to be run as needed by the developer, or it could specify a pre-condition pattern causing it to be applicable only when the given pattern is matched. Alternatively, in tools like AToMPM where rule scheduling is control flow-based, the rule could be arbitrarily sequenced with other rules. Once again, a pre-condition pattern could be specified to control its applicability. Unfortunately, the proposed technique is not devoid of accidental complexity. If a LHS pattern were to be specified, an identical RHS pattern would be required to avoid modifications to the source model. Also, means to prevent the rule from being executed repeatedly might need to be included in its condition and/or action code and/or as NAC patterns. A similar but neater and more domain-specific solution would be to enhance MTLs with printing functionality via a new type of rule: *PrintRules*. These would be included within the rule scheduling model and could be otherwise indistinguishable from any other rule. However, they should have different parameters: a LHS pattern, one or more optional NAC patterns, condition code and *printing code*, but no action code or RHS pattern. The natural semantics of such rules would be to print the result of executing the printing code if the condition code is satisfied when the LHS pattern is found in the host model and the NAC patterns are not. PrintRules could easily and automatically be translated to the contrived traditional rules described above using Higher-Order Transformation (HOT) rules[7] such as to leave the underlying model transformation execution engine unchanged. Figure 3.2b shows a sample rule scheduling where a PrintRule is sequenced between traditional rules. Figure 3.2a shows a tentative rendering of an empty PrintRule. The `printingCode` attribute is not visible, but the fact that no RHS pattern should be specified is made explicit.

Although it may seem counter-intuitive to enhance MTLs with support for a language construct whose usefulness is mostly restricted to debugging, we should remember that print statements, whose usefulness is mostly restricted to debugging, are supported in every modern GPL.

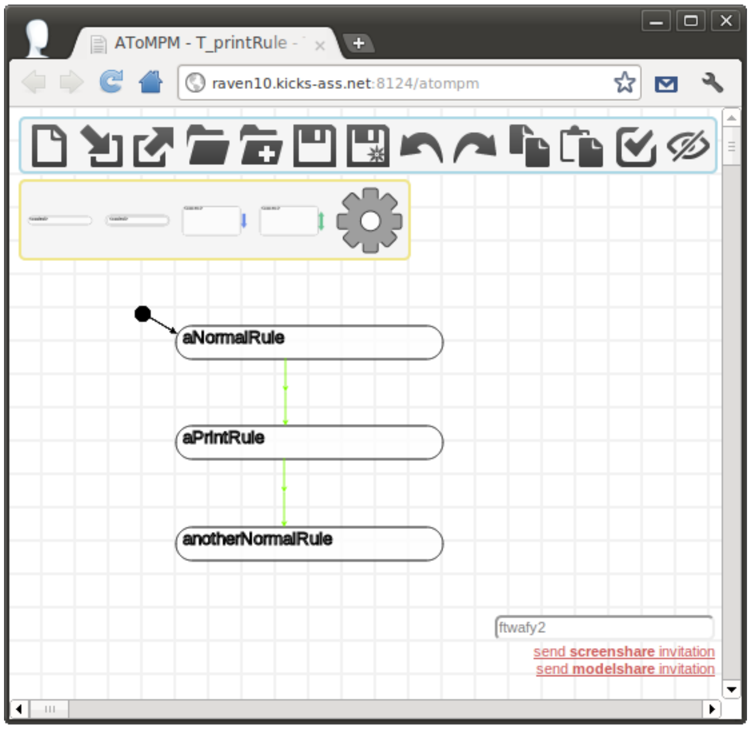
Assertions

Assertions in transformation models can be replicated in a manner very similar to print statements. The conditions they are meant to check can be encoded in the condition code and/or pre-condition patterns of traditional rules, while exception throwing code can be placed within the action code of the said rules. Thus, such a rule would only be applicable when the assertion's condition is met, and it would trigger an error reporting the failed assertion upon execution. Syriani et al. propose specifying assertions in just this manner in [SKV10b]. In case of failure, the rule's action code throws a user-specified exception that either interrupts and terminates the execution of the transformation or is caught and handled by exception handlers specified at the transformation language level. Unfortunately, forcing the developer to reproduce assertions using the default rule mechanism carries the same limitations as those mentioned for print statements; namely, the necessity of specifying identical

[7]Rules that take other rules as input and/or output.


AToMPM - R_print - R_pr
raven10.kicks-ass.net:8124/atompm
ftwafy2
send screenshare invitation
send modelshare invitation

(a)


AToMPM - T_printRule -
raven10.kicks-ass.net:8124/atompm
aNormalRule
aPrintRule
anotherNormalRule
ftwafy2
send screenshare invitation
send modelshare invitation

(b)

Figure 3.2: (a) An empty *PrintRule*. (b) An example transformation model where a *PrintRule* is inserted between two traditional rules.

LHS and RHS patterns and loop prevention mechanisms. Furthermore, requiring the manual entry of exception throwing code exposes the transformation developer to technical details which should remain hidden. A similar solution applies though. MTLs could be enhanced with assertion functionality via a new type of rule: *AssertRules*. Like PrintRules, these could be sequenced seamlessly with other rules within the rule scheduling model. They would be parametrized by a LHS pattern, one or more optional NAC patterns, condition code and an *assertion message*, and could be rendered similarly to PrintRules. Their natural semantics would be to throw the assertion message as an exception if the condition code is satisfied when the LHS pattern is found in the host model and the NAC patterns are not. Analogous reasoning as for PrintRules applies regarding the automated translation of AssertRules to traditional rules and the validity of their inclusion in MTLs.

An added benefit of AssertRules over the alternative (i.e., replicating them manually via traditional rules) is that they would facilitate the implementation of the most crucial property of assertions: the ability to enable and disable them without having to comment, remove or alter them. Indeed, such behaviour could be achieved through a trivial modification to the transformation execution engine or to the HOT rules that convert AssertRules into traditional rules. On the other hand, achieving this when dealing strictly with traditional rules would undoubtedly require the rule developer to somehow explicitly tag his rule such that the transformation execution engine could identify it as an assertion rule. This is clearly a less elegant solution, polluted by the accidental complexity introduced by missing MTL constructs. Furthermore, more advanced modelling tools could support showing and hiding assertions to improve the transformation model's readability. Such features would be increasingly difficult to implement if assertions where indistinguishable (or less easily distinguishable) from common rules.

Exceptions

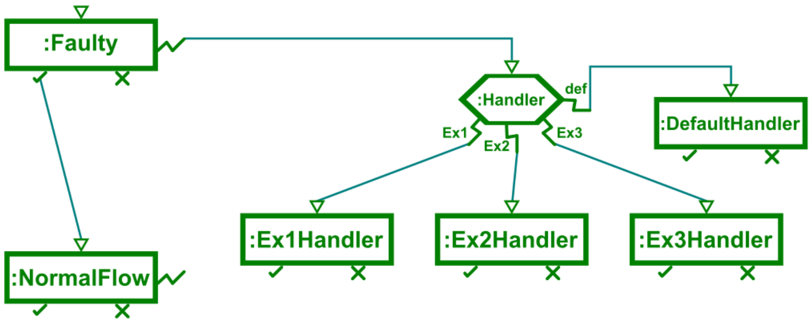
Exceptions and their handlers in the context of model transformation debugging were extensively studied in [SKV10b]. Syriani et al. provide a classification of several relevant exceptions that capture issues ranging from synchronization problems brought on by incorrectly parallelizing transformations to action language and rule specification errors. While most exceptions originate from within the transformation execution engine that throws them as it encounters problematic states, Syriani et al. also recognize that new user-defined exception types may be required. These can be thrown from within rule action code when application-specific exceptional situations arise. To handle the various exceptions that may result from a rule's execution, they propose enhancing MTLs with *handler blocks* and allowing an *exceptional* rule outcome. This outcome complements the existing "success" and "not applicable" outcomes. Rules can be sequenced to handler blocks from their exceptional outcome port. Handler blocks take the produced exception as input and then direct the flow of control to a traditional rule whose purpose is to attempt recovery or propagate the exception. This is a very elegant solution as no accidental complexity is introduced: new constructs specific to the problem at hand are introduced rather than clumsily attempting to mould existing constructs to a task they are not well suited to perform. Figure 3.3a, borrowed from [SKV10a], depicts a transformation model where traditional rules and handler blocks are interleaved. In a more interesting example, control might flow out of the success ports of the exception handler rules to rejoin the normal flow rule. Exception support in AToMPM is present but slightly limited: various exception types as well as the exceptional rule outcome are supported by the native MTL language but handler blocks have yet to be introduced. Thus, the handling of exceptions is less flexible than what is shown in Figure 3.3a, as evidenced in Figure 3.3b.

Execution Modes

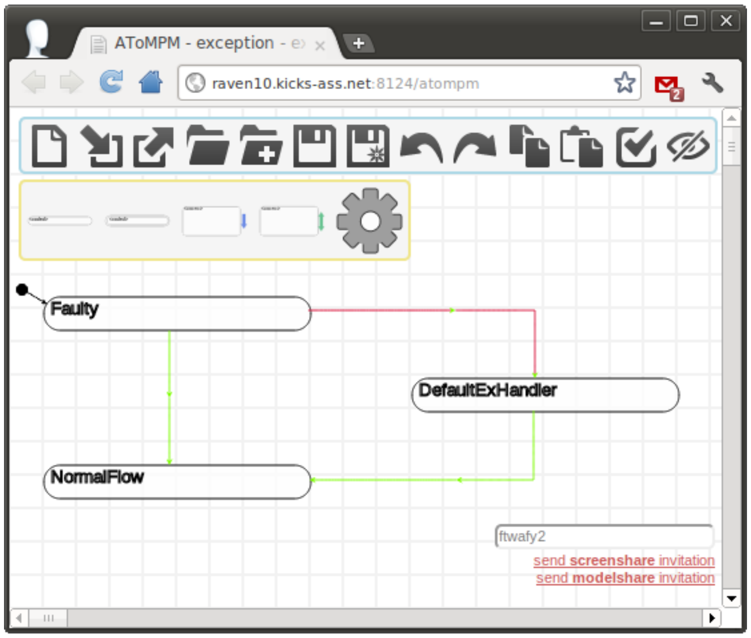
Certain tools, such as AToM^3 and AToMPM, natively support continuous and step-by-step execution modes. In the former mode, a model transformation is executed until it terminates, i.e., until no more rules are applicable or the implied or explicit scheduling has completed. In the latter mode, the user is prompted to run the remainder of the transformation in continuous mode or to step over a single applicable rule after every rule application. Thus, equivalents to the play, step over and stop facilities from modern code IDEs already exist in the realm of modelling tools. As for the pause functionality, to our knowledge, only AToMPM and VMTS (Visual Modeling and Transformation System) [LLMC06] offer means to pause the execution of an ongoing transformation. The semantics of such a facility is ambiguous though. While pausing a transformation exactly between the application of two rules should intuitively delay executing the next rule until the user chooses to continue in either continuous or step-by-step mode, what should be the appropriate response if the pause request were to be triggered *during* the application of a rule? In a system where rule application is implemented following a transactional approach, sensible behaviours might be to let the current rule complete or to roll-back to before its application before pausing. The former is the default behaviour in AToMPM. In a system based on T-Core [SV10], where rules are no longer atomic blocks but are instead arbitrarily composed sets of primitive components (e.g., *matchers*, which locate a rule's pre-condition pattern in the host graph, *rewriters*, which effect the transformation of a given match into the rule's post-condition pattern), ideal behaviour might be to pause the execution before invoking the next primitive operation. A third option is immediate interruption. Either choice has its merits and is heavily dependent on MTL and tool features. In determining which approach to choose, it seems sensible that pausing only occur when the system state is *consistent* and *observable*. For instance, it seems unreasonable to offer immediate interruption in a context where rule application is atomic and no facilities for observing any sort of meaningful intermediate system state are available. Finally, model transformations, like programs, should be executable in both debug and release modes, with pausing and stepping functionality disabled in the latter.

Steps

The notion of "stepping over" in the context of rule-based model transformations is trivial. It corresponds to the execution of one, possibly composed, rule. Stepping into, however, is



(a)


AToMPM - exception - e
raven10.kicks-ass.net:8124/atompm
Faulty
DefaultExHandler
NormalFlow
ftwafy2
send screenshare invitation
send modelshare invitation

(b)

Figure 3.3: (a) Exception handling via *handler blocks* and traditional rules. (b) Exception handling in AToMPM, via traditional rules only.

very dependent on MTL and tool features, much like the aforementioned pause operation. For instance, in languages such as MoTif, GReAT and QVT, and in tools like AToMPM, VIATRA [VB07] and VMTS which support rule composition (i.e., the composition of several rules together into "super rules"), stepping into a composite rule would conceivably allow the developer to "enter" the rule and execute its sub-rules one at a time. This process could of course be repeated recursively in the case of several layers of composition. For more advanced languages like MoTif-Core [SV10] – or any other T-Core-based language – where rules may not only be composed but also decomposed, it is sensible for the developer to even be able to step into a non-composite rule. Doing so would enable executing each primitive rule component in turn and possibly performing such actions as modifying a match before it is given to the rewriting component. Finally, "stepping out" obviously only makes sense for MTLs and tools where "stepping into" is meaningful. In such languages, stepping out would perform a very analogous task to that which it performs in code debugging. Stepping out of an inner rule would cause the continuous execution of any remaining rules within the parent composite rule. Stepping out of a T-Core primitive would cause the continuous execution of any remaining primitives within the parent rule. Note that the only stepping operation supported by AToMPM at this time is "step into".

Runtime Variable I/O

The notion of what is a variable in the context of model transformations is unclear. The parameters of a rule or the sequencing of the rules themselves may all be considered variables. Thus, when the execution is paused, the developer could choose to edit rule LHS, RHS and NAC patterns, condition code, action code and even the scheduling of the transformation rules. All of the previous propositions share a crucial requirement though: if the rules and/or the transformation model need to be compiled before they can be executed by the transformation engine, modifying either of them on-the-fly might be infeasible[8]. This is of course analogous to modifications to running C++ or Java code being ignored by IDE debuggers until the next compilation. Data flow- (e.g., GReAT, MoTif) and T-Core-based (e.g., MoTif-Core) languages lend themselves more intuitively to traditional runtime variable I/O operations because of their notions of inputs and outputs. It is conceivable that the developer may want to observe or change the inputs or outputs of a rule or T-Core primitive. This implies that a model transformation debugging tool should clearly expose these variables and offer means to view and modify them. Moreover, this should be done at the appropriate level of abstraction. For instance, if a developer wants to observe the input sub-graph of a T-Core rewriter block, it should be represented using domain-specific constructs rather than in some internal format. Similarly, if a developer wants to modify a sub-graph, he should be provided with a domain-specific model editor pre-loaded with appropriate meta-models. Finally, as is the case for code debuggers, ensuring that runtime modifications don't leave the system in an unstable or otherwise unreachable state is a desired but complex task. In

[8]To our knowledge, no existing modelling or model transformation tools currently support on-the-fly rule scheduling modifications. However, AToMPM interprets rule schedules (as opposed to compiling them) and could thus be relatively easily made to support on-the-fly scheduling changes.

this context, one such verification could be to ensure that the modified input of a T-Core rewriter block still contains exactly one instance of the specified pre-condition pattern.

Breakpoints

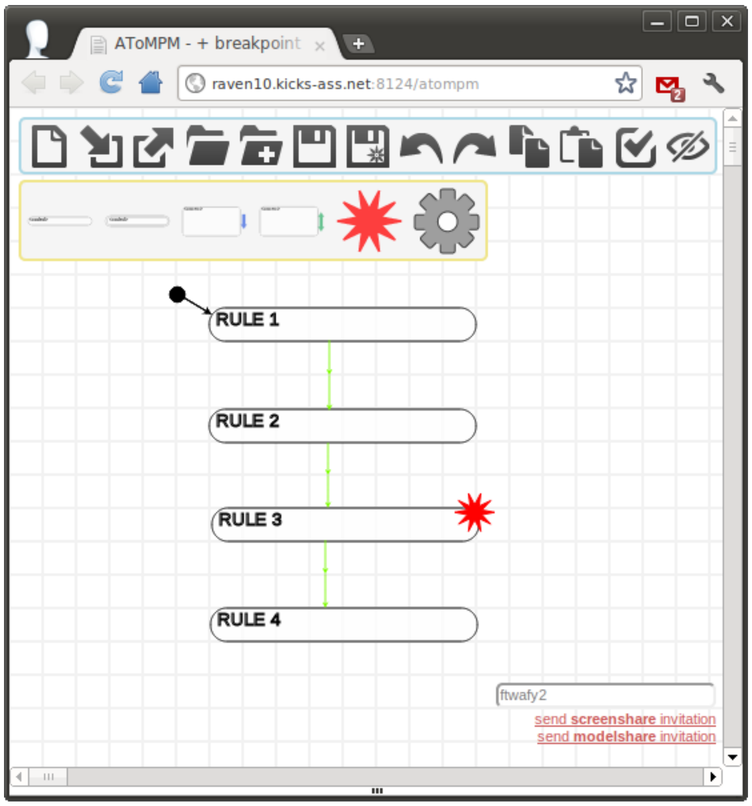
Breakpoints very intuitively translate to the model transformation debugging world. Given proper tooling, breakpoints could be associated to composite rules, non-composite rules and even primitive T-Core blocks causing the execution to pause when they are encountered. Hit counts and boolean conditions (written in a supported action language) could conceivably be specified in the same manner as in IDEs. Figure 3.4 depicts a transformation model with a breakpoint set on one of its rules, as well as its property dialog, which indicates that it is enabled and that it should always be triggered. Note that breakpoints are not yet fully supported by AToMPM.

Stack Traces

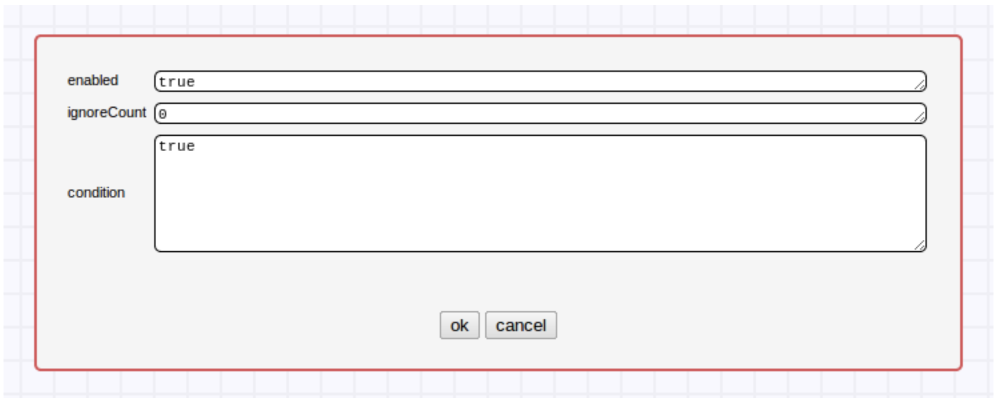
Like many of the discussed concepts, the availability of stack traces and their meaning depend heavily on MTL and tool features, specifically, on the levels of composition and granularity they support and expose. Hence, for languages where rules can neither be composed nor decomposed, stack traces would do little more than report the current rule and would thus be of very little use. However, for more complex languages, they should display a cascading view from top-level transformations down to the currently executing rule or T-Core primitive. They should also enable navigation between the contexts of sub-rules and their parent composite rules, and between the contexts of T-Core primitives and their enclosing rules.

AToMPM provides a facility similar to stack traces. While in debug mode, relevant contexts are loaded into new modelling windows as they are encountered and currently executing steps within them (where steps may be rules, composite rules or entire transformations) are clearly identified. This is depicted in Figure 3.5, which shows a snapshot of the debug mode execution of a sample transformation with two levels of nesting.

Every concept from the previous section has been revisited in the context of debugging DSML semantics specified via model transformations. The usefulness of such a mapping may be called into question by the unfortunate reality that it is very common in modern DSM efforts for semantics *not* to be specified via model transformations, instead privileging coded artifact generators. On the one hand, debugging concepts from the code world already apply to these generators as they are themselves specified as code and as such, no concept mapping is required. On the other, the fact that coded generators violate MDE best practices and MPM principles, coupled with their numerous disadvantages and with the equally numerous benefits of structured rule-based model transformation approaches, both listed in Chapter 2, leads to the assumption that they will eventually fall out of favour. The availability of (complete) debugging facilities or of blueprints to build them for model transformations will then be of paramount importance. Furthermore, their immediate availability may help hasten the


AToMPM - + breakpoint
raven10.kicks-ass.net:8124/atompm
RULE 1
RULE 2
RULE 3
RULE 4
ftwafy2
send screenshare invitation
send modelshare invitation

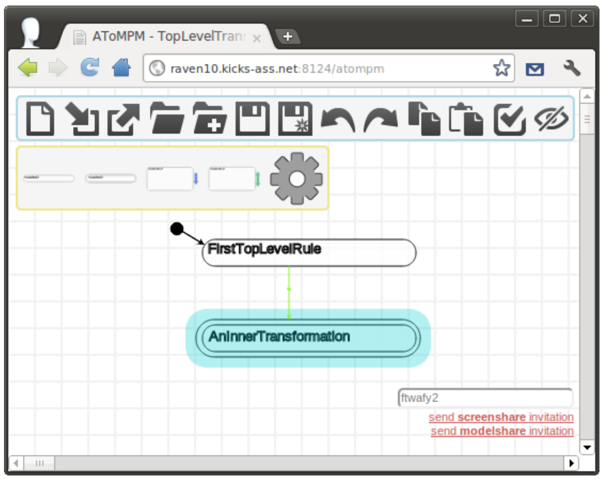
(a)



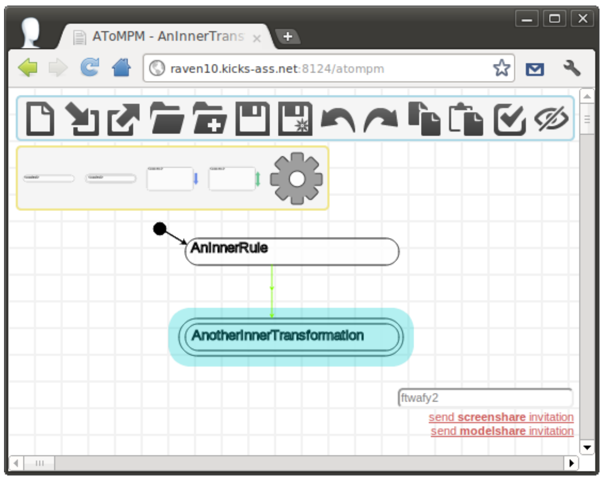
enabled `true`
ignoreCount `0`
condition `true`

(b)

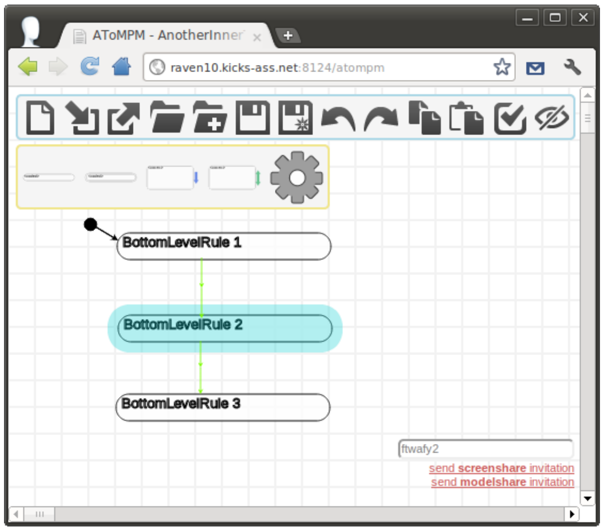
Figure 3.4: A breakpoint in a transformation model, and its property dialog.


AToMPM - TopLevelTran
raven10.kicks-ass.net:8124/atompm
FirstTopLevelRule
AnInnerTransformation
ftwafy2
send screenshare invitation
send modelshare invitation

(a)


AToMPM - AnInnerTrans
raven10.kicks-ass.net:8124/atompm
AnInnerRule
AnotherInnerTransformation
ftwafy2
send screenshare invitation
send modelshare invitation

(b)


AToMPM - AnotherInner
raven10.kicks-ass.net:8124/atompm
BottomLevelRule 1
BottomLevelRule 2
BottomLevelRule 3
ftwafy2
send screenshare invitation
send modelshare invitation

(c)

Figure 3.5: A model transformation debugging trace. (a) The top-level transformation with its current step highlighted. (b) The inner transformation from (a) with its current step highlighted. (c) The inner transformation from (b) with its current step highlighted.

DSM community towards more principled approaches to semantics specification. Finally, the next section's discussion regarding DSm and artifact debugging reveals that numerous and complex artifact instrumentations are required for debugging at the DSm level to be possible. However, Chapter 2 made clear that coded generators grow in complexity and decrease in modularity at a much faster pace than the complexity of the artifacts they produce, further motivating a shift towards semantics specification through model transformations.

3.5 Debugging in DSM: Debugging Models and Artifacts

Despite the importance and need for model transformation debugging facilities, they may not be sufficient for debugging models whose semantics are specified denotationally. Indeed, running such models implies the execution of synthesized artifacts[9] rather than of model transformations. Although not always the case in academic DSM efforts, it is sensible to assume that in industry, DSMLs (and their semantics) will often be defined by actors different than the end-users of the DSMLs. Similarly, the vast majority of C++ and Java programmers played no part in the development of those languages or their compilers, nor do they possess the required skills. Thus, in the following section, a distinction is made between two types of DSM users. On the one hand, *designers* are fully aware of and understand the model transformations that describe DSML semantics and generate lower-level artifacts. On the other, *modellers* have an implicit understanding of the semantics of their DSms via their mastery of the problem domain, and while they should be aware of the end-products generated from their models, there is no reason for them to have any knowledge about how these are produced. For instance, consider the running example from Chapter 2. A designer should fully understand DSms, all intermediate representations (including physical code files) and the traceability links between them, executable applications running on target devices, and all of the involved semantics model transformations. However, a modeller should only be concerned with DSms and generated executable applications.

As a consequence of their different expertise, the debugging scenarios for designers and modellers differ. Designer artifact debugging is akin to compiler/interpreter (and bytecode) debugging in the programming world. The goal being to ensure the correctness of the output of the DSm-to-artifact model transformations, or, more generally, to ensure the correctness of the compiler/interpreter's result. Modellers, on the other hand, may assume that available DSm-to-artifact model transformations are flawless and must instead establish the correctness of their models. Thus, their task is more akin to traditional code debugging but with model rather than code being compared to an executing artifact. Whether DSms and artifacts are debugged by modellers or designers, the basic workflow will entail observing the changes undergone by the DSm as the synthesized artifacts are executed. Designers may

[9]Although numerous non-executable artifacts can be generated from DSms, we restrict our attention here to programs and models.

wish to pay special attention to intermediate representations, if any. Note that debugging DSms and artifacts at the DSm level hinges on the availability of facilities to propagate state information from artifacts back to DSms. Similarly, the key advantage of IDE debuggers are the means they provide for variables at the GPL level to take on observable (via runtime variable I/O debugger facilities) values as a result of program execution. Without such facilities, it would be impossible to shield modellers from the internal details of synthesized artifacts other than by enhancing DSMLs with print statement functionality, which may often be undesired. Luckily, the approach to artifact synthesis presented in Chapter 2 fully enables the sort of artifact-to-DSm communication required in this context. Given that such communication is possible, we re-visit the debugging concepts described in Section 3.3 and explore how they translate to the debugging of DSms and synthesized artifacts.

Print Statements

The necessity of print statements is closely related to the aforementioned artifact-to-DSm state propagation functionality. When it is available, print statements are considerably less useful as the execution's path through the DSm can be explicitly observed (e.g., recall the DSm animation demonstrated in Figure 2.25) and changing construct properties can easily be read[10]. Nevertheless, it is conceivable for a modeller or domain expert to require explicit output for debugging purposes. A similar solution to the one proposed for the model transformation context may apply. DSM tools could provide means to specify print rules that output arbitrary information about models upon detecting conditions specified by the modeller in the form of patterns (e.g., `State` *s* is enabled, `Place` *p* contains at least *i tokens*). Recall though that modellers may have no knowledge about model transformations and might need to be introduced to the foreign concepts of rule patterns, conditions and actions. An added challenge from a tool implementer's point of view is that means to verify rule applicability and execute them concurrently with artitfact execution would be required. Print rules as described here are very powerful: they can essentially print anything, anytime. However, they are somewhat orthogonal to the rest of the DSm creation process. A better solution might be to (semi-)automatically integrate appropriate output constructs in DSMLs themselves at DSML design time. This might be achieved by adding a `print` attribute to language constructs. Then, appropriate extensions to the language's semantics would cause artifacts to be instrumented with instructions that output relevant information about DSm entities during execution. Alternatively, entire new `Printer` constructs could be introduced to languages with provisions to connect them with other domain-specific constructs. Again, language semantics would need to be co-evolved. Extending DSMLs in this manner is more closely related to what was proposed for MTLs and what is done in GPLs. Moreover, this would certainly be much more accessible to modellers than model transformation rules.

[10]Modern model editors provide effective runtime variable I/O facilities.

Assertions

Assertions can be thought of in a very similar manner as print statements: they can be implemented via orthogonal assertion model transformation rules or as explicit DSML constructs. In the latter case, the discussion about print statements naturally applies. In the former case, however, other considerations must be taken into account. First, provisions to enable and disable assertions must be provided by the modelling tool. Second, assertions should halt artifact execution upon failure. At the very least, this second point implies that synthesized artifacts should be (automatically) instrumented with callable “kill switches”, i.e., interfaces enabling remote termination. Note that assertions in this context are restricted to conditions that are verified during model execution, like *invariants*, not during model editing. Indeed, static meta-model constraints should not be confused with assertions.

Exceptions

The notion of exception in the context of debugging models is somewhat deferred. The reason for this is that exceptions occur at runtime and that although models may be made to appear to be executing, what is truly being executed are synthesized artifacts. Consequently, exceptions originate from these artifacts and are described at their level of abstraction. Such exceptions may be caused by I/O errors, by bugs in third party code libraries, by errors in modeller-provided attribute values at the DSm level, etc. – but not by errors in denotational semantics transformations as these are assumed correct. A key concern in the DSM debugging context is that exceptions that are propagated back to modellers be presented at the level of abstraction of DSms. This follows from the recurring notion that DSM can only successfully claim to raise abstraction if modellers are entirely shielded from artifact-level information. Some exceptions may be straight-forward to translate back into domain-specific terms. Others may describe transient issues that are entirely irrelevant to the modeller. Others may describe issues that are only of interest to the designer. Determining which category exceptions fall into is a task best suited for the DSML designer.

Given a categorization of exceptions and their target recipients, proper handling and propagation mechanisms must be put in place. Handlers should be inserted into artifacts during their synthesis (e.g., via designer-specified model transformation rules) to capture any conceivable exception. For modeller- and designer-irrelevant exceptions, which by definition are easy to recover from, handlers should return artifacts to normal execution flows. For relevant exceptions, handlers should effect exception translation to domain-specific terms and propagate their results to the modelling tool for presentation to the designer and/or modeller. Very similar tasks have already been demonstrated in Chapter 2. Indeed, the propagation of nicely formatted performance measurements from artifact to DSm involves a translation (e.g., producing a labelled and coloured textual message from a numeric measurement in milliseconds) and a propagation (e.g., transmitting the produced summary to the modelling tool for relevant entity tagging) step. Similarly, exception handlers in synthesized artifacts should translate caught exceptions into accessible (and attention-catching) messages and transmit these to the modelling tool such that the culprit(s) at the DSm level may be tagged with them. In principle, these messages should be sufficiently expressive for a modeller and/or designer to identify and correct the problem at the DSm level. This is thus yet another scenario where communication-enabling traceability information between artifacts and DSms is instrumental. Last but not least, means to present designer-relevant exception messages only to designers hint at the need to further refine the traditional debug mode into separate debugging modes for modellers and designers.

Execution Modes

The play, pause, stop and step commands each require communication to be possible between modelling tools and artifacts. Playing and stopping requires means to remotely start and terminate synthesized artifacts, respectively. This may mean the launching of an operational semantics model transformation (e.g., for model artifacts) or the closing of a program (e.g., for code artifacts). In practice, supporting even such seemingly trivial operations may be arduous as modelling tools must know how to launch a wide variety of artifacts (e.g., remote Google Android applications, Petri Net models), and generated artifacts must be responsive to termination commands. The former of these challenges can be met through tool plug-ins. Given tool support, arbitrary plug-ins can be added to enable the launching of artifacts from an unbounded number of target platforms. As for generating responsive artifacts, model transformation rules could trivially instrument artifacts with command interfaces, with commands being transmitted from modelling tool to artifact over one of many suitable protocols (e.g., SOAP, CORBA, HTTP). Incidentally, such an interface is instrumental in implementing the step and pause commands discussed below.

The meaning of taking one step in an arbitrary DSm is unclear. This ambiguity is compounded by the fact that the notion of behaviour, if any, may vary considerably from one DSML to the next. To this end, we propose a very general and widely applicable definition: *a step constitutes any modification to any attribute of any entity in a DSm, as well as entity creation and deletion.* Given this definition, an intuitive solution for step-by-step (as opposed to continuous) artifact execution emerges: in step-by-step mode, execution should halt after every state change propagated from artifact to DSm, as each of these constitutes a step at the DSm level. Yet again, instrumenting this behaviour into artifacts can be automated via added rules in denotational semantics model transformations. For code artifacts, such instrumentation might include thread *sleeping* and *waiting* commands.

Finally, pausing presents similar challenges as in the model transformation debugging context. The ideal behaviour of a pause thus depends on the structure and semantics of the DSm and artifacts. For modellers, a widely applicable and sensible approach is to interrupt the execution before beginning what would have been the next step at the DSm level. However, for designers, immediate pausing (i.e., pausing before executing the next step at the artifact level) may be desired and/or necessary. The undeniable need for these distinct pausing modes further motivates the previous proposal of separate debugging modes for designers and modellers.

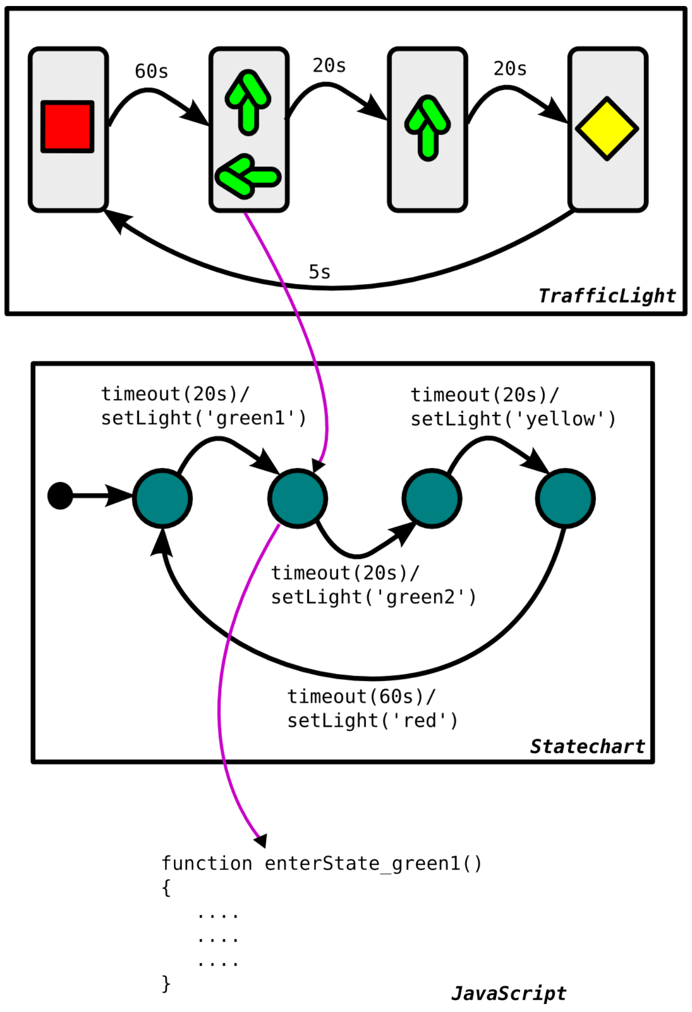
Steps

The notions of stepping over, into and out can be considered from two orthogonal perspectives: the modeller's and the designer's. On the one hand, all three notions intuitively translate to DSMLs which support composition hierarchies. Stepping into and out should enable downward and upward navigation within the hierarchy, respectively, while stepping over should enable the atomic execution of composite or non-composite constructs. For DSMLs with no notion of composition, only the stepping over functionality should be enabled, with each step executing one step at the DSm level. On the other hand, the fact that artifacts are generated from DSms creates an implicit hierarchy between them. Navigating this hierarchy might be significantly more interesting for designers. In this scenario, stepping into should take a step at the level of the next lower-level formalism, if any. This is depicted in Figure 3.6 where two successive "step into" operations lead a designer from a domain-specific traffic light model entity to a corresponding Statechart `State` and finally to a statement in generated code. Notice how both the availability of intermediate representations between DSms and artifacts, and of traceability links[11] between them make the debugging process more intuitive and incremental. Indeed, stepping into code directly from the DSm might be disconcerting even for designers. Moreover, following traceability links makes answering complex questions like "which lower-level entity does the current entity correspond to?" trivial. As for "stepping out" and "over", the former should trigger continuous execution until the designer is brought back to higher-level entities, while the latter should perform a step at the current level of abstraction (e.g., one code statement, one operational semantics rule). Once again, the need for distinct debug modes for designers and modellers is made explicit.

Runtime Variable I/O

At first glance, runtime variable I/O seems the most straight-forward of the debugger operations. For DSms and lower-level generated models, if any, the model editing tool itself can be used to view and modify variable values when the execution is paused. For synthesized code artifacts, if any, executing them within IDEs enables the reuse of IDE debugger runtime variable I/O facilities. Thus, means to manipulate state at all levels of abstraction are available to modellers and designers. The hidden challenge, however, is to propagate changes between artifacts, intermediate representations and DSms to ensure consistency across all views of the system. The technique to artifact synthesis presented in Chapter 2 provides effective means to address this challenge. These have been successfully prototyped in AToMPM in the context of the PhoneApp DSML. However, replicating such functionality while using coded artifact generators and/or without traceability links is clearly a complex and error-prone task.

[11]To avoid clutter, only those traceability links relevant to the example are shown.



```
function enterState_green1()
{
   ....
   ....
   ....
}
```

Figure 3.6: Stepping into from the *designer*'s perspective. Stepping into a *TrafficLight* model entity leads into a corresponding Statechart `State`. Further stepping into leads into generated code.

Breakpoints

Breakpoints translate rather intuitively to the model and artifact debugging world. Ideally, it should be possible for the designer to specify breakpoints at any level of abstraction from DSm to artifact. Of course, the modeller is only expected to specify them at the DSm level. The challenge here is *how* to specify the breakpoints. For modelling languages with an explicit representation of state (e.g., Statechart, PhoneApp), the trivial answer is to mark certain states with breakpoints. The challenge then shifts to tool support for the availability of means to mark model entities as breaking, and for the automated emission of pause commands as marked entities gain focus. However, for languages with an implicit representation of state (e.g., a Petri Net model's state is a mapping between all of its `Places` and the number of tokens they contain, called a *marking*), a breakpoint can not be set on a state by marking any given entity. For such formalisms, we propose that breakpoints instead be specified as patterns (e.g., a partial or complete Petri Net marking) causing the execution to pause when the patterns are encountered. Such an approach obviously hinges on tool support for specifying and evaluating *breakpoint patterns*. Furthermore, to avoid overly affecting performance, optimizations should be put in place to prevent unnecessarily re-evaluating all breakpoint patterns at every modification. Significant research is warranted to fully comprehend and solve this problem, which is beyond the scope of this thesis. Last but not least, breakpoints, specified as entity markings or patterns, should still be parametrized by hit counts and boolean conditions.

Stack Traces

Like in the code and model transformation debugging contexts, stack traces in the context of model and artifact debugging are tightly coupled with stepping into and out functionality. In the designer debug mode, stack traces should make explicit which DSm and intermediate representation entities correspond to the executing artifact's active "statement" (which may be a line of code or one or more model entity). In modeller debug mode, stack traces should show all of the parent constructs, if any, of the currently focused construct.

Every concept from Section 3.3 has been revisited in the context of debugging DSms and artifacts. Beyond the numerous proposals, discussions and suggestions, two recurring points emerge. First, synthesizing artifacts via model transformations, specifically ones that maintain traceability links between corresponding constructs at different levels of abstraction, is the key enabler for nearly all of the surveyed debugging facilities. Second, artifacts need to be instrumented to receive and emit commands and data if debugging is to be carried out from the modelling tool and without forced interaction with synthesized lower-level representations and artifacts.

3.6 Comparison with Related Work

This section reviews and situates the contributions from this chapter with respect to relevant work by others surveyed in Section 3.2.

Little research has focused on debugging in DSM. Debugging of DSML semantics and of DSms is still mostly a trial and error hands-on process systematically carried out at the artifact level of abstraction due to lacking tools and/or excessively low-level artifact generators. Although various tools and languages provide facilities that can be re-used for debugging, limited attention has been paid to replicating code debugging facilities in the DSM world.

Wu et al. proposed means to debug textual DSms and code artifacts at the DSm level within a specific tool, and from the modeller's perspective only. In contrast, the conceptual mapping and suggestions we provide are valid for textual and visual DSms and for code and model artifacts, they are not bound to any specific tools, instead clearly identifying necessary and/or desired tool features, and they address all facets of debugging in DSM.

As for Syriani et al., their work clearly established how exceptions and assertions could be incorporated into model transformation languages. Our work is enabled by and is a super-set of theirs as it extends to numerous other debugging concepts (and is not restricted to model transformations).

Kos et al. developed a fully featured DSm debugger that supports breakpoints, runtime variable I/O, print statements and more at the DSm level. Their work closely follows our past publication on the work from this chapter. Though their work presents an appealing real world realization of a DSm debugger, it is limited with respect to our proposals in two ways. First, the Ladybird debugger is intricately tailored and bound to a single tool and DSML. Second, Kos et al. give no mention as to the inner workings of their debugger (i.e., as to how they went about implementing breakpoints, etc. at the DSm level). Among other inconveniences, this makes their work difficult to reproduce.

3.7 Conclusion and Future Work

The work presented in this chapter was motivated by the archaic means that state-of-the-art DSM projects rely on for debugging. Indeed, manual instrumentation of coded generators and synthesized artifacts is common practice. This is akin to programmers being required to insert print statements into compilers and compiled forms of their programs to debug their code. The argument is easily made that GPLs would not have had such a tremendous impact on productivity if code debugging were carried out in such a convoluted manner.

To guide the development of much needed debugging facilities in the context of DSM, we propose a conceptual mapping from debugging concepts in the software development realm to the DSM realm, while discussing and evaluating possible implementations for each concept. This mapping distinguishes between the two different facets of DSM debugging: the debugging of operational and denotational semantics (specified as model transformations) and the debugging of DSms and artifacts. We made a further distinction between debugging scenarios for two types of DSM users. *Designers* are DSM experts who are fully aware of and understand the model transformations that describe DSML semantics and generate lower-level artifacts. *Modellers* are arbitrary domain experts who must be shielded from anything at a level of abstraction below that of their DSms.

Our work demystifies the amount of effort required to build DSM debuggers: numerous model transformation debugger features can be integrated into MTLs and their execution engines, while structured and traceability-focused artifact synthesis can considerably facilitate the implementation of numerous DSm and artifact debugger features. AToMPM, in its current state, demonstrates a number of the suggestions we propose.

Finally, completing the implementation of a full-fledged debugging environment in AToMPM and the further study of the meaning of breakpoints in the context of implicit state formalisms is deferred to future work.

Chapter 4

Domain-Specific Engineering of Domain-Specific Languages

This chapter introduces a novel approach to the engineering of DSMLs that raises the levels of abstraction at which their syntax and semantics are specified. The proposed approach is rooted in DSM principles and treats traditional DSML syntactic and semantic specifications as target domain artifacts to be synthesized from higher-level DSms whose problem domain is that of modelling language design. The insight that DSMLs commonly re-use fragments of certain modelling formalisms to allow for state-based, data-flow based, etc. modelling leads to the identification of a collection of *base formalisms* that are leveraged as problem-domain constructs. A template-based approach for combining these is introduced, which effectively enables DSML specification in terms of base formalisms in a manner that unifies syntactic and semantic concerns. Means to automatically translate these DSms of DSMLs into traditional syntactic and semantic specifications (for use in traditional tools) are described. The approach is demonstrated by means of two example DSMLs, one of which is PhoneApp.

4.1 Problem Statement and Outline

Despite the stated merits of DSM, the guiding principles that enable full artifact generation from DSms have yet to be incorporated into the workflow of DSML engineering. Although much effort has been spent on enabling domain experts with powerful and high level facilities, the implementation of these facilities remains fixed at a rather low level of abstraction. On the one hand, DSML abstract syntax specification is often an ad hoc process devoid of guidelines and reusable building blocks. On the other hand, artifact synthesis from DSms is still commonly performed via complex hand-coded and ad hoc text generators which manipulate internal model representations via tool APIs. Thus, the definition of both the syntax and semantics of modern DSMLs bears more resemblance to programming than to DSM.

Although recent research (including that presented in Chapter 2) has proposed improvements to the modularity and efficiency of the traditional approaches to DSML syntax and semantics specification, its impact on the overall elevation of the level of abstraction of DSML engineering has been limited due to the failure to bridge the vast divide between syntax and semantics that characterizes modern DSML design. An obvious manifestation of this divide is the *separate* specification of syntax and semantics: models of DSML abstract syntax hold no information about the language's semantics. Consequently, an effective means to bridge the gap between DSML syntax and semantics is to make explicit the conceptual links between the former and the latter in a unified representation.

Abstract syntax of a DSML is commonly specified as a UML Class Diagram meta-model. An undesired consequence of this is that – much like common UML models of programs do not hold sufficient information to generate complete program code – these meta-models do not hold sufficient information for the full semantics of the modelled languages to be automatically inferred and complete applications to be synthesized. The manual definition of DSML semantics is both non-trivial and repetitive: it is conceivable that numerous modelling languages share some semantics. Consider, for instance, the subset of all DSMLs where the notions of state and transition exist. It is reasonable to assume that model transformations or coded generators for these languages will have some amount of similarity and even overlap. Non-trivial yet repetitive problems are often prime targets for automation via DSM. In this context, we wish to create explicit models of DSMLs and produce from these models DSML abstract syntaxes (i.e., appropriate UML Class Diagrams and constraints) and semantics (i.e., DSm-to-artifact transformations). We believe that a number of well-understood formalisms that encompass commonly recurring concepts in DSMLs (e.g., Statechart for reactive behaviour, states, and transitions [Har87, HK04]) and for which precise semantics exist can form a basis for such modelling of DSMLs. Challenges then include the selection of these *base formalisms*, the identification of means by which they may be leveraged and combined to create models of new DSMLs, and how these DSML models may be transformed into explicit abstract syntax and semantics specifications.

The vast number of conceivable DSMLs and the syntactic and semantic similarities between them naturally invite the application of DSM principles to ease, elevate the level of abstraction of, and partially automate the DSML specification process. Our main contribution is the introduction of a novel approach to defining DSMLs that elevates the level of abstraction of their specification, builds on existing low-level formalisms, makes conceptual links between syntax and semantics explicit, enables automatic synthesis of abstract syntax models and DSm-to-artifact model transformations, and effectively alleviates much of the burden traditionally placed on DSM experts.

The rest of this chapter is structured as follows. In Section 4.2, we survey relevant research work. Section 4.3 explains the rationale behind our novel approach to DSML engineering and proposes a set of low-level modelling formalisms that form a basis for (re-)constructing – ideally – any conceivable DSML. Section 4.4 shows how new DSMLs can be specified in terms of the base formalisms from Section 4.3, and how these DSML specifications are transformed into meta-models and model transformations that respectively capture DSML abstract syntax and semantics. Section 4.5 describes in detail how two non-trivial DSMLs can be defined using our novel approach, and explicitly shows their generated abstract syntax and semantics specifications. Section 4.6 discusses the extensibility, applicability and limitations of our approach, and provides a critical comparison with traditional DSML design and engineering practices. In, Section 4.7, we compare our approach to the relevant works presented in Section 4.2. Finally, Section 4.8 discusses future work and provides some closing remarks.

4.2 Survey of Relevant Research Work

The purpose of this section is to provide an in-depth survey of research work relevant to the proposed scientific contributions of this chapter. Our contributions are neither compared here nor are they situated with respect to the reviewed works. Such comparisons are instead provided in Section 4.7.

4.2.1 Specifying and Representing Abstract Syntax

Numerous modern DSM tools support meta-modelling based on the OMG's MOF or some variant of it, such as Ecore [EFb]. These include GME (Generic Modeling Environment) [LMB$^+$01], AToM^3, AToMPM and EMF (Eclipse Modelling Framework) [EFb]. In such tools, DSML abstract syntax is commonly described using UML Class Diagrams. Strictly textual DSM tools often favour the use of HUman readable Textual Notations (HUTNs) as the means to specify DSML abstract (and concrete) syntax. Example HUTNs include TXL [Cor06], Stratego/XT [BKVV08] and MetaDepth [dLG10]. Meta-modelling-based and grammar-based visual and textual abstract syntax representations each have their strengths, weaknesses and target audiences. For instance, it turns out that UML Class Diagrams very concisely, intuitively and most of all accessibly capture DSML abstract syntax and are thus currently the representation of choice. However, for DSM experts and modellers with stronger biases towards programming, HUTNs are often a more natural choice. A property of these representations is that neither of them is meant to carry any significant information about the modelled language's semantics. Whether this is a limitation or a strong point is the subject of debate, one that is taken up in this chapter.

4.2.2 Combining Modelling Languages

The problem of DSML combination has been the subject of recent research. On the one hand, DSML combination is desired for merging distinct views of a single system (e.g., a UML Class Diagram describing a system's structure with one or more Statecharts describing its behaviour). In [Val10], Vallecillo argues that it is unrealistic to model large and complex systems with a single instance model of a single DSML, and that instead it is preferable to model different facets of such systems with distinct instance models of distinct DSMLs. In essence, his argument is that as systems grow, the MPM principles move from best practices to bare necessities. Vallecillo provides a broad survey of existing model and meta-model combination approaches, and introduces the *viewpoint unification* technique. The reviewed and proposed approaches are mostly aimed at the merging of interrelated models and meta-models.

On the other hand, other authors have tackled the problem of DSML combination from an engineering perspective, studying how recurring patterns and structures in DSMLs can be turned into generic building blocks. In [ES06], Emerson and Sztipanovits target the reuse of parts or all of existing meta-models to address the repeated redefinition of popular meta-modelling patterns. Their technique, *template instantiation*, consists in presenting the meta-modeller with a library of *templates* that each capture some common abstract syntax pattern (e.g., composition hierarchies of composite and atomic objects, Statechart-style modelling). The meta-modeller can then instantiate these templates with his own domain-specific concepts yielding appropriately customized meta-model patterns in a timely, less error-prone and more standardized manner. A limitation of this approach is that it only strives to generate syntactic patterns within meta-models, with no focus on semantics.

White et al. also considered DSML combination from the perspective of reuse. In [WHT$^+$09], they describe how Feature Diagrams can be used to describe the interactions and dependencies between a collection of collaborating DSMLs. They argue that given appropriate code and model generation infrastructure, the selection of an arbitrary configuration of features (from one such feature model) could be used to automatically synthesize an appropriate meta-model. Their work borrows numerous ideas from SPLs and as such, is targeted at scenarios where there is a need to produce a finite number of *similar* meta-models.

4.2.3 Specifying and Representing Semantics

Although modern DSML syntax engineering does still lack in formal and standardized guidelines, the most ad hoc step of any DSM project remains the definition of a DSML's semantics.

The common approach – at least, for DSms of executable systems – is to encode the semantics of a language in a hand-crafted code generator[1] [Met, Saf07, KT08, WGM08]. Although certain tools, perhaps the most advanced of which is JetBrains' Meta Programming System (MPS) [Jet], propose means to organize and modularize the code generation process, it remains at a very low level of abstraction. Approaches that make use of model transformations, despite being at a more appropriate higher level of abstraction, often lack in formality and are equally ad hoc.

Certain authors have begun to suggest that both hand-coded and modelled DSm compilers rely too heavily on the ad hoc manual specification of semantic mappings. In [CSN05], the notion of *semantic anchoring* is explored in the context of formalizing and semi-automating DSML semantics specification. First, library-like reusable *semantic units* are defined to capture commonly occurring semantic patterns. Then, syntactic templates – much like those from [ES06] – are mapped onto appropriate *semantic units* by tool developers. The result is a language engineering environment where *partial* semantics can be generated, thereby alleviating the DSML designer burden and providing a standard solution to recurring problems. Related ideas are proposed by Pedro in [Ped09]. He introduces an approach in which DSML designers can instantiate parametrizable meta-models and transformations with their own manually defined meta-model and transformation fragments. This enables the reuse of the non-parametrizable portions of parametrizable meta-models and transformations. A key point of interest of these approaches, which may or may not be seen as a limitation depending on context, is that they only strive to generate *partial* semantics (and syntax).

4.3 DSML Building Blocks

A plethora of modelling formalisms exists. Some formalisms only differ in their concrete syntax, or have slightly different abstract syntax. However, formalisms are only truly different if their semantics differ. Seemingly small syntactic differences may have large associated semantic differences. For example, the simple introduction of inhibitor arcs in Place/Transition Petri Nets drastically changes their expressiveness from Turing-incomplete to Turing-complete. One way of classifying formalisms is based on systems theory. The nature of the state-space (continuous/discrete, finite/infinite) as well as the notion of time (continuous/discrete/partial order) allow for a structured distinction between different formalisms. The different notions of concurrency and determinacy allow for further classification. Despite the immense variety of (possible) modelling formalisms, a relatively small number of formalism *classes* can be distinguished. These are listed below. For each class, we give a representative, which we will henceforth refer to as *base formalisms*.

- Timed, reactive, deterministic descriptions where state is explicit (e.g., Statechart)

[1]Thus, the semantics of high-level models is defined in terms of the well-understood semantics of low-level programming languages.

- Descriptions of non-determinism, concurrency and synchronization where the notion of state is implicit (e.g., Petri Net)
- Descriptions focused on processes and/or interactions (e.g., Communicating Sequential Processes [Hoa85])
- Discrete- or continuous-time dataflow descriptions of algebraic computation (e.g., Causal Block Diagram, such as The Mathworks' Simulink [The])
- Layout (constraint) descriptions (e.g., *GUI*, see meta-model in Figure 4.1)
- Descriptions of computations such as found in procedural programming languages (e.g., *ActionCode*, see meta-model in Figure 4.2)

It has been our experience that the majority of DSMLs constructed by ourselves and others is (conceptually) based on a combination of (fragments of) one or more of the listed class representatives. The notion that most – or all – DSMLs are the result of the combination of a limited set of such *building blocks* is the rationale behind the approach proposed in this chapter. This notion very nicely ties into the approach to DSML semantics specification presented in Chapter 2. Indeed, if DSMLs are compositions of base formalisms, it stands to reason that decomposition means might allow DSms to be projected onto instance models in relevant base formalisms that could then be merged back together into coherent artifacts. Such disassembly and merging tasks are indeed reminiscent of the concern isolation and projection and intermediate representation merging model transformations that produced Google Android applications from PhoneApp models.

We make no claim towards the exhaustiveness of the proposed list of formalism classes. However, the meaning of the "exhaustiveness" property in this context is unclear. If it indicates that the set of proposed classes suffices to reconstruct any conceivable DSML, then any set of classes that includes a Turing-complete formalism is immediately exhaustive. This is a trivial and empty claim: the fact that any problem can in theory be mapped onto code or enhanced Petri Net models does little to elevate the abstraction of DSML design and engineering. Indeed, the exhaustiveness of a Turing-complete set of base formalisms in the context of providing abstractions for common types of system architectures (e.g., discrete-time deterministic) is of limited relevance. A more interesting property is the set's ability to provide *sufficiently high-level* abstractions, which we term its *usefulness*. This is of course an informal and subjective property which is a function of DSML designer tastes and of existing/required system architectures. We can of course not claim that the listed formalism classes are universally (as) *useful* for all DSML designers, or that its usefulness will not decay over time. These concerns are more thoroughly addressed in Section 4.6. Fortunately, they are of little consequence with respect to the upcoming description of how an arbitrary set of base formalisms may be leveraged and combined to form DSMLs.

Finally, note that the provided formalism classes mostly focus on capturing system architectures and behavioural paradigms. Indeed, our usefulness criteria are biased towards

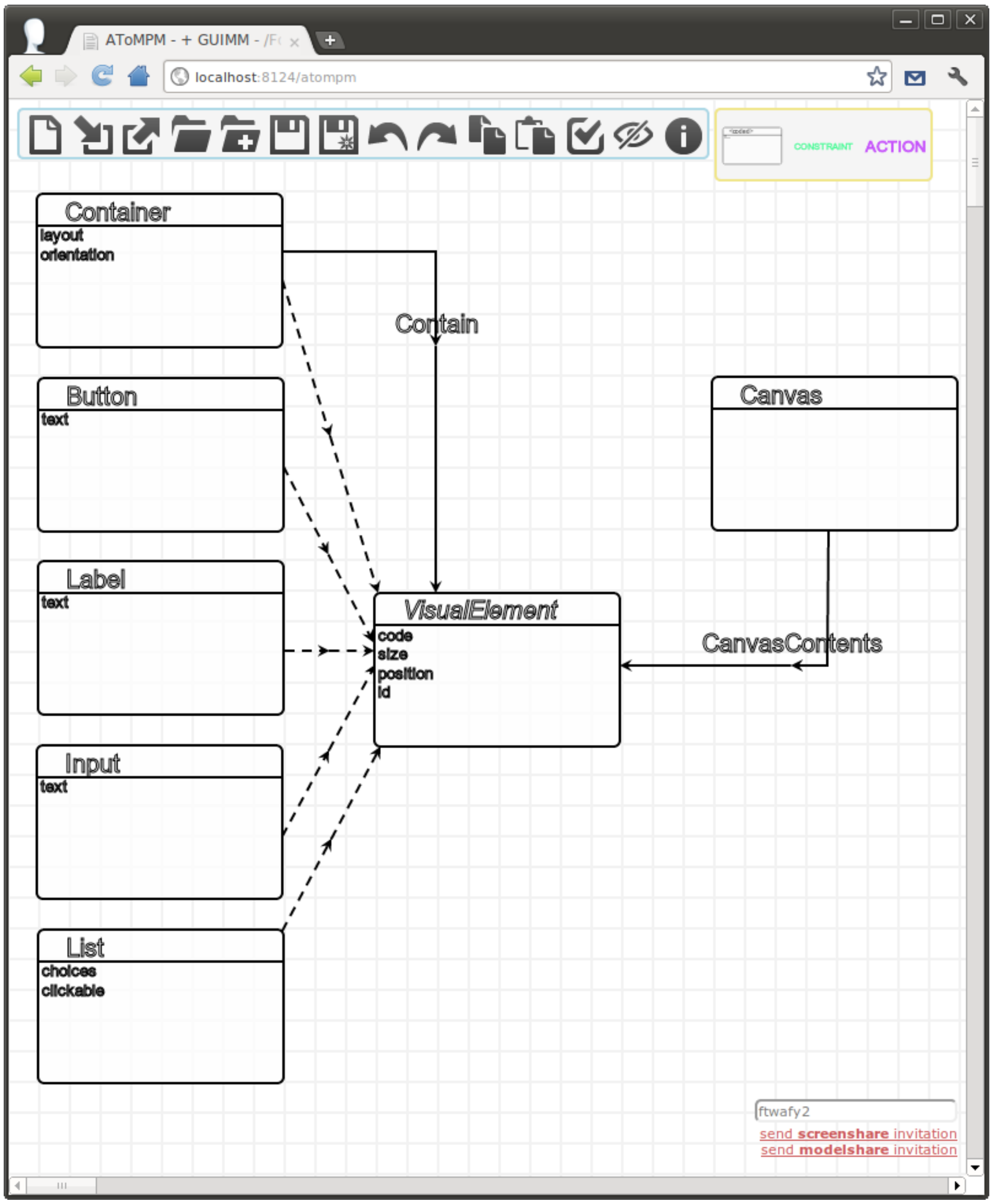


Figure 4.1: The *GUI* meta-model.

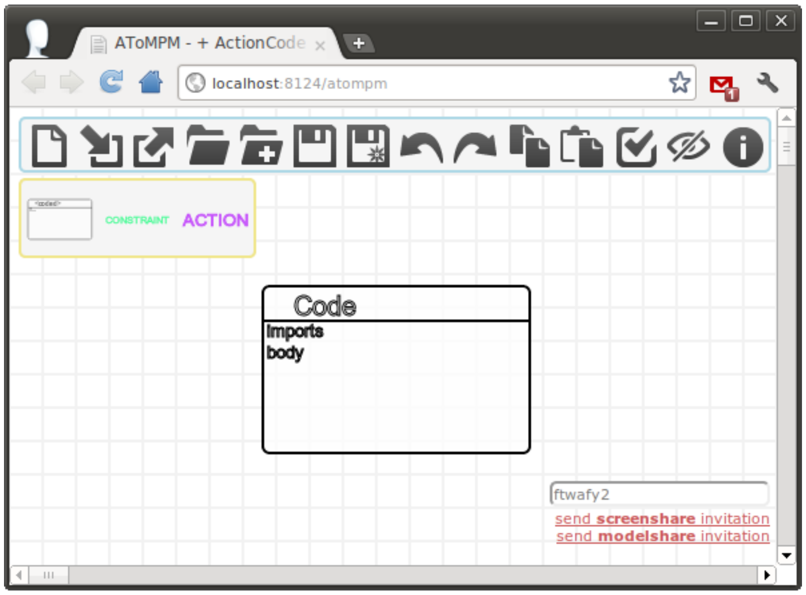


Figure 4.2: The *ActionCode* meta-model.

DSms of executable systems for which precise semantics are required. Nevertheless, our approach is not restricted to any spectrum of DSMLs or target platforms. A different (and more *useful*) set of base formalisms – possibly a superset of the proposed set – could include abstractions on top of which DSMLs for describing structure and variability, for instance, could be built.

4.4 A Unified Representation of DSML Syntax and Semantics

Despite its advantages, the artifact synthesis technique presented in Chapter 2 still has two unfortunate limitations in the context of DSML engineering. First, it requires a considerable amount of non-trivial craftsmanship since the semantic mapping of DSms to artifacts still needs to be specified manually. DSML designers must manually identify which portions of their languages to project onto which lower-level formalisms, how to carry out the said projections, and how to weave their results back into coherent artifacts. Second, it does not address the specification of DSML syntax, which it assumes has already been defined separately (e.g., as a UML Class Diagram). Section 4.3 introduced a collection of low-level formalisms that we argued could be combined to form more complex new DSMLs. In this section, we propose an approach to perform the said combination such that resulting models of DSMLs carry both syntactic *and* semantic information. From these, automated synthesis of traditional syntax and semantics specifications can be performed, thereby eliminating the aforementioned limitations of our previous work while retaining all of its benefits. Keeping in mind the said benefits, a desirable technique for the construction of new DSMLs in terms of base formalisms should enable the elegant automation of the deconstruction and projection of DSms onto base formalism instance models, while maximally shielding the DSML designer from any resulting accidental complexities.

4.4.1 Specifying DSML Syntax and Semantics

We propose that the specification of DSML syntax and semantics be reduced to the instantiation of a set of syntactically and semantically rich templates, which we term *Semantic Templates* (STs), with arbitrary domain-specific concepts. These templates are much more than placeholders for isolated syntactic and/or semantic structures as was the case for the template-based approaches presented in Section 4.2. Instead, they are *interfaces* to the base formalisms that are combined to form new DSMLs. Each base formalism exposes a set of STs that encode the unambiguous mapping of arbitrary domain-specific entities onto (syntactic and semantic) entities from the given base formalism. Each ST encompasses the knowledge of which domain-specific concepts should be mapped onto which base formalism, as well as how to carry out the mapping. In practice, the specification of a base formalism's STs forms a meta-model, to which models formed by instantiated templates conform. This is illustrated in Figure 4.3. Figures 4.3a and 4.3c show ST meta-models for the Statechart and Petri Net base formalisms, respectively. Figures 4.3b and 4.3d show sample models where each available ST from the relevant ST meta-model is instantiated. The meaning of the depicted templates and of the `EventList` construct present in both ST meta-models will be detailed shortly.

ST meta-models define one class for each valid ST. The attributes of such classes represent the parametrizable fields of their associated ST. For instance, the *IsAStateTemplate* ST takes two parameters, which its associated class, `IsAStateTemplate`, captures via the `__1__` and `__2__` attributes. Note that little care is put into giving ST parameters meaningful names. Their intended meaning is instead conveyed in the concrete syntax of instantiated STs, as shown in Figure 4.3.

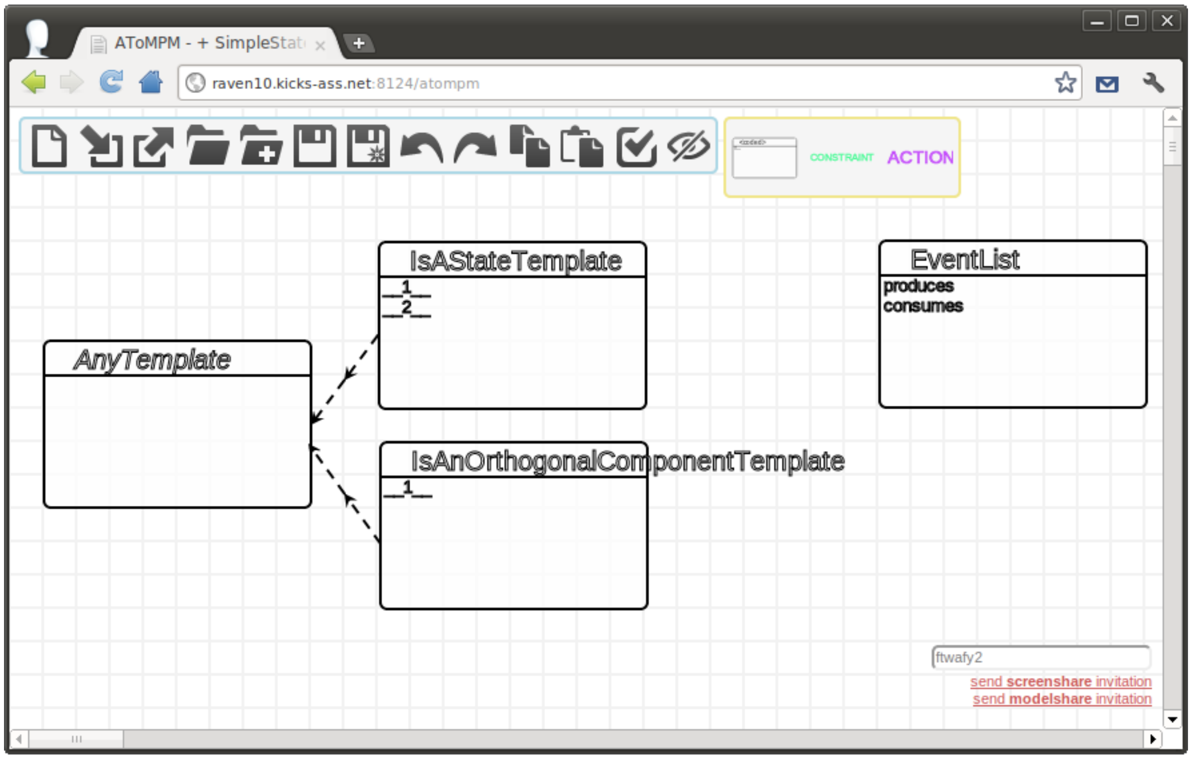
ST meta-models (like any other) may include arbitrary constraints. Indeed, much of the power and productivity increases attributed to DSM are the results of restricting modellers to the creation of valid models. The same rationale naturally extends to DSML design and engineering when DSMLs themselves are viewed as DSms. In this context, meta-model constraints are meant to ensure that given STs are not instantiated too often or too little, that they are not instantiated more than once with identical parameters, that they are not given invalid parameters (e.g., a negative number where a positive one is expected), but most importantly, to prevent logical inconsistencies that would otherwise result from conflicting template instantiations. For example, given a template of the Petri Net base formalism that encodes the mapping of one domain-specific concept onto a `Place` and of another domain-specific concept onto a `Transition`, a conflict would occur if the template were to be instantiated twice with reversed parameters. The first instance would imply that domain-specific concepts `X` and `Y` are to be respectively mapped onto a `Place` and a

`Transition`, while the second instance would imply the opposite. Thus, meta-model constraints are essential to ensure the composability of a base formalism's STs.

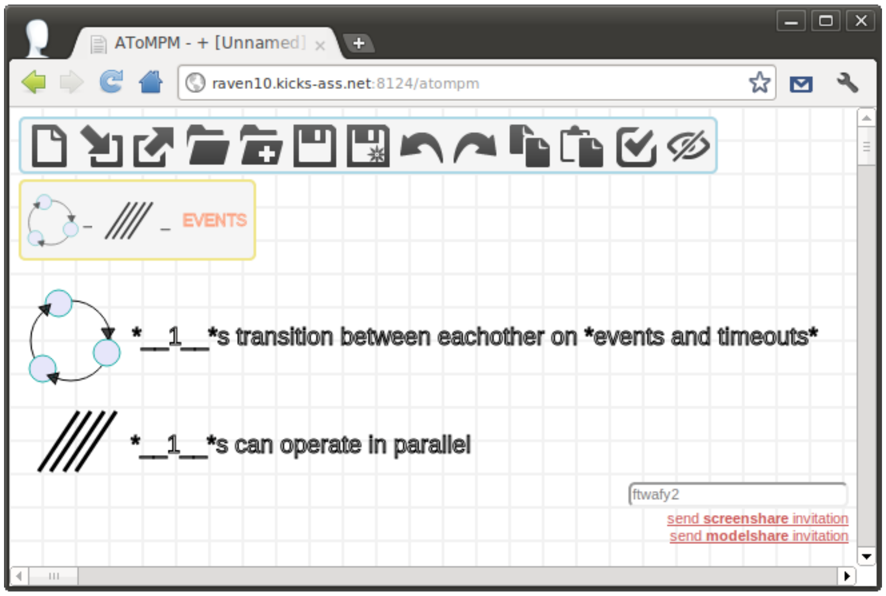
Our approach follows the underlying principle of DSM, namely, the *leveraging of expertise using maximally constrained notations*. In DSM, domain experts are called upon to create models of problems in their domains, DSM experts are called upon to create models of DSML syntax and semantics, and programming experts may be called upon to implement APIs (or other target platform artifacts) amenable to code generation. Each actor is meant to perform only the tasks that fall within the range of his expertise, using only familiar concepts and constructs. Although DSM experts should be aware of the best practices that govern DSML design, the aforementioned task allocation also forces them to have a non-negligible mastery of an arbitrarily large number of domains and target platforms for artifact generation (e.g., the Petri Net formalism, the Python programming language, the Google Android API). This is unrealistic and contrary to the very principles of DSM. Moreover, the relatively small number of DSM experts (who are repeatedly forced to learn new problem and solution domains) quickly emerges as a bottleneck for the adoption of DSM. Our approach aims at alleviating the burden placed on DSM experts by appropriately redistributing their tasks. First, there exist experts on the specification of how arbitrary concepts should be mapped onto a formalism F (which may be a modelling formalism or a programming language). Indeed, these are none other than experts of formalism F, and, in our approach, they – rather than (possibly otherwise F-oblivious) DSM experts – specify F's STs, and provide the knowledge of how domain concepts that instantiate a ST of base formalism F should be mapped onto syntactic and semantic concepts of F. In addition, this knowledge is only provided *once*, when the base formalism and its STs are first introduced. This contrasts with current, code-generator- or model transformation-centric approaches, where numerous DSM experts with varying levels of expertise with formalism F each re-implement (probably strikingly similar) mappings onto F for (possibly) many of their projects. Second, given sufficiently *useful* sets of base formalisms, the specification of DSMLs in terms of their STs becomes more accessible and could often conceivably be accomplished by problem domain experts. Indeed, once such activities as the definition of UML Class Diagram meta-models and of coded or modelled DSm-to-artifact compilers have been abstracted away (i.e., within a ST-enabled tool), common DSML design and engineering tasks no longer require DSM experts. Thus, just as the DSM community would often have programming experts move behind the scenes, away from problem domain experts, by elevating solution design and implementation above notions of code, we would often have DSM experts take a similar step back, by elevating DSML design and implementation above lower-level DSM and meta-modelling notions.

4.4.2 The Semantics of Semantic Templates

STs were introduced as interfaces exposed by base formalisms that carry sufficient information to automate the projection of DSms onto base formalism instances. In practice, the meaning of a ST of base formalism F is specified (by an expert of F) in two separate steps.

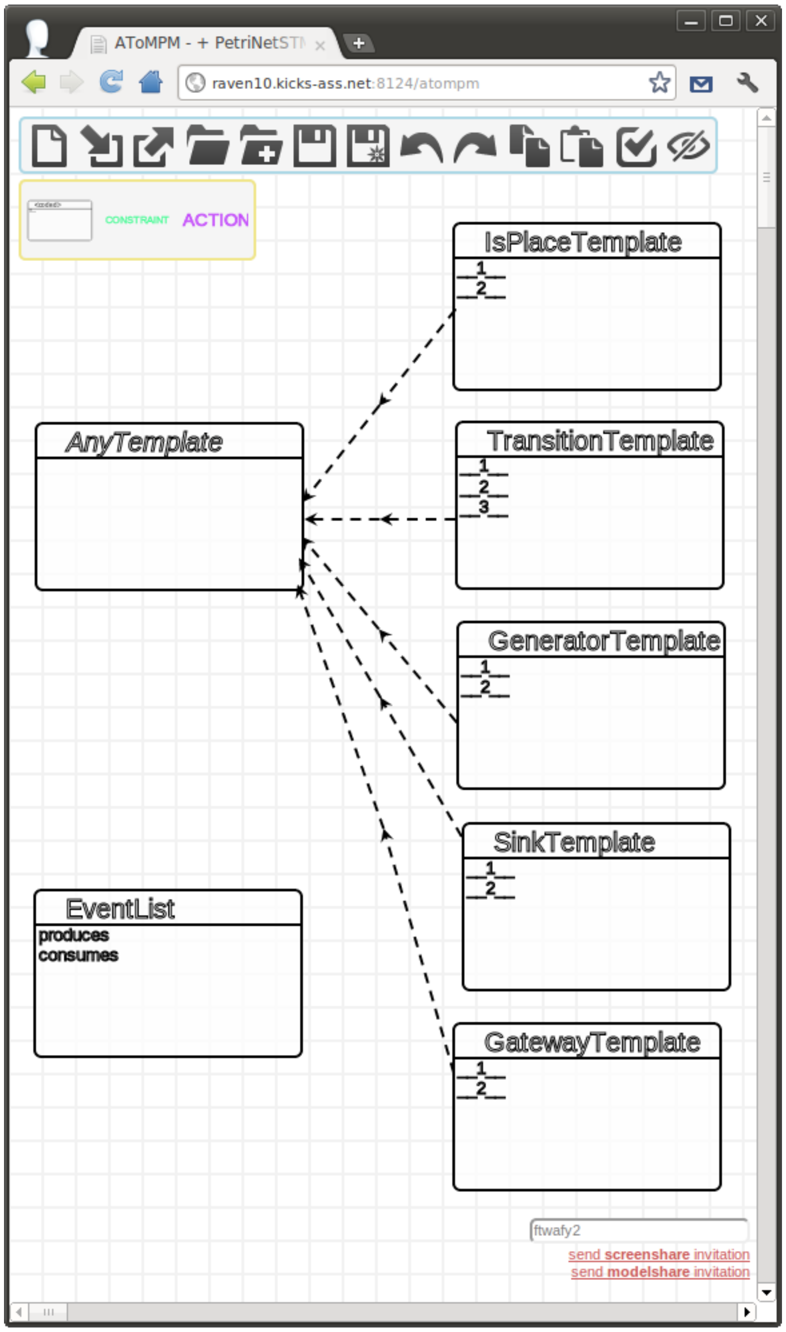

ATоMPM - + SimpleStat
raven10.kicks-ass.net:8124/atompm
CONSTRAINT
ACTION
IsAStateTemplate
__1__
__2__
EventList
produces
consumes
AnyTemplate
IsAnOrthogonalComponentTemplate
__1__
ftwafy2
send screenshare invitation
send modelshare invitation

(a) The ST meta-model of the Statechart formalism.


ATоMPM - + [Unnamed]
raven10.kicks-ass.net:8124/atompm
EVENTS
*__1__*s transition between eachother on *events and timeouts*
*__1__*s can operate in parallel
ftwafy2
send screenshare invitation
send modelshare invitation

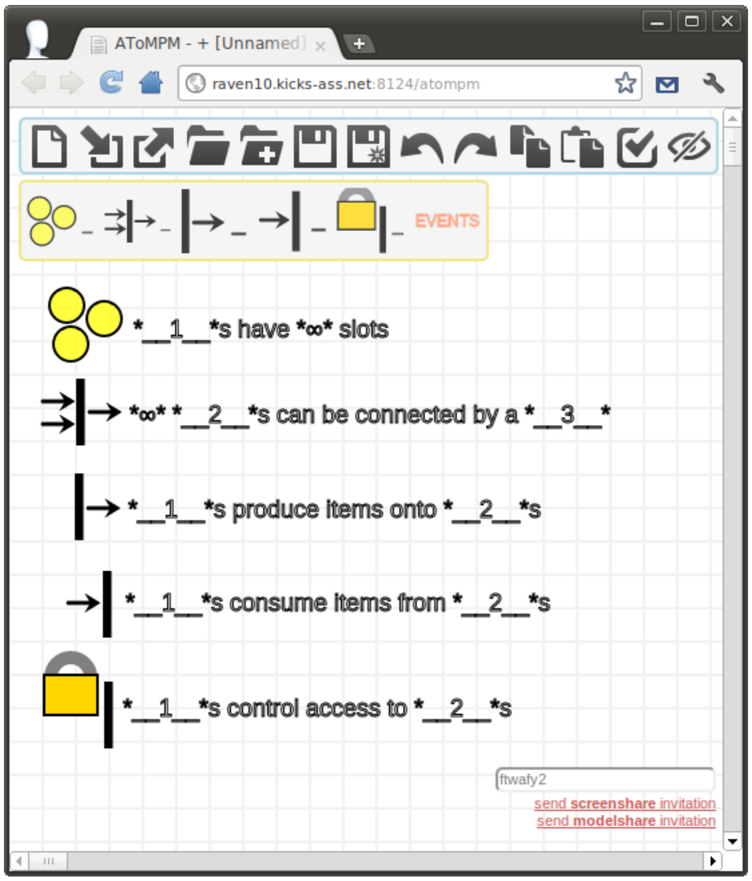
(b) A model that instantiates both templates from the Statechart ST meta-model.

Figure 4.3: ST meta-models and example instance models *(continued)*.



(c) The ST meta-model of the Petri Net formalism.

Figure 4.3: ST meta-models and example instance models *(continued)*.


AToMPM - + [Unnamed]
raven10.kicks-ass.net:8124/atompm
EVENTS
*__1__*s have *∞* slots
∞ *__2__*s can be connected by a *__3__*
*__1__*s produce items onto *__2__*s
*__1__*s consume items from *__2__*s
*__1__*s control access to *__2__*s
ftwafy2
send screenshare invitation
send modelshare invitation

(d) A model that instantiates both templates from the Petri Net ST meta-model.

Figure 4.3: ST meta-models and example instance models.

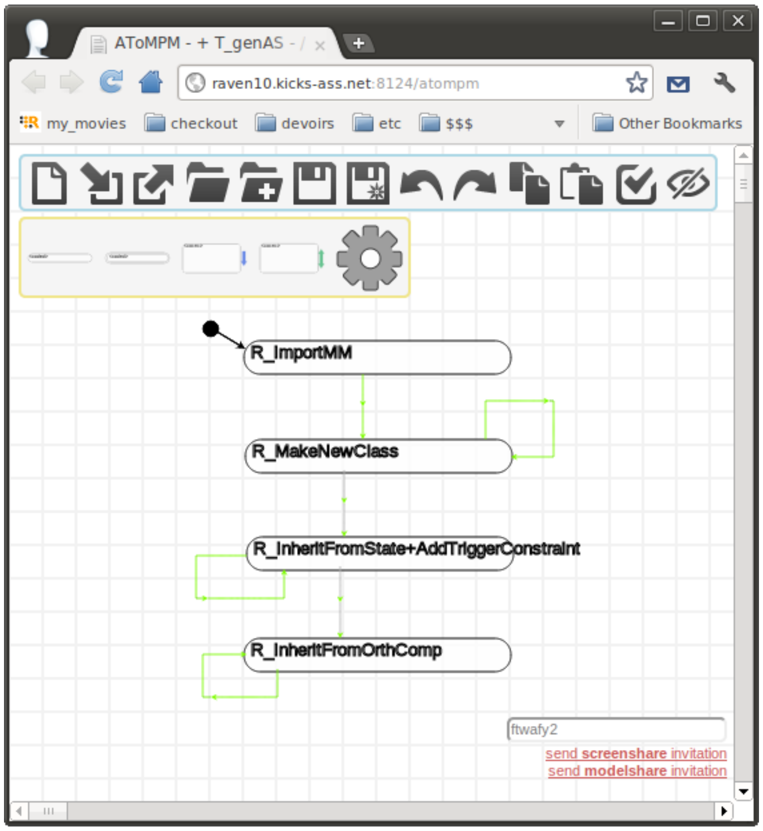
First, the syntactic implications of the ST (i.e., the impact(s) of its instantiation on the to-be formalism's abstract syntax) are defined. Then, its semantic implications (i.e., the impact(s) of its instantiation on the to-be formalism's semantics). In our approach, meta-models are still represented via UML Class Diagrams augmented with constraints, and semantics are still represented via model transformations. We do not propose to eliminate these proven and effective representations, but rather to automatically synthesize them from models of DSMLs that hide the infamous divide between syntax and semantics within a higher-level and unified representation.

A ST's syntactic implications consist of a collection of constraints and a partial UML Class Diagram. The set of constraints and the full UML Class Diagram that result from the instantiation of a collection of STs (from possibly many base formalisms) will form the meta-model of the new DSML. Thus, each ST contributes to a fraction of the final DSML's abstract syntax. To avoid unnecessary duplication and complexity (in particular during the subsequent specification of the semantic implications), we strongly favour the use of inheritance relationships to relate classes corresponding to domain-specific concepts with classes within the UML Class Diagrams of base formalisms. In turn, this enables attributes, associations and constraints to be “passed down” from the base formalisms, or, in other words, for portions of their syntax to be inherited. In practice, the syntactic implications of a base formalism’s STs are encoded in its *genAS* (for generate abstract syntax) model transformation. Its rules match STs in their LHSs and produce the aforementioned partial meta-models in their RHSs. Figures 4.4a and 4.4c show the *genAS* transformations of the Statechart and Petri Net base formalisms, respectively. Both transformations operate very similarly. First, they import the entire meta-model of the relevant base formalism (via *R_ImportMM* rules). Then, a new class is created for each domain-specific concept that instantiates a ST (via *R_MakeNewClass* rules). Next, appropriate inheritance relationships are created between domain-specific concept classes and base formalism classes (via *R_InheritFrom** rules). Finally, additional attributes, constraints and multiplicities are injected as needed. Figure 4.4b depicts the *R_InheritFromOrthComp* rule, which matches a Statechart ST, a base formalism class and a domain-specific concept class, and creates an inheritance relationship between the latter two. A NAC is used to elegantly avoid re-applying the rule on previously processed input. See Figures 4.17 and 4.24 in Section 4.5 for examples of full meta-models generated from STs.

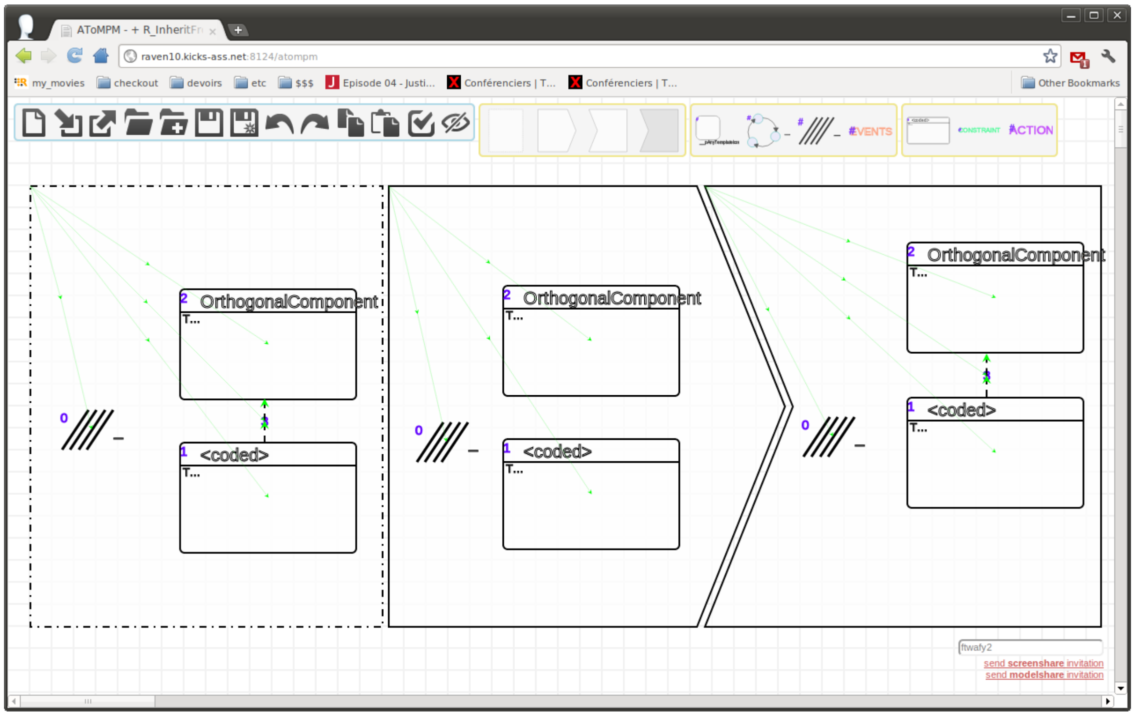
The semantic implications of a base formalism’s STs are fully encapsulated within its *mapTo* model transformation, which projects DSms onto it. Its rules match patterns in DSms and produces semantically equivalent patterns in base formalism instances. Thanks to *subtype matching* and to the inheritance approach described above, these rules are generic and need not specify domain-specific constructs to match them. Figures 4.5a and 4.5b show the *mapTo* transformations of the Statechart and Petri Net base formalisms, respectively. These transformations are also conceptually quite similar. They create appropriately parametrized base formalism entity instances (e.g., Statechart `State`s and Petri Net `Place`s) for each subtyping domain-specific entity (via *R_*Subtype2** rules). Then, they connect the new entities in a manner that reflects connections, if any, in the DSm. In the case of the Petri Net *mapTo* transformation, the last three rules are used to realize the notion of capacity (i.e., a `Place`’s maximum number of tokens) which is not a native part of the Petri Net meta-model. This is further explored later in this section. Figure 4.5c depicts the *R_PlaceSubtype2Place* rule. Not shown are the subtype matching flag and condition code which cause the `Place` labelled “0” to only match domain-specific entities that subtype the `Place` type. Thus, for each such subtype, this rule yields an appropriately parametrized `Place`. Note that generic links are produced to connect DSm and base formalism entities such that the numerous aforementioned benefits of traceability might be reaped later on. Once a *mapTo* transformation has run its course, any transformation defined on the targeted base formalism (e.g., a Statechart to Java transformation) may be used to manipulate the resulting base formalism instance model. The means by which individual STs are transformed into traditional DSML syntax and semantics have been explored. Concrete examples are depicted and explained in the following paragraphs.

Figure 4.6 details a ST of the Petri Net base formalism. Its general and instantiated forms are shown in Figures 4.6a and 4.6b respectively, while Figures 4.6c and 4.6d shows the latter's syntactic impacts. The first of them is that a class is created for the `X` domain-specific concept and made to subclass the `Place` class from the imported Petri Net meta-model. Side-effects of this are that instances of `X` in the DSm may now be connected to `Transition` instances (and vice versa) and possess an `nbTokens` attribute. The second of the ST's parameters dictates the maximal number of tokens `X` may carry. When set to infinity, it has no syntactic impact. However, when set to a fixed value k, a `capacity` attribute and two constraints are added to `X`. The first, *metaEnforeCapacity* ensures that the `capacity` attribute of instances of `X` in the DSm never exceeds k. The second, *enforceCapacity*, ensures that the `nbTokens` attribute of instances of `X` in the DSm never exceeds their `capacity` attribute. These are thus constraints that guide modellers towards creating only valid models. As for the semantic impacts, as mentioned earlier, these are captured by the Petri Net *mapTo* transformation. As `X` subclasses `Place`, its instances will be matched by the transformation's rules (such as the aforementioned *R_PlaceSubtype2Place* rule) that will extract all relevant information from them to create appropriate `Place`s. However, this extraction process is not as trivial as simply copying the `nbTokens` attribute from each `X` into its corresponding new `Place`. The `capacity` attribute must also be accounted for if the resulting Petri Net instance model is to be truly semantically equivalent (as far as those concerns captured by Petri Net models go) to the DSm. Thus, the final three rules of the Petri Net *mapTo* transformation translate the concept of capacity into Petri Net terms. Essentially, via the addition of an appropriately connected *capacity `Place`*, any transition that wishes to add a token to the `Place` representing an `X` will be disabled if that `Place` already contains a number of tokens equal to its capacity. This is illustrated in Figure 4.27 from Section 4.5. Note that if the capacity were to be left infinite, the final three rules of the Petri Net *mapTo* transformation would be non-applicable. Thus, despite the fact that *mapTo* transformations are already defined and that the purely semantic impacts of STs (and of their parametrizations) may seem negligible, they do indeed exist and should not be underestimated.

Figure 4.7 details another ST of the Petri Net base formalism. Again, general and instantiated forms are shown. As for syntactic impacts, classes are created for the `X` and `Z` domain-specific concepts and respectively made to subclass the `Place` and `Transition` classes from the imported Petri Net meta-model. Note that the *R_MakeNewClass* rules from Figure 4.4 ensure that exactly one class is created for each domain-specific concept. All other abstract syntax generation rules merely match and augment them (e.g., with attributes or inheritance relationships). Side-effects of the aforementioned syntactic impacts are that `X` instances in the DSm may now be connected to instances of `Transition` *and* of `Z` (and vice versa). This specific template further implies that `Z` instances should not have any inputs. This constraint is reflected in the `cardinalities` attribute of the `Z` concept's class which sets the maximum number of incoming connections from `Place` subtypes to zero, as shown in Figure 4.7d. As above, semantic impacts are captured by the Petri Net *mapTo* transfor-

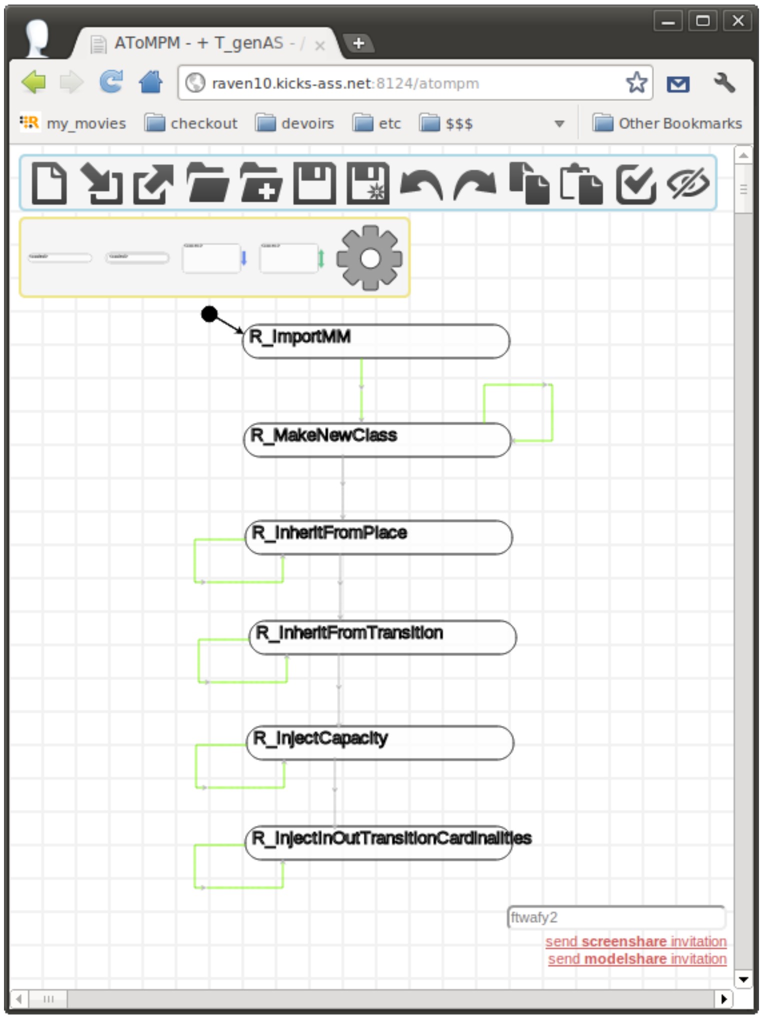

AToMPM - + T_genAS
raven10.kicks-ass.net:8124/atompm
my_movies
checkout
devoirs
etc
$$$
Other Bookmarks
R_ImportMM
R_MakeNewClass
R_InheritFromState+AddTriggerConstraint
R_InheritFromOrthComp
ftwafy2
send screenshare invitation
send modelshare invitation

(a) The *genAS* model transformation of the Statechart formalism.


AToMPM - + R_InheritFr
raven10.kicks-ass.net:8124/atompm
OrthogonalComponent
<coded>
ftwafy2
send screenshare invitation
send modelshare invitation

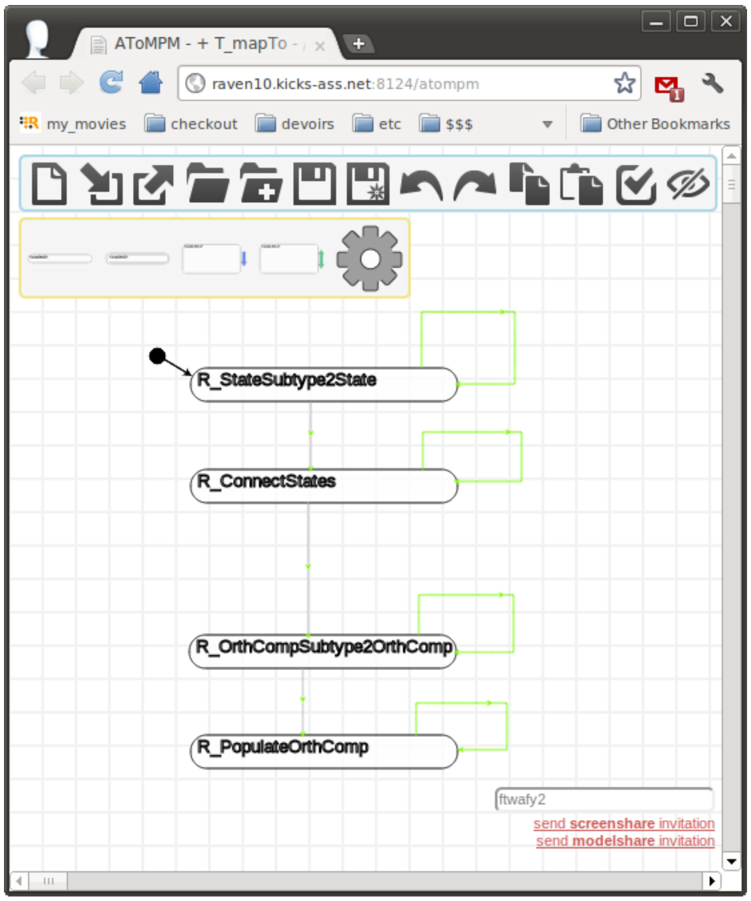
(b) The *R_InheritFromOrthComp* rule from (a).

Figure 4.4: Abstract syntax generating model transformations and rule *(continued)*.



(c) The *genAS* model transformation of the Petri Net formalism.

Figure 4.4: Abstract syntax generating model transformations and rule.

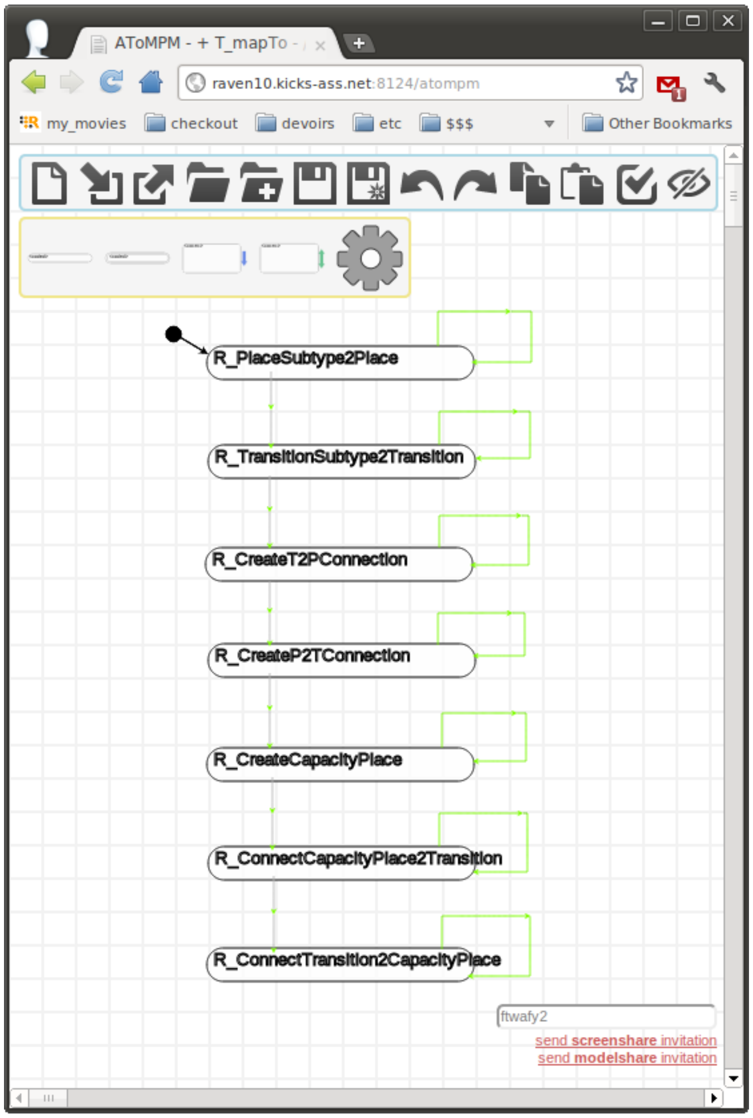


(a) The *mapTo* model transformation of the Statechart formalism.

Figure 4.5: Base formalism instance generating model transformations and rule *(continued)*.

mation.

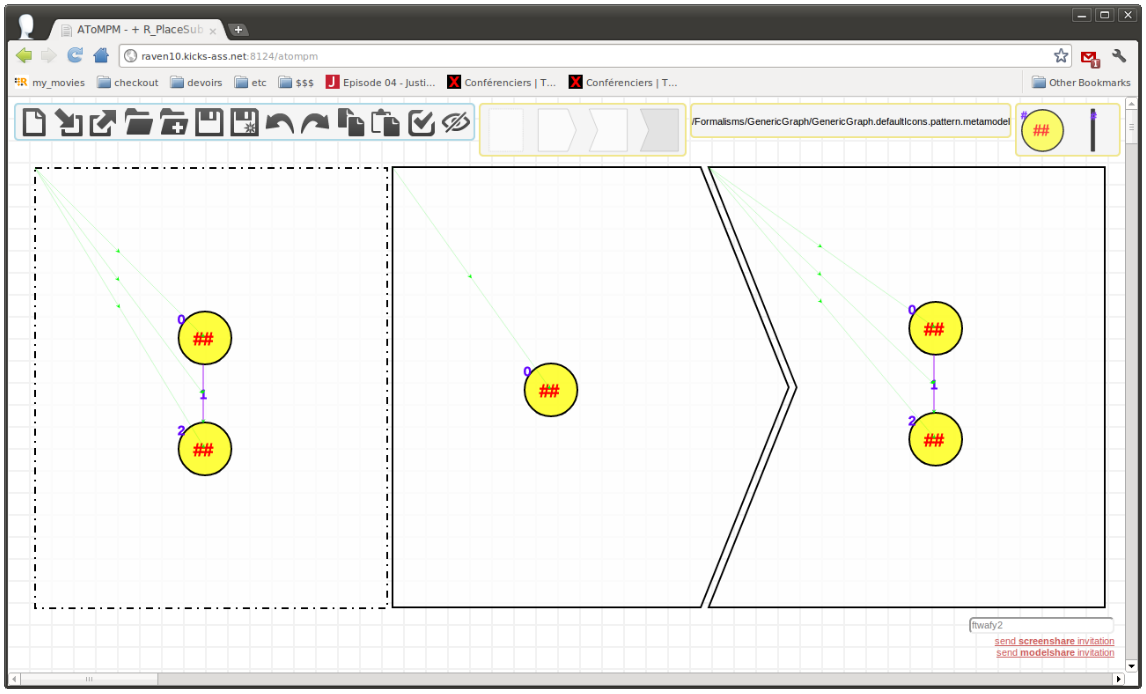
Figure 4.8 details a ST of the Statechart base formalism. The first of its syntactic impacts is that a class is created for the `A` domain-specific concept and made to subclass the `State` class from the imported Statechart meta-model. A side-effect of this is that `A` instances in the DSm may now be connected to each other via `Transition` links. The second of the ST's parameters allows DSML designers to hide part of the Statechart formalism's features from the domain-specific modeller. In this case, inter-state transition triggers are restricted to events only. This may be desirable to avoid accidental complexity in a DSML where delayed transitions are not needed. This restriction is compiled into the generated abstract syntax by the *R_InheritFromState+AddTriggerConstraint* rule from Figure 4.4a. It is realized as the *restrictEventTypes* constraint which ensures DSms are only valid if none of their transition triggers are delays. Analogously to previous cases, semantic impacts are captured by the Statechart *mapTo* transformation, which maps networks of connected `A` instances onto semantically equivalent (as far as those concerns captured by Statechart models go) Statechart


AToMPM - + T_mapTo -
raven10.kicks-ass.net:8124/atompm
my_movies
checkout
devoirs
etc
$$$
Other Bookmarks
ftwafy2
send screenshare invitation
send modelshare invitation

R_PlaceSubtype2Place
R_TransitionSubtype2Transition
R_CreateT2PConnection
R_CreateP2TConnection
R_CreateCapacityPlace
R_ConnectCapacityPlace2Transition
R_ConnectTransition2CapacityPlace

(b) The *mapTo* model transformation of the Petri Net formalism.

Figure 4.5: Base formalism instance generating model transformations and rule *(continued)*.


AToMPM - + R_PlaceSub
raven10.kicks-ass.net:8124/atompm
/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
##
0
1
2
send screenshare invitation
send modelshare invitation

(c) The *R_PlaceSubtype2Place* rule from (b).

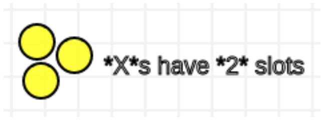
Figure 4.5: Base formalism instance generating model transformations and rule.

instance models.

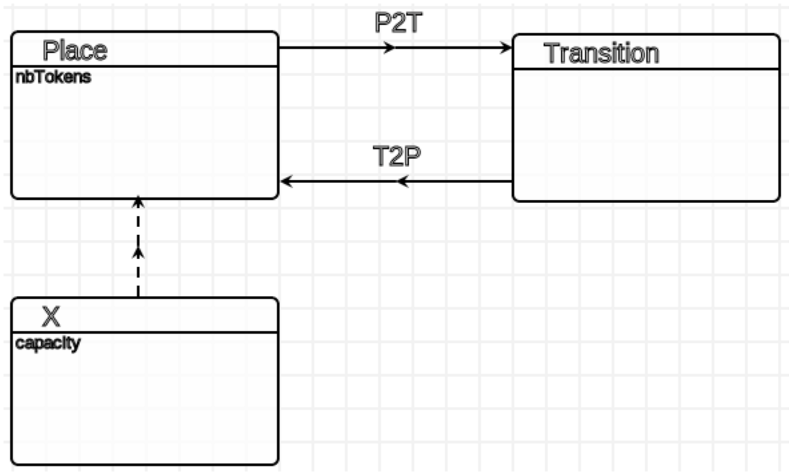
Figure 4.9 details a ST of the *ActionCode* base formalism. The first of its syntactic impacts is that a class is created for the `C` domain-specific concept and is made to subclass the `Code` class from the imported *ActionCode* meta-model. This ST is extremely simple. Given the need and proper tool support, a second parameter could be added to restrict the language of code specified in the `body` and `imports` attributes of instances of `C`. In turn, this could lead to meta-model constraints (e.g., arbitrarily complex verifications of code validity) or serve to provide appropriate syntax highlighting in the DSM tool's attribute editor. Semantic impacts are captured by the ActionCode *mapTo* transformation, which trivially creates a `Code` entity with identical attributes for each `C`.

Figure 4.10 details a *generic* ST. This ST has no associated base formalism. Its sole purpose is to organize and abbreviate DSML specifications, much like inheritance relationships in the code world often effectively reduce complexity and duplication. Its trivial syntactic impact is shown in Figure 4.10c, and is defined by ST tool developers (as opposed to base formalism experts). In practice, the second of its parameters is often used in a ST from some other base formalism. For instance, if the ST from Figure 4.8 were present, an instance of this generic ST might be "Ps and Qs are types of As". This would imply the creation of inheritance relationships between the classes of concepts `P` and `Q` and that of `A`, *and* that the Statechart *mapTo* transformation would now also match and transform instances of `P` and _s have $\underline{\infty|k}$ slots

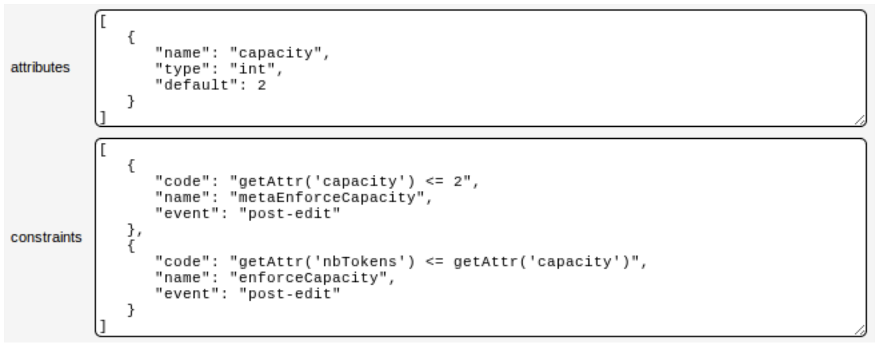
(a)



(b)



(c)


attributes

```
[
    {
        "name": "capacity",
        "type": "int",
        "default": 2
    }
]
```

constraints

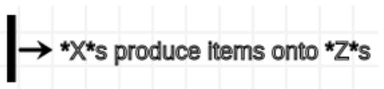
```
[
    {
        "code": "getAttr('capacity') <= 2",
        "name": "metaEnforceCapacity",
        "event": "post-edit"
    },
    {
        "code": "getAttr('nbTokens') <= getAttr('capacity')",
        "name": "enforceCapacity",
        "event": "post-edit"
    }
]
```

(d)

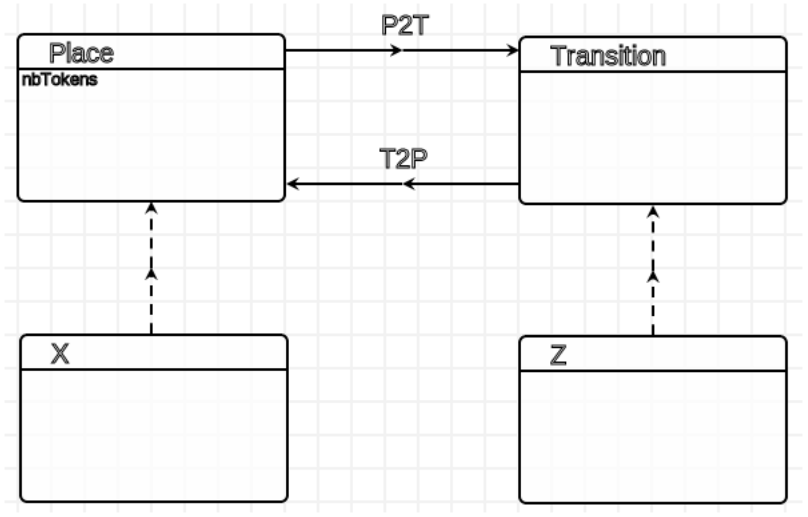
Figure 4.6: (a) A ST of the Petri Net base formalism, (b) an example instance, and (c-d) its abstract syntax implications.

_s produce items onto _s

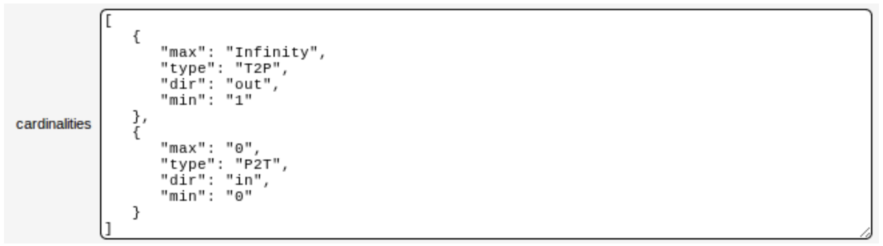
(a)



(b)



(c)



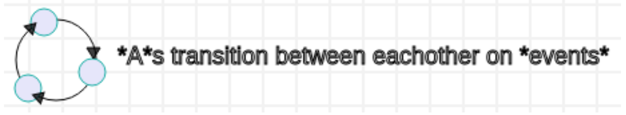
```
[
   {
      "max": "Infinity",
      "type": "T2P",
      "dir": "out",
      "min": "1"
   },
   {
      "max": "0",
      "type": "P2T",
      "dir": "in",
      "min": "0"
   }
]
```

(d)

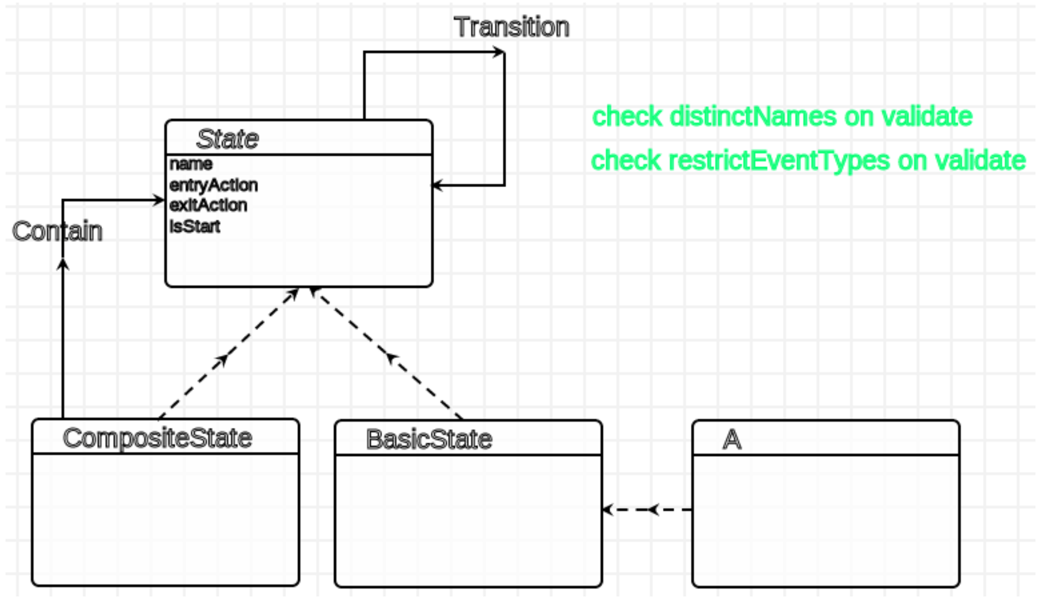
Figure 4.7: (a) Another ST of the Petri Net base formalism, (b) an example instance, and (c-d) its abstract syntax implications.

_s transition between eachother on *events|timeouts|events and timeouts*

(a)



(b)



(c)

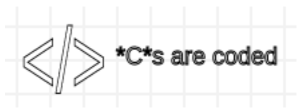
Figure 4.8: (a) A ST of the Statechart base formalism, (b) an example instance, and (c) its abstract syntax implications.

`Q` into `State`s. Similar cases are encountered in the examples of Section 4.5.

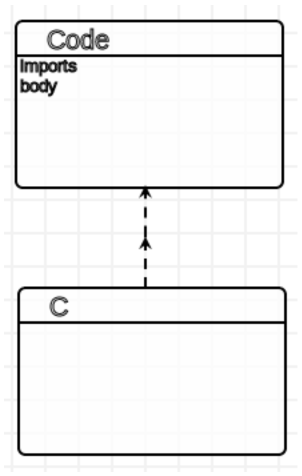
Five STs have been reviewed. Some are easily transformed into partial abstract syntax patterns. Others require more elaborate non one-to-one syntactic and/or semantic mappings as they reduce base formalism expressiveness within DSMLs (e.g., restrictions on `Transition` trigger types) and/or abstract and bundle non-trivial base formalism structures into higher-level representations (e.g., `Place` capacity structures as attributes). Regardless, all STs are given meaning in the same manner: via base formalism (and generic) *genAS* and *mapTo* transformations. Together, these enable a collection of instantiated STs to define complex DSML abstract syntax *and* the means by which domain-specific constructs should be mapped onto semantically equivalent (as far as those concerns captured by the target base formalism go) base formalism instance models in a single, concise and high-level representation. Note that in the presence of STs from several base formalisms, a DSML's semantics must include several `mapTo` transformations. Thus, in practice, when the top-level model transfor-

_s are coded

(a)



(b)



(c)

Figure 4.9: (a) A ST of the *ActionCode* base formalism, (b) an example instance, and (c) its abstract syntax implications.

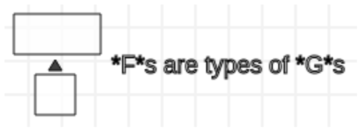
mation that captures a new DSML's semantics is synthesized from a collection of STs, each ST contributes its base formalism's *mapTo* transformation (if it hasn't already been) to the top-level transformation. This is by far the most crucial and explicit semantic impact of STs.

Finally, note that in the depictions of syntactic impacts, classes from imported base formalism meta-models are not abstract. Thus, revisiting the ST from Figure 4.6, DSms may contain entities of `X`, of `Place`, and of `Transition`. On the one hand, it may be confusing for concepts not used to instantiate any STs (e.g., `Transition`) to be accessible parts of the synthesized DSML. On the other hand, this enables much more concise DSML specifications. In practice, DSML designers can very easily include and exclude constructs from DSMLs without having to manipulate the generated abstract syntax (e.g., to make undesired concepts's classes abstract) by defining concrete syntax only for those concepts that are wanted in DSms[2]. For instance, if concrete syntax representation were only defined for `X`s and `Transitions`, `Place`s could not be instantiated within DSms.

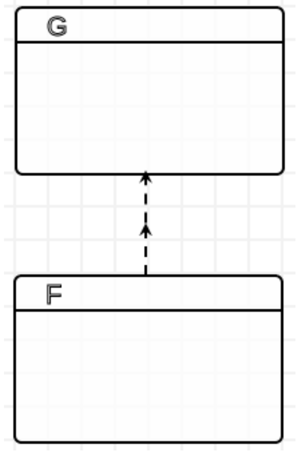
[2]Note that our approach entirely ignores concrete syntax. It is left to the DSML designer or domain experts to specify proper icons for domain-specific concepts.

_s are types of _s

(a)



(b)



(c)

Figure 4.10: (a) A *generic* ST, (b) an example instance, and (c) its abstract syntax implications.

4.4.3 Merging Base Formalism Instance Models

We've argued that STs indeed encode sufficient information to fully specify the projection of DSms onto instance models m_i of base formalisms F_i. However, recalling the artifact synthesis process introduced in Chapter 2, these m_i must be merged back together to produce cohesive target platform artifacts that properly reflect DSm semantics. The need for such merging originates from the facts that certain domain-specific entities may have synthesized counterparts in more than one m_i, and that it may be necessary for events and/or data from one m_i to be translated and propagated to another. This scenario was discussed in Chapter 2 where mobile phone `Screen` constructs at the DSm level were mapped onto instances of both the Statechart and AndroidScreens formalisms. In that example, it was necessary for entry into a Statechart `State` corresponding to a certain `Screen` to trigger the display of the AndroidScreens `AndroidScreen` corresponding to that same `Screen`. The required event propagation was implemented by the DSML designer, who first had to decide how to instrument the behaviour artifact such that it made appropriate use of other artifacts (e.g., layout), and then had to manually enhance artifact generating model transformation rules to facilitate and implement this weaving. This was arguably one of the most complex steps in the proposed semantics specification approach. Luckily, its difficult and menial tasks can be fully automated by making a small adjustment to the artifact synthesis process, and by making use of STs to describe event translation. First, rather than literally merging m_i models together, target platform artifacts a_i are now generated from each of them. Additionally, a *main* artifact (i.e., the artifact to execute) is responsible for initializing and launching each a_i. Throughout their execution, these a_i each emit and consume a number of events for which DSML designers must specify translations (e.g., event e from a_i should cause event f to be sent to a_j) via generic STs.

Before we can truly examine how events are mapped across artifacts, we must first establish the spectrum of existing events. To this end, we further develop the idea of "STs as interfaces to base formalisms" into a new direction. Along with a set of STs, base formalisms are also characterized by a collection of input and output *events*[3]. These are chosen and defined by the base formalism expert such that meaningful interactions are made possible. Table 4.1 shows a tentative list of events produced and consumed by each base formalism[4]. The introduced events are defined at the base formalism level. However, for the DSML designer to make (proper) use of them, they need to be made available at the DSML level. In our ST approach, events are treated in a manner analogous to class properties and methods: DSMLs *inherit* the events of base formalisms they are built on. More specifically, domain-specific constructs inherit the events of base formalism constructs they subtype (in the generated DSML meta-model). Thus, referring to Table 4.1 and recalling the running example from Chapter 2, a mobile phone screen construct that appropriately instantiates STs from both the Statechart and GUI base formalisms would inherit all the input and output events of both formalisms. It may then make sense to translate `enteredState` events produced by the Statechart artifact to `draw` events and redirect them to the GUI artifact.

The mapping of events between different base formalisms only makes sense in cases where a domain-specific concept inherits events from more than one base formalism. Identifying such concepts in a ST DSML definition trivially reduces to identifying domain-specific concepts that are used to instantiate STs of more than one base formalism, and is thus amenable to validation by meta-model constraints. A single additional generic ST suffices to capture how synthesized artifacts should emit and respond to events. Figure 4.11 shows both this new ST and its implications. It has three parameters: the "multi-base formalism" domain-specific concept, the event to translate and its translation. In Figure 4.11b, these are respectively `X`, `E` and `F`. Note that more complex translations are supported and are demonstrated in Section 4.5. This new ST's semantic impacts are snippets of event management code that describe the appropriate matching, translation and propagation of events, as shown in Figure 4.11c. First, the event is matched. One must determine if the base formalism entity instance that produced the `E` event has a corresponding associated domain-specific `X` entity (Figure 4.11c, line 1). Then, the base formalism artifact sensitive to the `F` event is retrieved

[3]The impacts of this event-based scheme on our approach's applicability are discussed in Section 4.6

[4]The `Place.block` and `Place.unblock` events are syntactic sugar around the `Place.setTokenCount:c` event. The former sets a `Place`'s number of tokens to zero while the latter restores a `Place`'s number of tokens to whatever it was before it was "blocked", if ever.

(Figure 4.11c, lines 2-3) and a new `F` event message targeted at its entity corresponding to `X` is sent to it (Figure 4.11c, lines 4-5). Note that two cross-formalism correspondences must be resolved. These are resolved by navigating a compiled form of the web of traceability links left behind by artifact synthesis. The collection of event management snippets generated from instances of the event translation ST are bundled together with transmission protocol code to form the *event manager*.

The event manager sits at the center of all target platform artifacts (i.e., of all the a_is). It translates the events they produce into meaningful events in the formalism of their intended recipient (as specified by instances of the event translation ST) and propagates them to that recipient. In practice, artifacts synthesized from base formalisms might conform to different meta-models (i.e., they might be targeted at different platforms). To ensure the proper transmission of events between them, our approach requires each artifact to implement a specific platform-independent transmission protocol. This protocol includes initialization, termination, and event transmission and reception facilities. Transmission facilities consist of means to communicate output events to the event manager in a standardized and meaningful manner. Reception facilities consist of means to receive events and translate them into appropriate commands (e.g., a function call to alter a Petri Net's marking as a result of the reception of the `Place.setTokenCount:c` event). The described transmission protocol enables and ensures the proper emission, delivery and handling of events between artifacts. Hence, specifying the mapping of events from one base formalism to another, an operation previously achieved through the complex and manually specified merging of various models, now merely involves the straightforward and intuitive instantiations of a single generic ST.

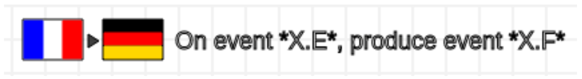
Formalism	**Produces**	**Consumes**
Statechart	`State.enteredState:name`, `State.exitedState:name`	`*MODEL*.handleEvent:ev`
PetriNet	`Transition.transitionFired`, `Place.tokenCountChanged:c`	`Place.block`, `Place.unblock`, `Place.setTokenCount:c`
Causal Block Diagrams	`Block.outputChanged:value`	`Block.setInput:value`, `Block.setOutput:value`
GUI	`Canvas.buttonClicked:id`, `Canvas.listClicked:id,choice`, `Canvas.textChanged:id,text`	`Canvas.draw`
ActionCode	`Code.codeProduced:ret`	`Code.runCode`

Table 4.1: A tentative list of events produced and consumed by each base formalism.

The need for inter-artifact communication was motivated by the necessity for event *and* data exchange. More generally, data produced by any artifact may be required by itself or another later during execution. This was the case in the conference registration example

On event _, produce event _

(a)



(b)

```
if( CrossFormalismIdMap.getAncestorTypes(ev.id).contains("X") && ev.msg.equals("E") )
  artifacts.
    get( event2mm.get("F") ).
      handleEvent(
        new ArtifactEvent( CrossFormalismIdMap.getSibling(ev.id, event2mm.get("F")), "F"));
```

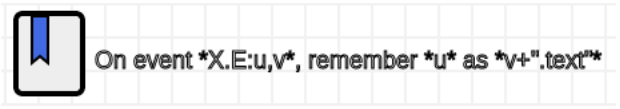
(c)

Figure 4.11: (a) The generic event mapping ST, (b) an example instance, and (c) its semantic implications.

from Chapter 2, where content entered in a text field was required to populate the body of a subsequent text message. To meet this need in a consistent manner, one last generic ST is introduced. Figure 4.12 shows both this new ST and its implications. It has three parameters: the fully specified event, a neutral expression that evaluates to the value to remember, and a neutral expression that evaluates to the key to store that value under. In Figure 4.12b, these are respectively `X.E:u,v`, `u` and `v+"text"`, which means that on event `X.E:u,v`, `u` will be stored in a data cache under `v+"text"`. This is realized by the data management code snippet shown in Figure 4.12c, which is bundled with event management code snippets, if any, during the event manager's construction.

On event _, remember _ as _

(a)



(b)

```
if( CrossFormalismIdMap.getAncestorTypes(ev.id).contains("X") && ev.msg.equals("E") ) {
   String u = ev.args[0];
   String v = ev.args[1];
   rememberValueAs(v+".text",u);
}
```

(c)

Figure 4.12: (a) The generic data storage ST, (b) an example instance, and (c) its semantic implications.

We have presented the key elements of our technique for DSML design and engineering. The entire process, which is mostly reduced to no more than the instantiation of a number of base formalism and optionally event translation and helper STs, is summarized below.

1. ST meta-models are loaded.
2. ST instances are created and instantiated with domain-specific concept names.
3. The *generate DSML* functionality (provided by ST-enabled tools such as AToMPM) is launched, prompting the DSML designer for
   - the new DSML's name, and
   - the desired target platform(s) for artifact synthesis

   and then automatically creating
   - a new meta-model M for the DSML,
   - the "main" artifact (responsible for initializing and launching future artifacts),
   - the event manager, and
   - a semantics model transformation T that sequences appropriate *mapTo* transformations, then base formalism-to-target platform transformations, and finally a web of traceability links-to-target platform transformation.
4. M is loaded and DSms are created.
5. Target platform artifacts are produced from DSms via T.

In the next section, we provide concrete examples that illustrate the creation of two DSMLs via STs, and show their synthesized abstract syntax and semantics.

4.5 Case Studies

The case studies in this section illustrate how different DSML design tasks can be performed using our ST approach. In both cases, we first describe the problem domain and requirements of the associated DSML. We then show the STs that form its specification, along with the generated abstract syntax and semantics, and example instance models of the new DSMLs.

4.5.1 A DSML for Mobile Device Applications

In Chapter 2, we introduced PhoneApp, a DSML for modelling mobile device applications. Its *manually defined* meta-model is shown again in Figure 4.13. With a series of *manually defined* multi-rule model transformations, PhoneApp instance models were translated to increasingly lower-level formalisms until a Google Android application was synthesized. We

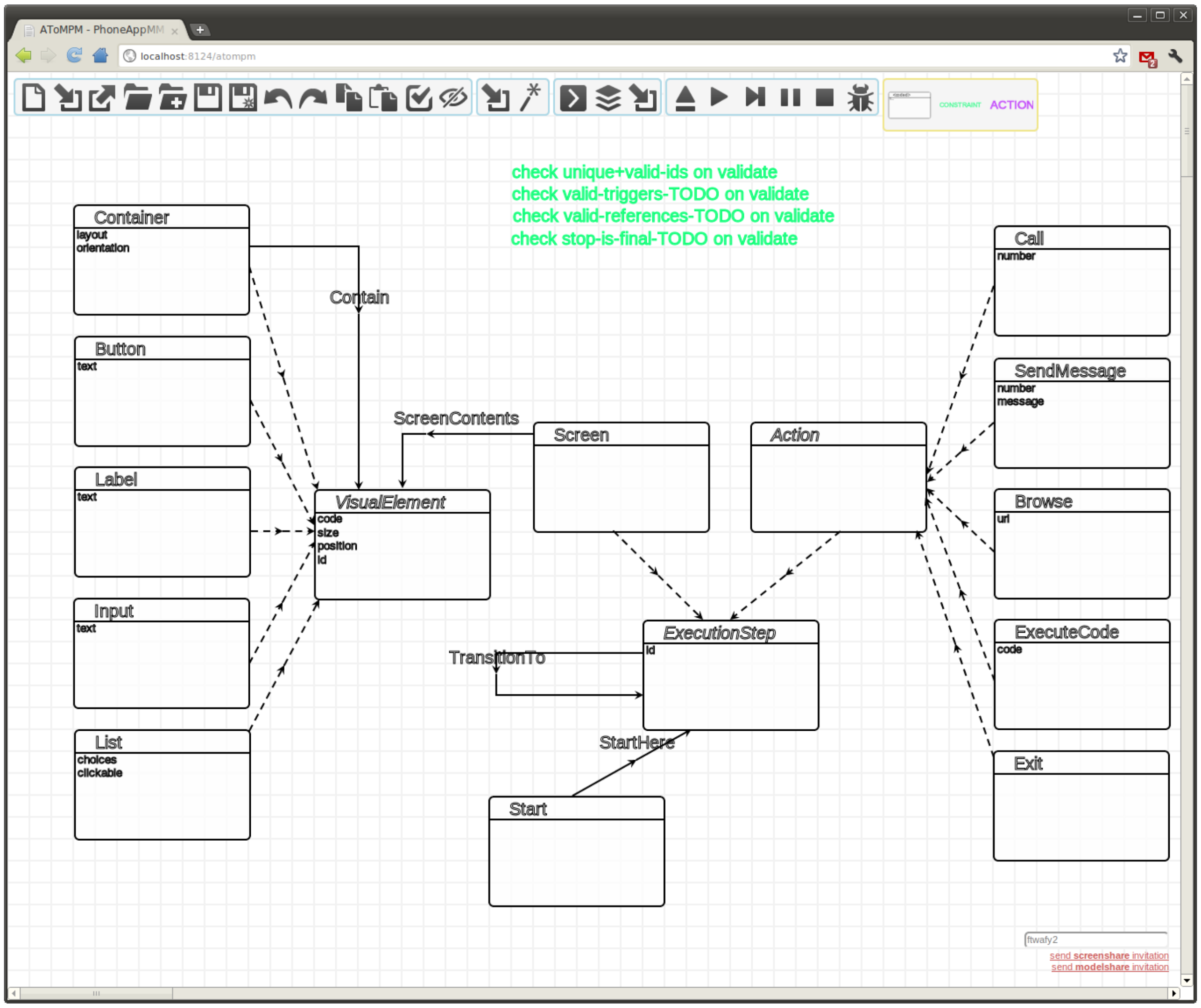


Figure 4.13: The *manually defined* PhoneApp meta-model.

now describe how the PhoneApp DSML can be redefined in terms of STs such that syntax and semantics analogous to those we had previously defined manually can be fully automatically generated. Before proceeding, recall the three typical concerns of a mobile device application. The first is its visual interface, which is essentially described by the placement of widgets on the various screens the user must interact with. The second is its behaviour, which is described by the timed and user-prompted transitions between the aforementioned screens. The third encompasses features and functions specific to mobile applications.

The *PhoneApp*$^{2.0}$ DSML should be built on top of base formalisms that (ideally) disjointly capture domain concerns. A possible choice is to combine the GUI, Statechart and ActionCode base formalisms. Unfortunately, this choice forces the domain-specific modeller to manually input the Google Android code that enacts desired mobile device features. This, of course, contradicts the DSM and MPM principles that we have repeatedly defended. A better approach – which more closely reflects the version of the DSML presented in Chapter 2 – is to model mobile device features at a high level of abstraction, and generate their associated Google Android code, thus sparing the modeller the need to learn and use the Google Android API. To this end, we introduce a new *non-native, impure* base formalism: *AndroidAPI*. Its meta-model is shown in Figure 4.14. Abstractions are provided for a small number of mobile device- and Android-specific functionalities[5]. Superficially, i.e., to the DSML designer, *native* and *non-native* base formalisms are indistinguishable. They both expose a set of STs and input/output events, and can be combined arbitrarily to form new DSMLs. Non-native base formalisms are our approach's built-in extensibility means. They enable any DSML designer to promote his DSMLs to base formalism status pending the specification of its STs (and their implications) and events, which he is after all the most apt to specify. The number, variety, and level of abstraction of STs available in a ST-enabled DSM framework is thus unbounded. As for *non-native, impure* base formalisms, they are non-native base formalisms for which partial or complete syntax and semantics are defined manually, outside the scope of the ST-enabled DSM framework. Non-native, impure base formalisms are our approach's means to provide *escape semantics*. They allow DSML designers to promote DSMLs for which syntax and/or semantics already exist to base formalism status without having to rebuild the DSMLs using existing STs. Section 4.6 discusses in detail the criteria that enable the differentiation of native and non-native base formalisms, and the scenarios where it is preferable for DSML designers to choose an impure approach to base formalism specification. Note that in this particular case, the AndroidAPI formalism could just as well have been a *non-native, pure* formalism built on top of `ActionCode`. Our choice here was merely motivated by the intention to introduce non-native, impure base formalisms.

Figure 4.15 introduces the STs of the AndroidAPI base formalism. It shows example instances and their collective syntactic implications. These are analogous to those of previously introduced STs, and are realized via the AndroidAPI base formalism's *genAS* transformation, while semantic implications are captured by its *mapTo* transformation[6]. Lastly, the AndroidAPI base formalism is sensitive to the `APIFeature.launch` event, and produces the `APIFeature.finished` event.

Now that its constituting base formalisms have been clearly defined, a specification of the PhoneApp$^{2.0}$ DSML can be created. One such specification is shown in Figure 4.16, and again in condensed, textual form in Table 4.2. This representation clearly requires a small fraction of the time, space and know-how necessary for the manual specification of the PhoneApp DSML's abstract syntax and semantics (using a UML Class diagram augmented with constraints and model transformations) as was detailed in Chapter 2. Nevertheless, it carries sufficient information to generate them both. Figure 4.17 shows the synthesized

[5]This meta-model is of course incomplete with respect to the full Google Android API but it is sufficient for the purposes of this demonstration.

[6]The AndroidAPI's *genAS* and *mapTo* model transformations are very similar to those we've already explored and, as such, they are not detailed here. Instead, the full process of promoting a DSML to base formalism status is explored in a more complex setting in the second case study.

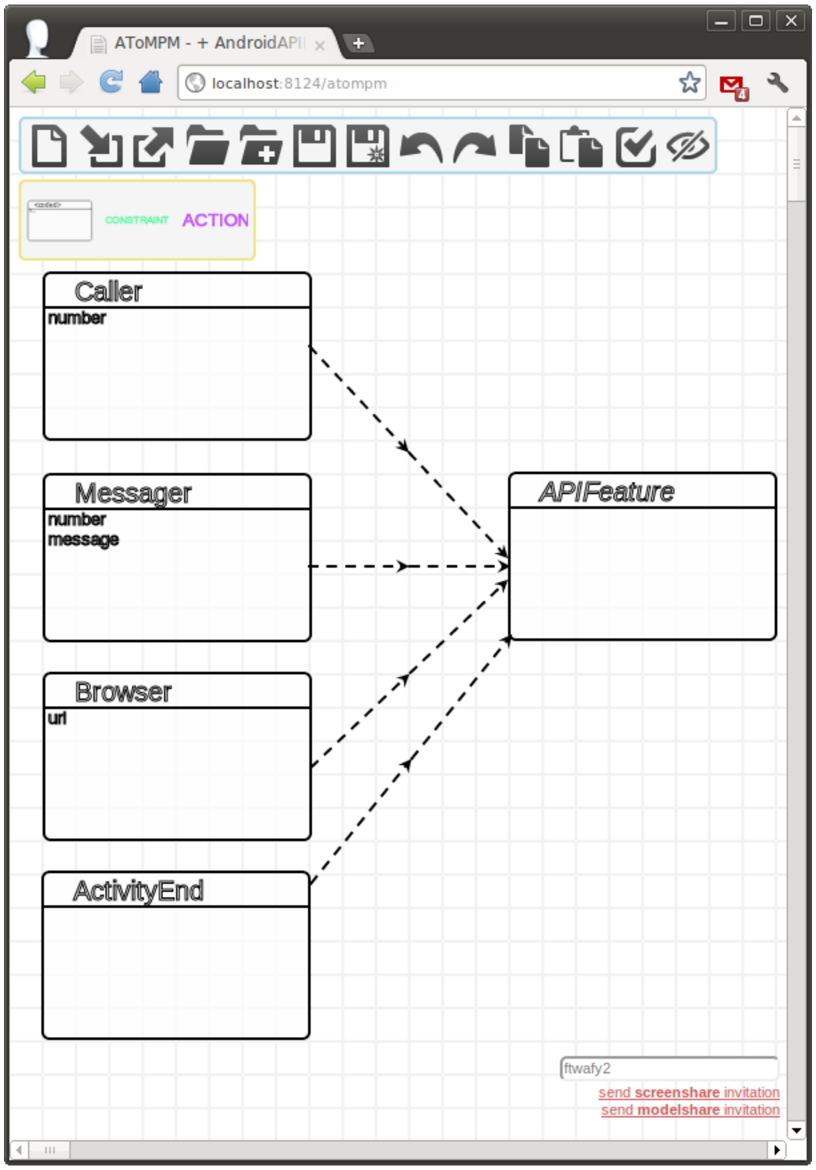
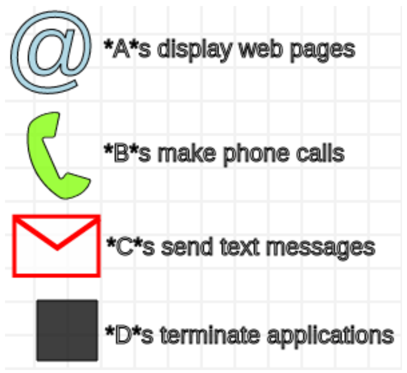
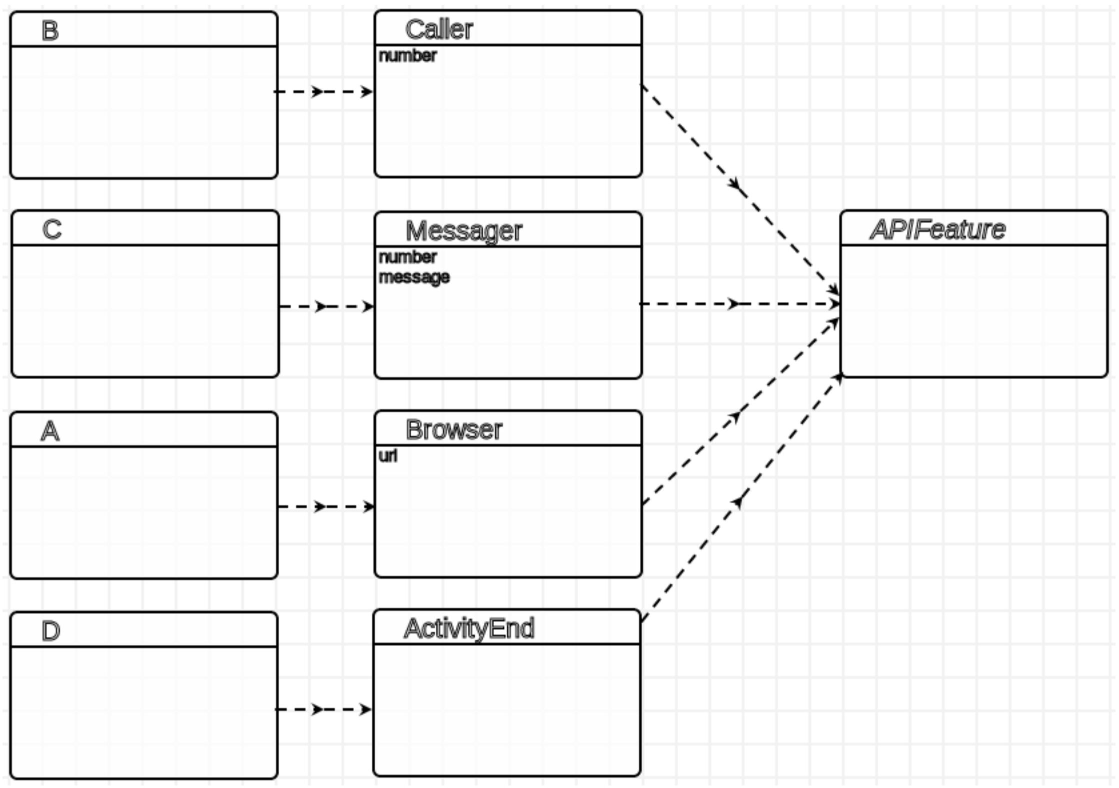


Figure 4.14: The *AndroidAPI* meta-model.



(a)



(b)

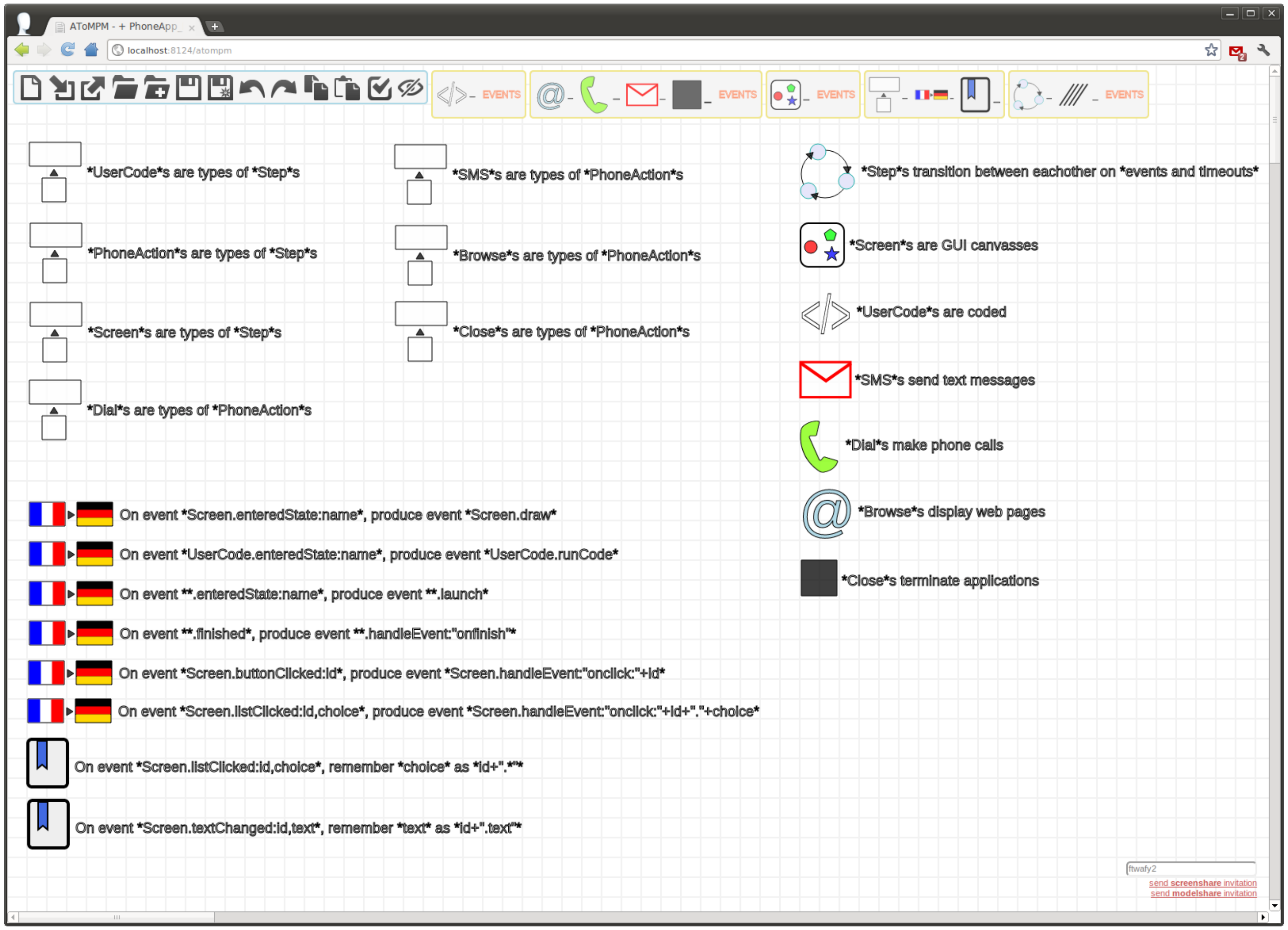
Figure 4.15: (a) Example instances of each of the AndroidAPI base formalism's STs, and (b) their collective abstract syntax implications.

meta-model resulting from the syntactic implications of the STs that form the PhoneApp$^{2.0}$ specification. It combines classes from four imported base formalism meta-models with those created for the domain-specific concepts. These are identified with "*"s appended to their names for clarity. The resemblance of the generated meta-model to the manually defined PhoneApp meta-model from Figure 4.13 is undeniable, and it is indeed virtually equivalent. As for the semantic implications of the STs that form the PhoneApp$^{2.0}$ specification, they are mostly captured by the synthesized model transformation shown in Figure 4.18. Four *mapTo* transformations (one for each of the four base formalisms) produce base formalism instance models from DSms. These are then each compiled into FileSystem models by four more transformations, and the resulting full FileSystem model is output to disk by another. Lastly, the web of traceability links left behind by all previous transformations is compiled and output to disk. The remaining pieces of the generated semantics are the "main" artifact and the event manager. The former initializes the four target platform artifacts, while the latter captures the eight event translation and data storage STs.

Formalism	**Template**
< *generic* >	`UserCode`s, `PhoneAction`s and `Screen`s are types of `Step`s.
< *generic* >	`Dial`s, `SMS`s, `Browse`s and `Close`s are types of `PhoneAction`s.
Statechart	`Step`s transition between eachother on `events and timeouts`.
GUI	`Screen`s are GUI canvasses.
ActionCode	`UserCode`s are coded.
AndroidAPI	`SMS`s send text messages.
AndroidAPI	`Dial`s make phone calls.
AndroidAPI	`Browse`s display Web pages.
AndroidAPI	`Close`s terminate applications.
< *generic* >	On event `Screen.enteredState:name`, produce event `Screen.draw`.
< *generic* >	On event `UserCode.enteredState:name`, produce event `UserCode.runCode`.
< *generic* >	On event `*.enteredState:name`, produce event `*.launch`.
< *generic* >	On event `*.finished`, produce event `*.handleEvent:"onfinish"`.
< *generic* >	On event `Screen.buttonClicked:id`, produce event `Screen.handleEvent:"onclick:"+id`.
< *generic* >	On event `Screen.listClicked:id,choice`, produce event `Screen.handleEvent:"onclick:"+id+"."+choice`.
< *generic* >	On event `Screen.listClicked:id,choice`, remember `choice` as `id+".*"`.
< *generic* >	On event `Screen.textChanged:id,text`, remember `text` as `id+".text"`.

Table 4.2: The PhoneApp$^{2.0}$ DSML, as a set of STs (in condensed, textual form).

Given the abstract syntax and semantics generated from the ST representation of the PhoneApp$^{2.0}$ DSML, a domain-specific modeller has all the necessary tools at hand to create



*UserCode*s are types of *Step*s

*PhoneAction*s are types of *Step*s

*Screen*s are types of *Step*s

*Dial*s are types of *PhoneAction*s

*SMS*s are types of *PhoneAction*s

*Browse*s are types of *PhoneAction*s

*Close*s are types of *PhoneAction*s

*Step*s transition between eachother on *events and timeouts*

*Screen*s are GUI canvasses

*UserCode*s are coded

*SMS*s send text messages

*Dial*s make phone calls

*Browse*s display web pages

*Close*s terminate applications

On event *Screen.enteredState:name*, produce event *Screen.draw*

On event *UserCode.enteredState:name*, produce event *UserCode.runCode*

On event **.enteredState:name*, produce event **.launch*

On event **.finished*, produce event **.handleEvent:"onfinish"*

On event *Screen.buttonClicked:id*, produce event *Screen.handleEvent:"onclick:"+id*

On event *Screen.listClicked:id,choice*, produce event *Screen.handleEvent:"onclick:"+id+"."+choice*

On event *Screen.listClicked:id,choice*, remember *choice* as *id+".*"*

On event *Screen.textChanged:id,text*, remember *text* as *id+".text"*

Figure 4.16: The *PhoneApp*$^{2.0}$ DSML, as a set of STs.

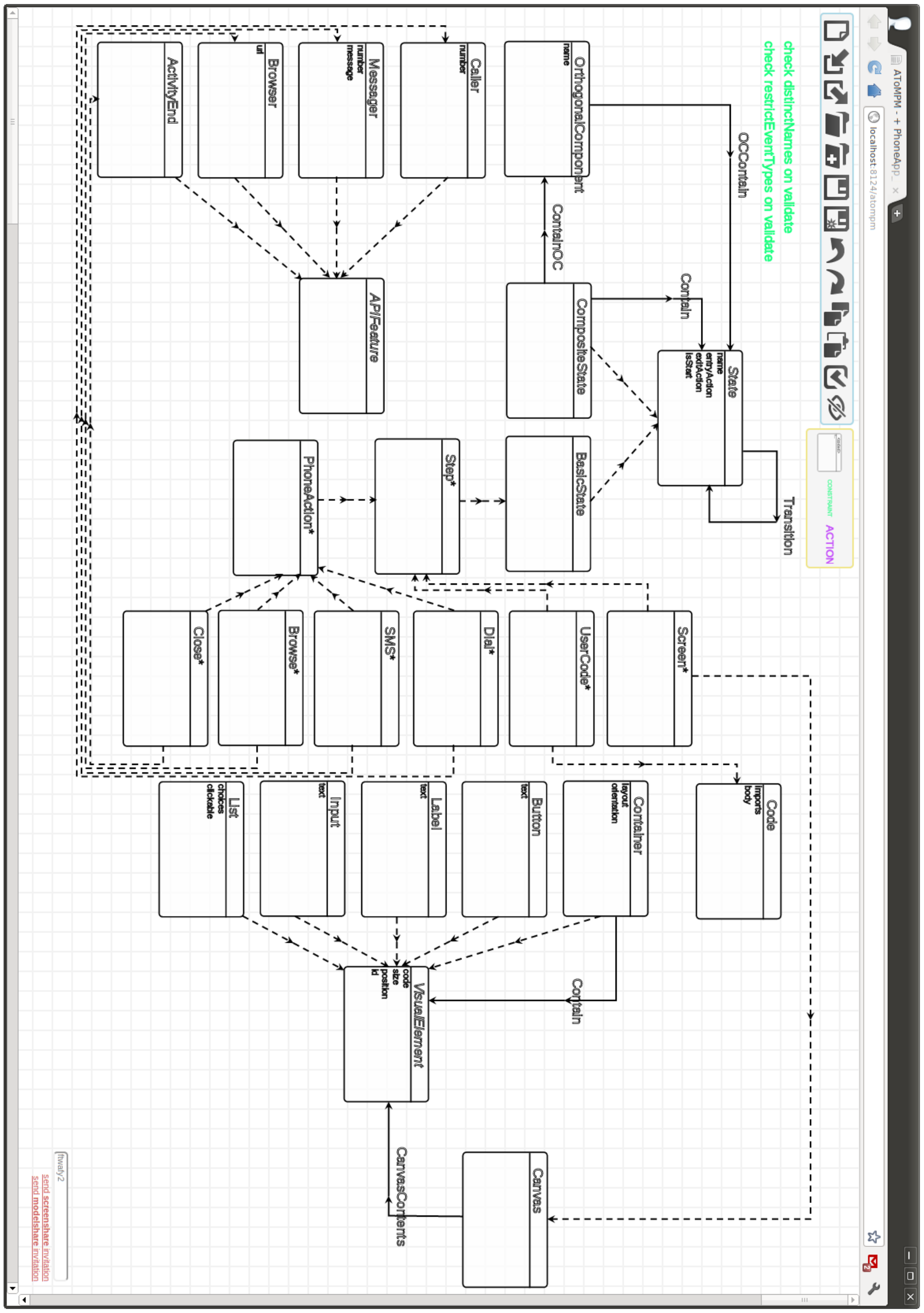


Figure 4.17: The *generated* PhoneApp$^{2.0}$ meta-model. Domain-specific concept classes are identified with a “*”.

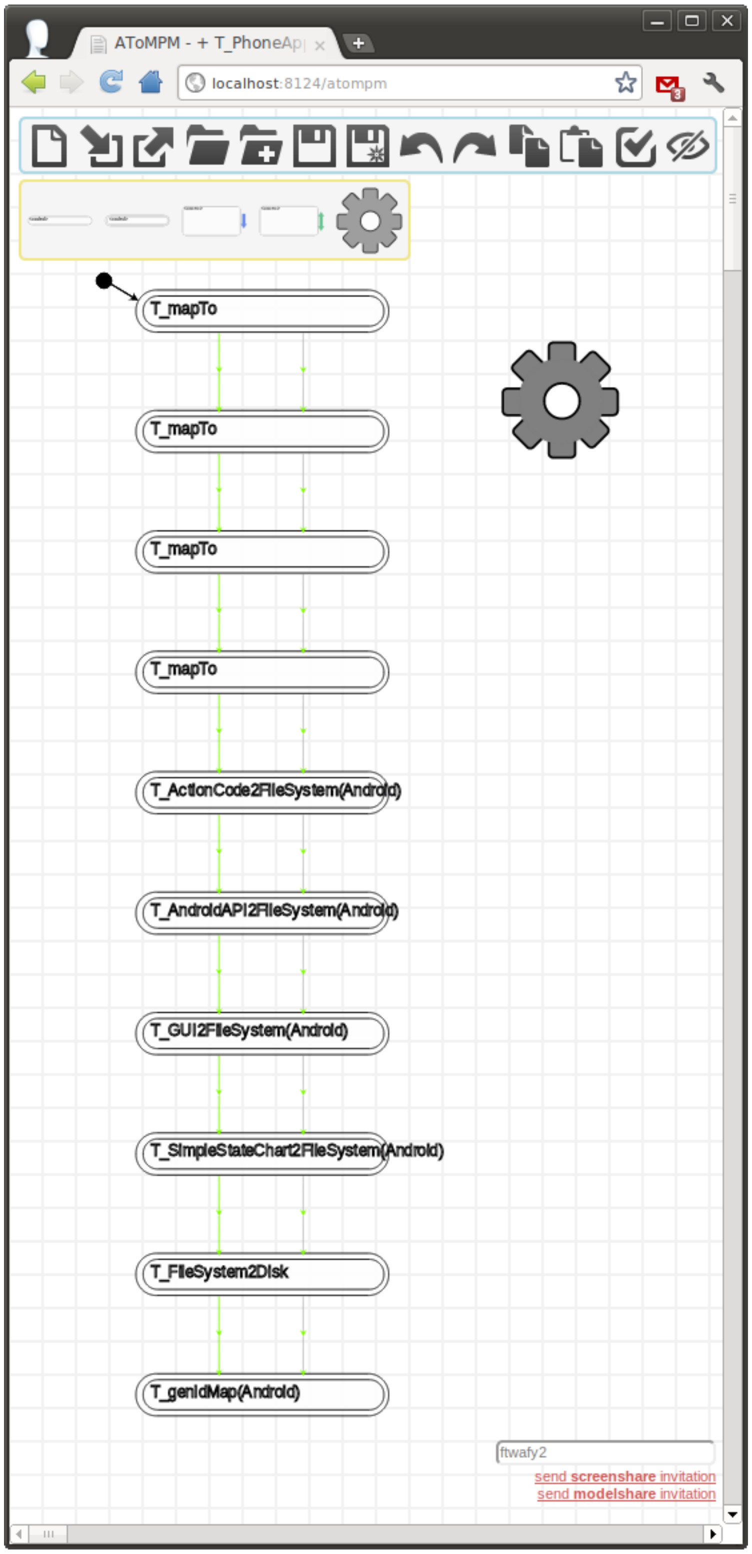

AToMPM - + T_PhoneAp
localhost:8124/atompm
T_mapTo
T_mapTo
T_mapTo
T_mapTo
T_ActionCode2FileSystem(Android)
T_AndroidAPI2FileSystem(Android)
T_GUI2FileSystem(Android)
T_SimpleStateChart2FileSystem(Android)
T_FileSystem2Disk
T_genIdMap(Android)
ftwafy2
send screenshare invitation
send modelshare invitation

Figure 4.18: The *generated* PhoneApp$^{2.0}$ top-level semantics model transformation.

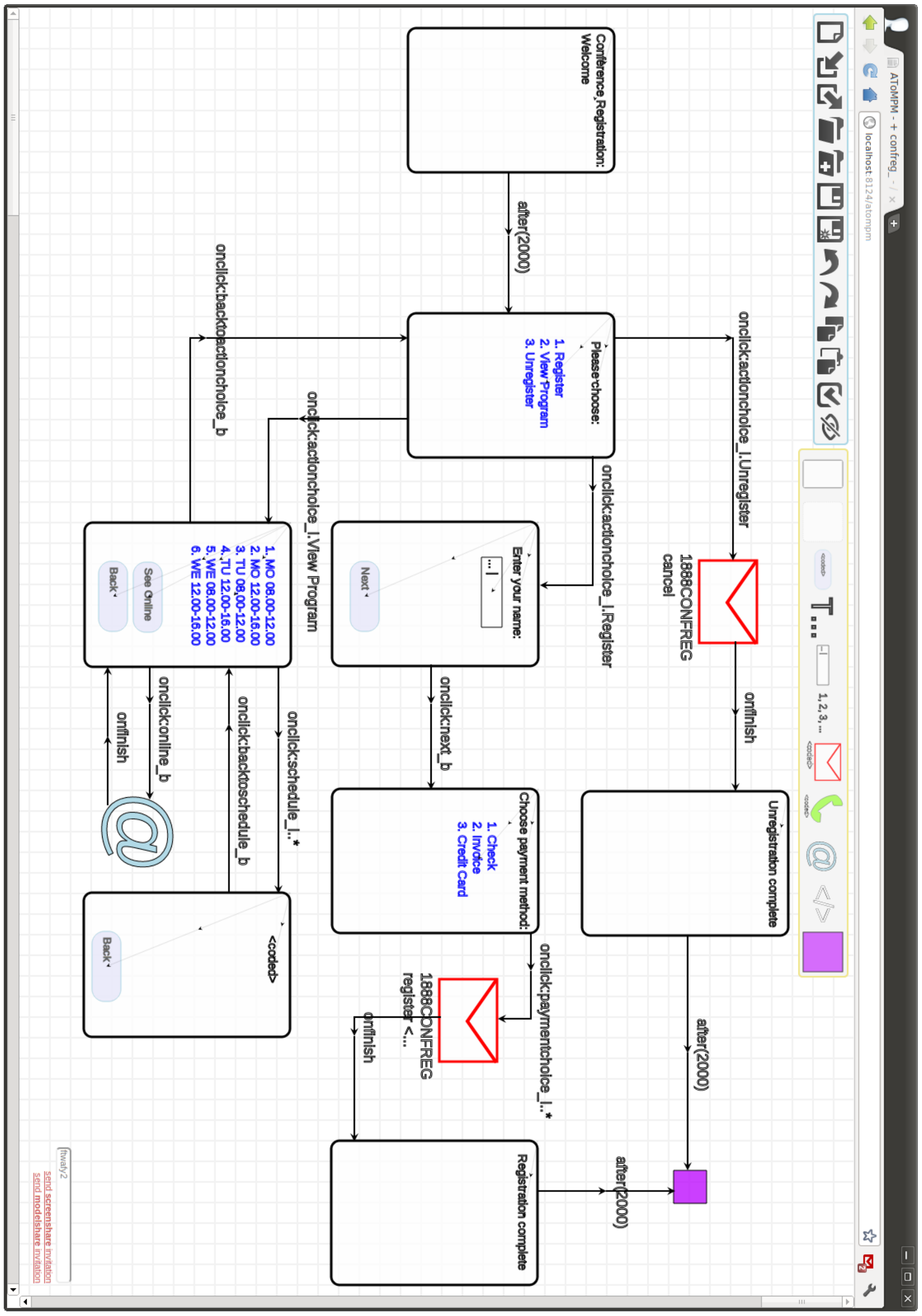


Figure 4.19: The conference registration application from Chapter 2, as a PhoneApp$^{2.0}$ model.

DSms and transform them into target platform artifacts. Figure 4.19 demonstrates this with a re-creation of the conference registration example from Chapter 2. There are some noticeable differences with the original conference registration model from Figure 2.19. First, the triggers on edges between `Step`s have changed to conform to the syntax specified by the event translation and data storage STs. Also, there is no more `Start` construct. Instead, each `Step` has an `isStart` attribute, exactly one of which is set to *true*. This is a direct consequence of the simplified version of the Statechart formalism (e.g., explicit "initial state" markers are replaced by the `isStart State` attribute) provided in AToMPM, which causes formalisms built on top of it to suffer from its limitations. This is discussed further in Section 4.6. Finally, note that the icon toolbar at the top of Figure 4.19 contains only relevant icons. For instance, there are no icons for `PhoneAction`, `Code` or `OrthogonalComponent`. As explained earlier, even though these are part of the synthesized meta-model, DSML designers can choose which concepts should be part of DSms, and which should not.

4.5.2 A DSML for Traffic Networks

The DSML presented in this subsection is arguably much simpler than PhoneApp$^{2.0}$. However, it will allow us to make explicit certain still unexplored capabilities and properties of our approach.

We wish to create a DSML for the modelling of traffic networks that satisfy the following description:

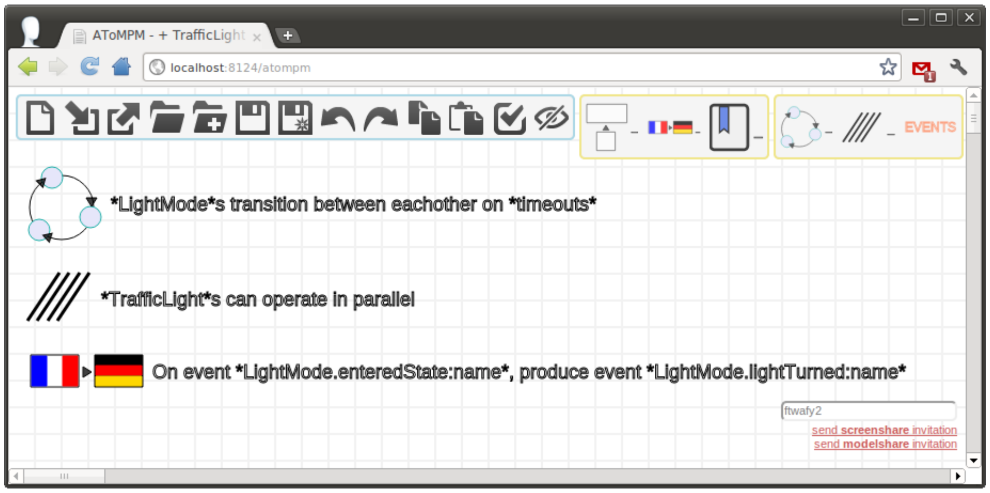
- Cars move randomly across a network of road segments connected via intersections;
- Traffic lights control access to intersections;
- Traffic lights are described by the timed transition between several states (e.g., red, green);
- Cars enter and exit the network randomly via highways;
- Accidents may block intersections;
- Accidents occur and are cleaned given arbitrary probabilities.

The concerns of this "domain" are the *non-deterministic* movement of cars between road segments via *blockable* intersections, the *delay- and state-based* behaviour of traffic lights, and the *probability-based* occurrence and cleansing of accidents. One possible approach to the specification of the *SimpleTraffic* DSML is to construct it on top of the Petri Net, Statechart and ActionCode base formalisms. However, a more modular and principled approach would be to model the inner workings of traffic lights at a higher level of abstraction and generate their associated Statechart model, thereby sparing the domain-specific modeller from the need to learn and use abstractions too tightly coupled to the Statechart formalism.

To this end, we introduce a new *non-native, pure* base formalism, *TrafficLights*, whose syntax and semantics are entirely defined using STs. Figure 4.20a shows the STs that define the TrafficLights DSML. Their implications, realized by generic and Statechart *genAS* and *mapTo* transformations, are that valid TrafficLights instance models may contain several `TrafficLight` entities, each containing an arbitrary number of `LightMode` entities connected to each other via delayed transitions, all of which can be mapped onto analogous `OrthogonalComponent` and `BasicState` Statechart structures. Additionally, a higher-level version of the Statechart `enteredState:name` event is exposed in its place. The realized forms of these implications are depicted in Figures 4.20b and 4.20c, which respectively show the generated abstract syntax, and generated top-level semantics model transformation of the TrafficLights DSML.

The TrafficLights DSML's syntax and semantics have been introduced. TrafficLights instance models can now be created. However, promoting this new DSML to base formalism status still requires three tasks to be performed. First, STs and input/output events must chosen. These will form an ST meta-model. Then, their meaning must be encoded in *genAS* and *mapTo* transformations. Figure 4.21 presents the sole ST of the TrafficLights base formalism. The first of its syntactic impacts is that a class is created for the `L` domain-specific concept and is made to subclass the `TrafficLightModel` class. Note that there is no such class in the generated TrafficLights meta-model from Figure 4.20b, which is not even imported, as was the case for all previously introduced STs. Indeed, importing base formalism meta-models is but one of several ways of realizing syntactic and semantic implications. In this particular case, it is an overly simple and insufficient means, as each `L` concept actually represents an entire structure within a TrafficLights model (i.e., a `TrafficLight` with contained and appropriately connected `LightModes`). Subclassing `TrafficLightModel`, beyond the obvious passing down of attributes necessary for the specification of a traffic light's modes and time spent in each of them before moving to the next, enables TrafficLights' *mapTo* transformation to match domain-specific entities while remaining generic. Indeed, with subtype matching enabled, specifying how `TrafficLightModel` entities should be mapped onto equivalent TrafficLights structures also defines how this mapping should be carried out for `L` entities. TrafficLights' *genAS* and *mapTo* transformations, which capture the above implications, are shown in Figure 4.22. Note that the *R_ImportMM* rule from the former produces the `TrafficLightModel` class rather than import the TrafficLights meta-model. As for the latter, its first rule produces one `TrafficLight` for each `TrafficLightModel` subtype. Next, `LightMode` entities are created inside `TrafficLights` as per their associated `TrafficLightModel` subtype's `modes` attribute. Lastly, the generated `LightModes` are appropriately connected via Statechart `Transitions` parametrized by timeouts that reflect the relevant `delays` attributes. The TrafficLight base formalism is not sensitive to any events, but produces the `LightMode.lightTurned:name` event.

Now that its constituting base formalisms have been clearly defined, a specification of the SimpleTraffic DSML can be created. One such specification is shown in Figure 4.23, and

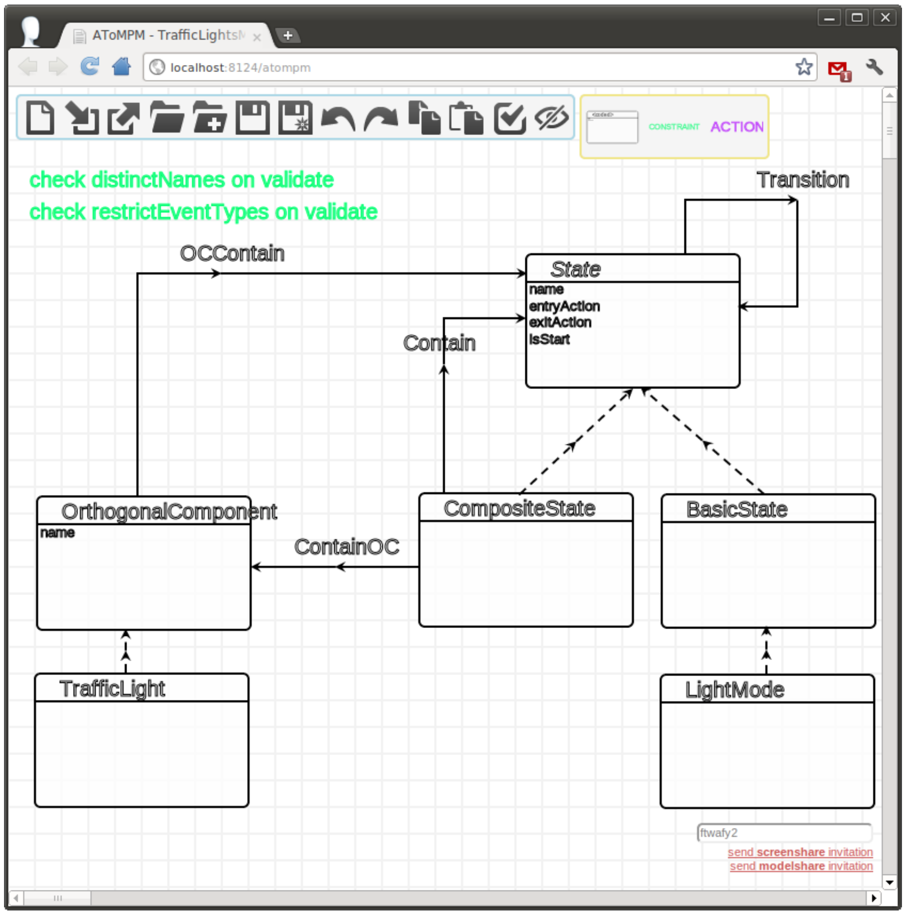


*LightMode*s transition between eachother on *timeouts*

*TrafficLight*s can operate in parallel

On event *LightMode.enteredState:name*, produce event *LightMode.lightTurned:name*

(a) The TrafficLights DSML, as a set of STs.

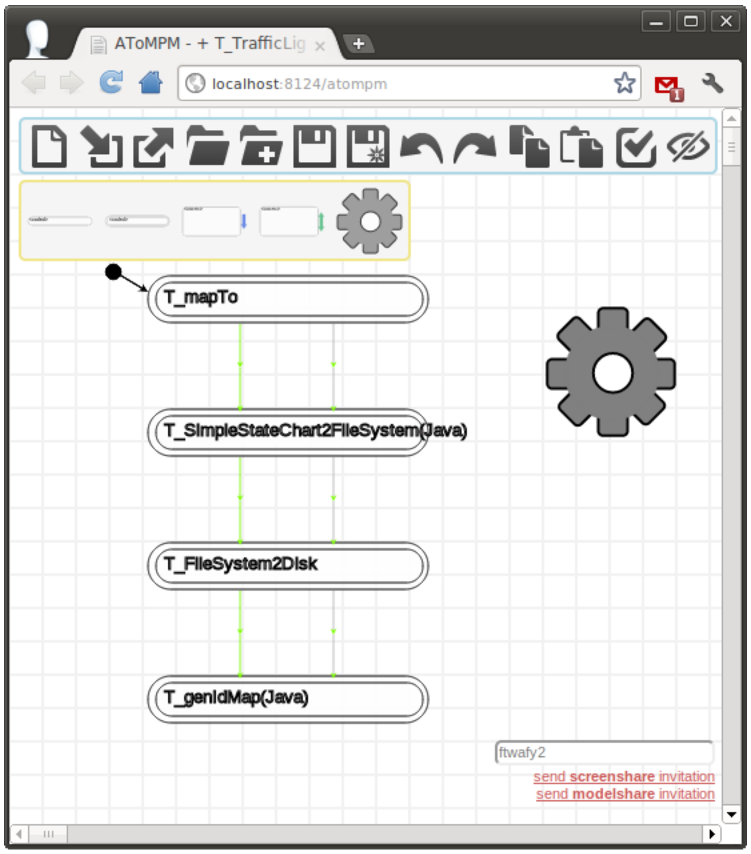


check distinctNames on validate

check restrictEventTypes on validate

(b) The *generated* TrafficLights meta-model.

Figure 4.20: The *TrafficLights* DSML *(continued)*.

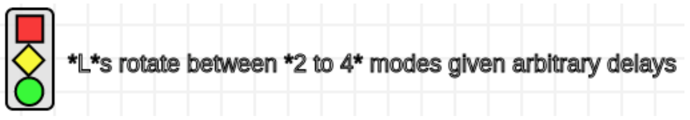


(c) The *generated* TrafficLights top-level semantics model transformation.

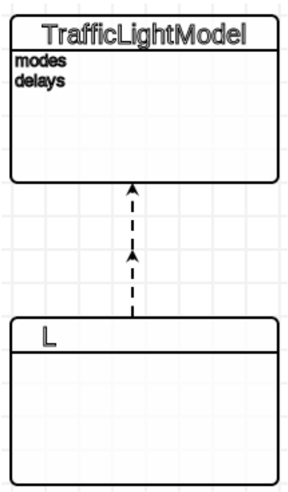
Figure 4.20: The *TrafficLights* DSML.

_s rotate between _ to _ modes given arbitrary delays

(a)



(b)



(c)

Figure 4.21: (a) The sole ST of the TrafficLights base formalism, (b) an example instance, and (c) its abstract syntax implications.

again in condensed, textual form in Table 4.3. Figure 4.24 shows the synthesized meta-model resulting from the syntactic implications of the STs that form the SimpleTraffic specification. As for the semantic implications of the STs that form the SimpleTraffic specification, they are mostly captured by the synthesized model transformation shown in Figure 4.25, which produces a Java application that simulates the DSm and intermediate representations (via backward link commands).

Given the abstract syntax and semantics generated from the ST representation of the SimpleTraffic DSML, a domain-specific modeller has all the necessary tools at hand to create DSms and transform them into target platform artifacts. Figure 4.26 shows a domain-specific SimpleTraffic model. Figure 4.27 shows the model from Figure 4.26 as well as all intermediate representations and traceability links produced during artifact synthesis. Noteworthy elements are that the number of current items displayed on each `RoadSegment` corresponds to the number of tokens in their associated `Place`, that the presence of a token in the `Place` corresponding to a `Light` dictates its current color, and that the presence of a token in the `Place` corresponding to an `Accident` dictates its current status.

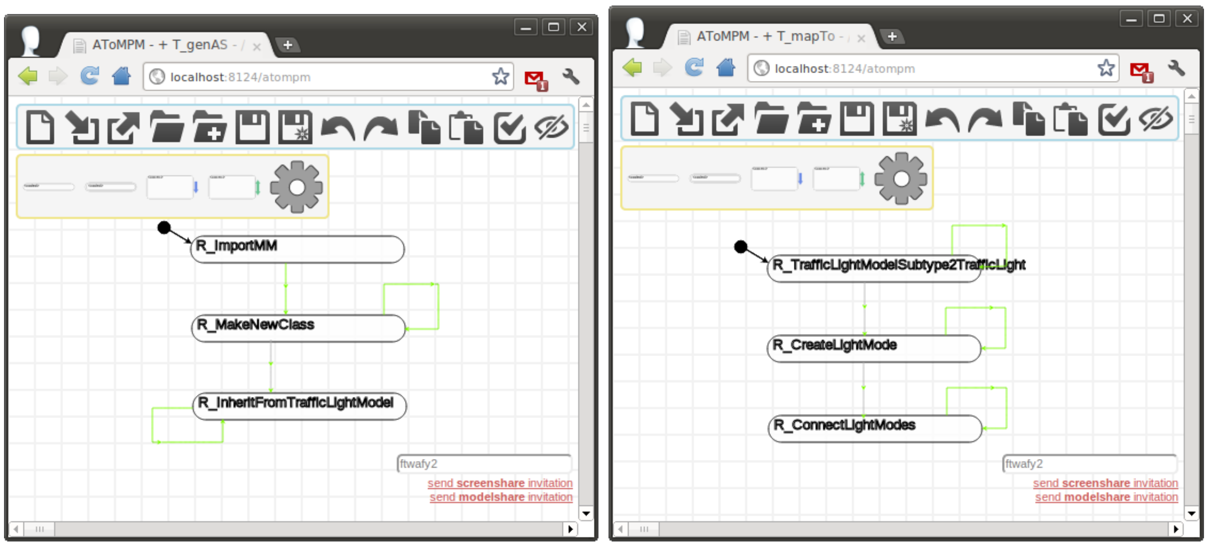
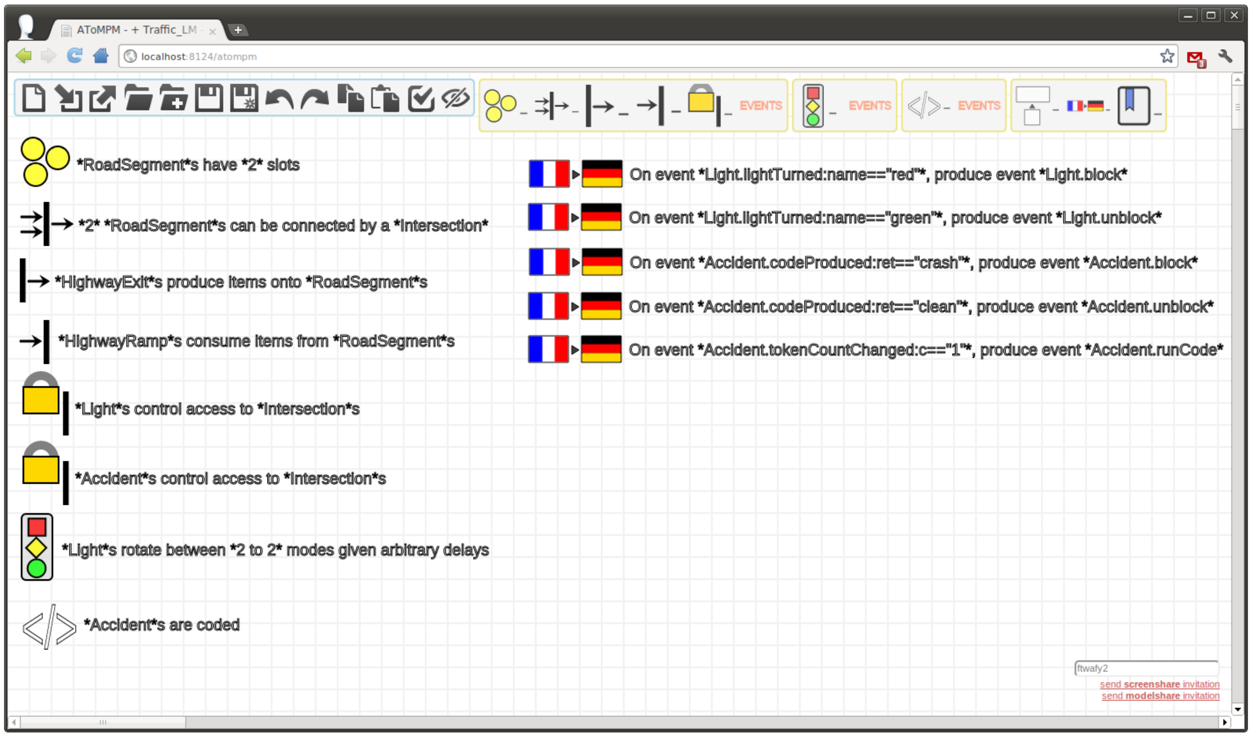

ATOMPM - + T_genAS
localhost:8124/atompm
R_ImportMM
R_MakeNewClass
R_InheritFromTrafficLightModel
ftwafy2
send screenshare invitation
send modelshare invitation
ATOMPM - + T_mapTo
localhost:8124/atompm
R_TrafficLightModelSubtype2TrafficLight
R_CreateLightMode
R_ConnectLightModes
ftwafy2
send screenshare invitation
send modelshare invitation
(a)
(b)

Figure 4.22: (a) The *TrafficLights genAS* and (b) *mapTo* model transformations.



*RoadSegment*s have *2* slots

2 *RoadSegment*s can be connected by a *Intersection*

*HighwayExit*s produce items onto *RoadSegment*s

*HighwayRamp*s consume items from *RoadSegment*s

*Light*s control access to *Intersection*s

*Accident*s control access to *Intersection*s

*Light*s rotate between *2 to 2* modes given arbitrary delays

*Accident*s are coded

On event *Light.lightTurned:name=="red"*, produce event *Light.block*

On event *Light.lightTurned:name=="green"*, produce event *Light.unblock*

On event *Accident.codeProduced:ret=="crash"*, produce event *Accident.block*

On event *Accident.codeProduced:ret=="clean"*, produce event *Accident.unblock*

On event *Accident.tokenCountChanged:c=="1"*, produce event *Accident.runCode*

Figure 4.23: The *SimpleTraffic* DSML, as a set of STs.

Formalism	**Template**
Petri Net	`RoadSegment`s have 2 slots.
Petri Net	2 `RoadSegment`s can be connected by an `Intersection`s.
Petri Net	`HighwayExit`s produce items onto `RoadSegment`s.
Petri Net	`HighwayRamp`s consume items from `RoadSegment`s.
Petri Nets	`Light`s control access to `Intersection`s
Petri Nets	`Accident`s control access to `Intersection`s
Petri Nets	`Light`s unblock `Intersection`s on event `Light.lightTurned:green`
TrafficLights	`Light`s rotate between 2 to 2 modes given arbitrary delays.
ActionCode	`Accident`s are coded.
< *generic* >	On event `Light.lightTurned:name=="red"`, produce event `Light.block`.
< *generic* >	On event `Light.lightTurned:name=="green"`, produce event `Light.unblock`.
< *generic* >	On event `Accident.codeProduced:ret=="crash"`, produce event `Accident.block`.
< *generic* >	On event `Accident.codeProduced:ret=="clean"`, produce event `Accident.unblock`.
< *generic* >	On event `Accident.tokenCountChanged:c=="1"`, produce event `Accident.runCode`.

Table 4.3: The SimpleTraffic DSML, as a set of STs (in condensed, textual form).

The SimpleTraffic case study presents two key points of interest. The first is that, unlike previous domain-specific concepts which merely specialized base formalism concepts, each `Light` entity effectively encompasses a non-trivial base formalism structure. This demonstrates our approach's ability to cope with more complex syntactic and semantic mappings from DSm to base formalism. The second is the insight that the combination of base formalisms may not be a strictly additive process. The combination of the Petri Net, Statechart and ActionCode formalisms, and especially the programmatic editing of Petri Net markings to otherwise unreachable markings (e.g., via the `block` and `unblock` events), causes formal analysis on the resulting Petri Net intermediate representation to be impossible and/or meaningless, as this effectively modifies Petri Net semantics. Means to enable formal analysis with no sacrifices made to the DSML's expressiveness might introduce undesired accidental complexity into the DSML specification, and would probably require the use of base formalisms for which mappings onto Petri Net instance models exist (and are available). Nevertheless, if analysis is not required, the provided ST representation of the SimpleTraffic DSML fully captures the described domain.

Lastly, one might argue that the notion of Petri Net "tokens" is at too low a level of abstraction and that it might be more appropriate for SimpleTraffic `RoadSegments` to have a `nbCars` attribute rather than a `nbTokens` attribute. One option is to introduce more

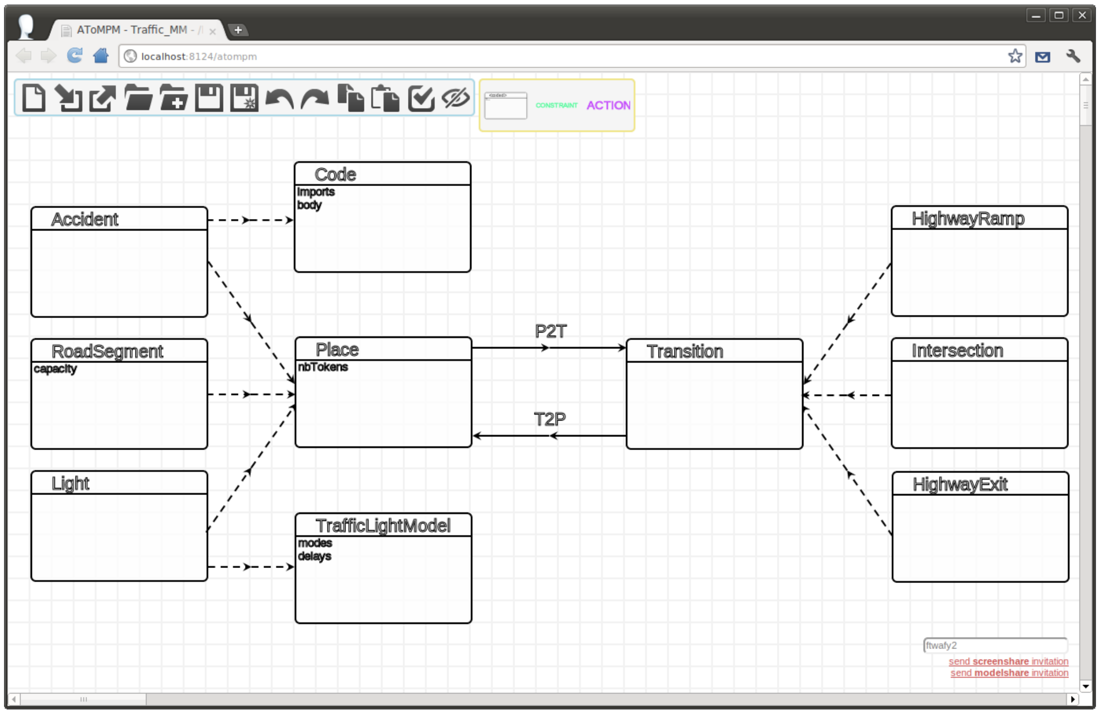


Figure 4.24: The *generated* SimpleTraffic meta-model.

elaborate STs and take a non-specialization approach to abstract syntax generation (as was the case with TrafficLights). For instance, the ST from Figure 4.6 could become “$\underline{X}$s have $\underline{\infty|k}$ slots for $\underline{\texttt{cars}}$”. Then, rather than subclass `Place` (and inherit the `nbTokens` attribute), $\underline{X}$ could be given a `nbCars` attribute, and be made to subclass some other class, call it `Place`$^{++}$. A small extension to the Petri Net *mapTo* transformation would suffice to map `Place`$^{++}$ subtype entities onto equivalent `Place` instances. Another, perhaps less convoluted, approach might be to replace the Petri Net base formalism by one of its more convenient yet semantically equivalent extensions, such as the Coloured Petri Net formalism [Jen86], where tokens are explicitly modelled. Indeed, if a `Token` class were present, it could be subclassed (e.g., by a `Car` class) thereby achieving the desired outcome (of hiding the Petri Net-specific notion of tokens) much more easily. This discussion is reminiscent of the earlier one regarding the missing `Start` concept in PhoneApp$^{2.0}$, and is taken up again in the next section.

4.6 Discussion

This section provides a detailed discussion of our approach's extensibility, applicability and limitations, as well as a critical comparison with traditional DSML design and engineering practices.

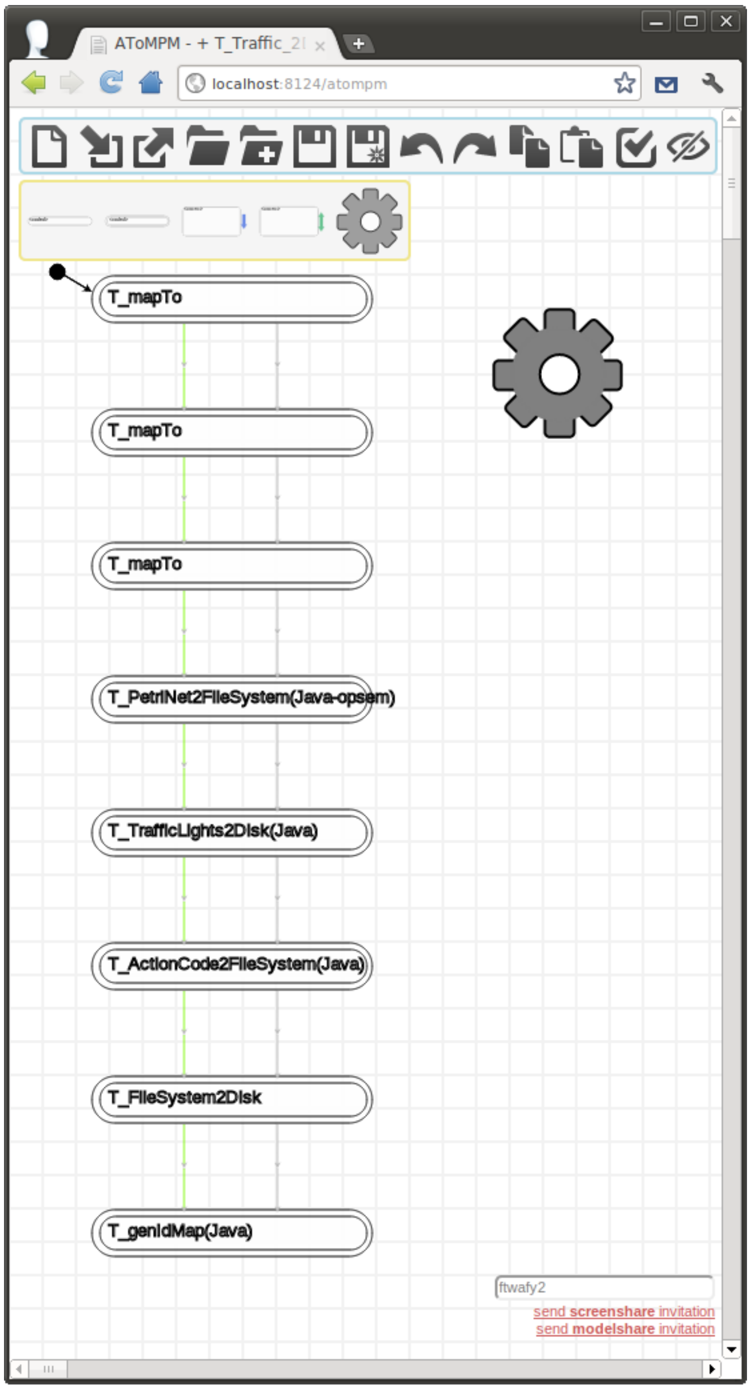

AToMPM - + T_Traffic_2
localhost:8124/atompm
T_mapTo
T_mapTo
T_mapTo
T_PetriNet2FileSystem(Java-opsem)
T_TrafficLights2Disk(Java)
T_ActionCode2FileSystem(Java)
T_FileSystem2Disk
T_genIdMap(Java)
ftwafy2
send screenshare invitation
send modelshare invitation

Figure 4.25: The *generated* SimpleTraffic top-level semantics model transformation..

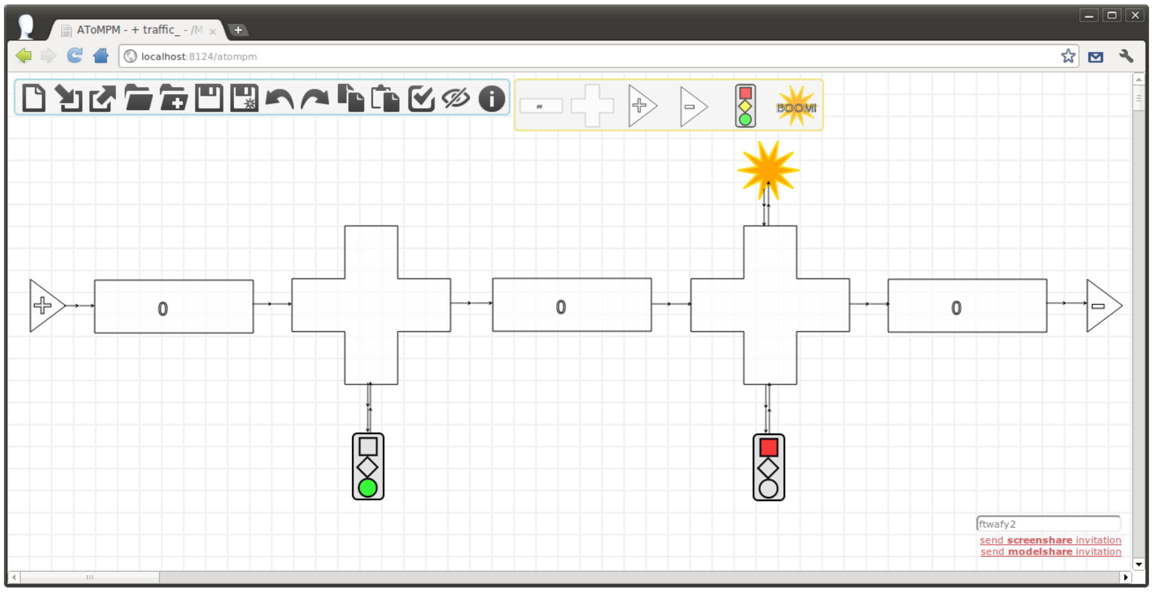


Figure 4.26: A domain-specific SimpleTraffic model.

4.6.1 Extensibility

Past template-based development approaches, particularly in the context of programming, have been met with much reserve. The main reasons are their restrictiveness and limited flexibility. Both of these pitfalls are especially relevant in the context of our approach. Indeed, the number (or rather the variety) of sensible combinations of six base formalisms may appear somewhat limited. Furthermore, the provided base formalisms may be at too low a level of abstraction for many DSML design efforts. To address these concerns, we introduced *non-native* base formalisms in Section 4.5.1 as our approach's built-in extensibility means. Indeed, empowering DSML designers with the means – which mainly consist of facilities for the specification of new STs and their syntactic and semantic implications, all of which are natively provided by modern modelling and meta-modelling tools – to promote their own DSMLs to base formalism status enables the increase in expressiveness, variety and flexibility of the available collection of STs. Of course, increases in expressiveness only result from the addition of base formalisms that aren't constructed from existing base formalisms. These instead serve to elevate the abstraction and increase the overall *usefulness* of the set of available base formalisms, as was demonstrated with the TrafficLights formalism.

Allowing the unbounded growth of the number of base formalisms and STs may lead to a framework that is too complex and/or cluttered to use. Consequently, a mature ST-enabled tool should group *semantically equivalent* base formalisms together beneath a single banner. Beyond the obvious, we define two base formalisms F_i and F_j as semantically equivalent if F_i can be represented using a limited (finite) number of F_j's STs, and vice versa. Having several semantically equivalent base formalisms available may be useful to reduce accidental complexity in DSML specifications. For instance, although Statechart and DEVS models can

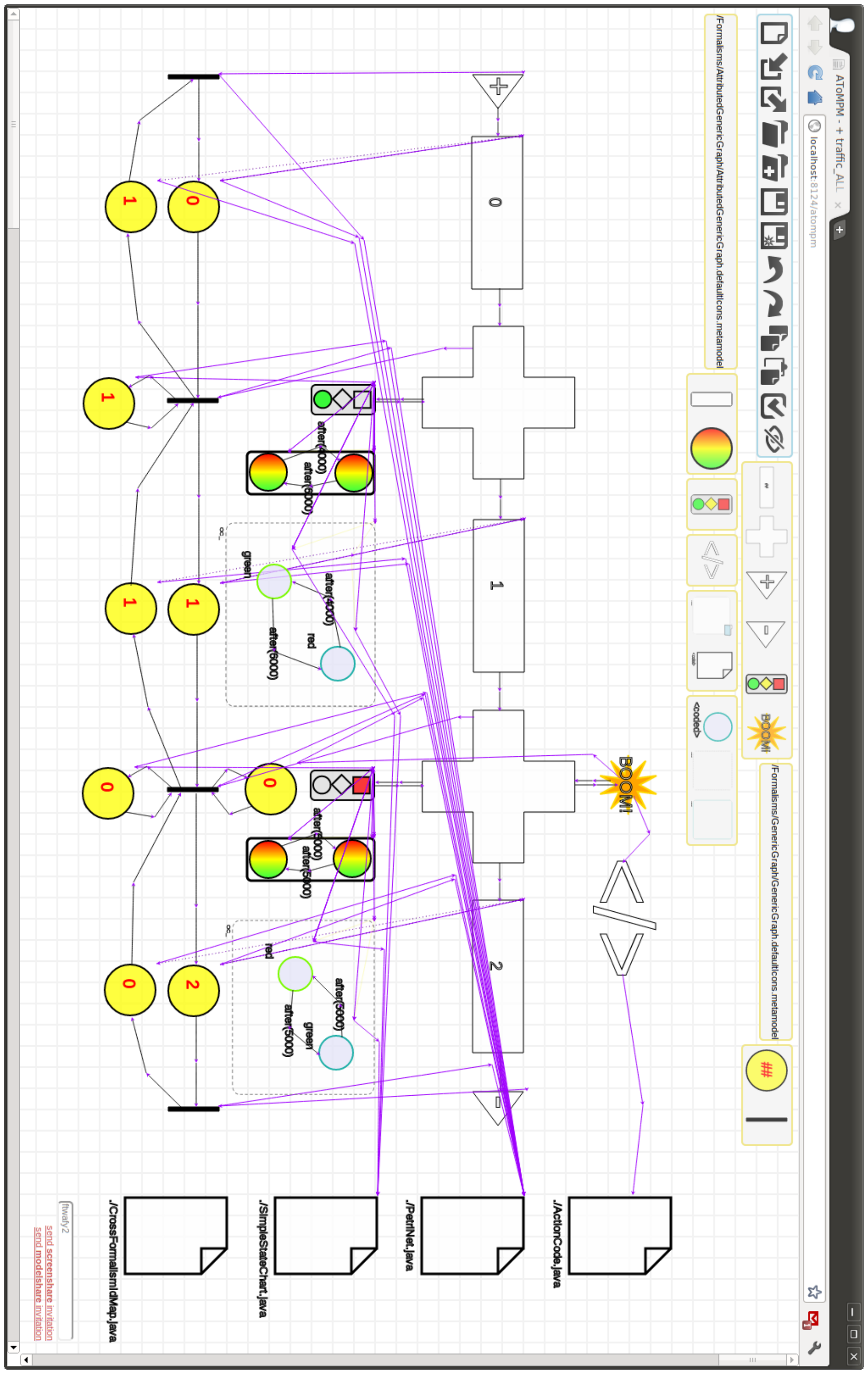

AToMPM - + traffic_ALL
localhost:8124/atompm
/Formalisms/AttributedGenericGraph/AttributedGenericGraph.defaultIcons.metamodel
/Formalisms/GenericGraph/GenericGraph.defaultIcons.metamodel
0
1
2
after(4000)
after(5000)
green
red
BOOM!
<coded>
./ActionCode.java
./PetriNet.java
./SimpleStateChart.java
./CrossFormalismIdMap.java
send screenshare invitation
send modelshare invitation

Figure 4.27: An executing domain-specific SimpleTraffic model with visible intermediate representations and traceability links.

readily be mapped onto one another, it may be unnatural for certain DSMLs to be specified using STs of one formalism rather than the other. In practice, upon choosing which classes of formalisms he wishes to build his DSML on, a designer could peer further into each class and choose the formalisms and/or STs he wishes to make use of.

In the interest of establishing theoretical foundations for the combination of low-level formalisms to form higher-level ones, we do prefer to clearly distinguish (at least internally) between native and non-native base formalisms. The core difference between them is that native base formalisms can not be re-constructed by combining other base formalisms, while non-native base formalisms can. As an exception, since any formalism can theoretically be mapped onto code (and thus re-constructed using code), the only mappings onto code that grant non-native status are trivial ones. Thus, the base formalisms introduced in Section 4.3 are all native base formalisms, while those introduced in Section 4.5, namely, AndroidAPI and TrafficLight, are both non-native.

4.6.2 Limitations and Applicability

Escape semantics enable the use of lower-level abstractions within higher-level models. For instance, in PhoneApp, arbitrary Java or Android-specific code can be specified in `ExecuteCode` entities. Such code might carry out the rendering of an unsupported widget or enact a device-specific activity not included in the PhoneApp DSML (e.g., taking a camera snapshot). Although escape semantics are somewhat contrary to the DSM and MPM paradigms of development, which recommend shielding developers from concepts outside of the problem domain, they are often a necessary evil. On the one hand, they offer means for a DSML to capture an entire domain without being overly coupled to it, by allowing the manual specification of certain details using non-domain-specific constructs (such as code). Similarly, as reported by Safa in [Saf07], the overhead associated with deploying a DSM development environment can be considerably reduced if a more minimalistic DSML is adopted. Designing and developing a DSML that captures most common scenarios in a domain using domain-specific constructs while providing lower-level means to support possibly many fringe cases can be a sound compromise when considering the time and effort required to define abstract syntax and semantics for each domain-specific concept. This is especially true in contexts where developers have some level of familiarity with target platform artifacts and where DSM is introduced to speed up development rather than to enable non-developers (e.g., code-oblivious arbitrary domain experts) to participate in it. On the other hand, escape semantics offer a form of isolation from domain evolution, by ensuring that a DSML remains useful (or at least usable) until it can properly capture newly added domain concepts using domain-specific entities. Thus, although they often draw attention to what appear to be the limitations of DSM, escape semantics are merely means to preserve the flexibility that is lost when raising the level of abstraction of development efforts.

In the context of DSML design, being overly coupled to the domain can be correlated with providing too many STs. Indeed, in the limit, each possible instance of a base formalism could be captured by a different ST. As for domain evolution, it could be understood as the emergence (or existence) of domains that can not be elegantly modelled by any combination of the available base formalisms. Thus, for our approach to be truly widely applicable despite its raising of the level of abstraction of the specification of DSML syntax and semantics, an escape semantics solution is necessary. To this end, we introduced *non-native, impure* base formalisms in Section 4.5.1. Empowering DSML designers with the means to promote their own DSMLs to base formalism status without having to (re-)build them using STs ensures that lacking base formalisms will never entirely hamper the applicability of our approach. In theory, every member of the current set of native base formalisms can be seen as impure, since neither one of them is built using STs, and each of them was introduced to capture at least one concern that could not be elegantly[7] projected onto any of the others. By extension, the inability to elegantly map a given concern onto an available base formalism is an effective means of identifying a missing base formalism (or formalism class). Recall the accidental complexities we underlined in Section 4.5, namely, the lack of a dedicated construct to identify initial Statechart states and of simple means of abstracting the notion of Petri Net tokens. These were both strong indicators of deficiencies in the current set of base formalisms. In both cases, solutions can be devised without replacing either base formalism, i.e., mere adjustments to their *mapTo* transformations are sufficient. If that were not the case however, the sum of missing concepts could help shape the outline of a missing base formalism (or formalism class).

Another, perhaps less extreme, context where generating syntax and semantics may be undesirable is that of performance-critical systems. For instance, artifacts produced from PhoneApp DSms should, in theory, be more efficient that those produced from PhoneApp$^{2.0}$ DSms. Indeed, the manual merging of intermediate representations produces a single instrumented artifact whereas the proposed means of automating this merging produces a number of disjoint artifacts which must create, transmit, wait for, decode and react to (or ignore) a wide variety of events. While having several artifacts communicate is arguably easier to implement (and automate) than literally merging bodies of code, event management adds an undeniable performance overhead that may be unacceptable in certain scenarios. Thus, performance-critical systems may favour the approach from Chapter 2 over a higher-level ST approach to DSML specification. This limitation is not strictly bound to our approach, however. The use of low-level assembly code is still common practice to implement performance-critical tasks within systems implemented in higher-level programming languages.

Our approach shares another limitation with past and current attempts at raising abstraction. The expressiveness of a collection of STs in terms of the spectrum of “reachable” target platforms is limited by the availability of “base formalism to target platform” transformations. This limitation is analogous to that of a programming language being all but useless if programs written in that language can not be compiled to deployment platforms. Consider a DSML designer who wishes to model a state-based domain and requires Ruby

[7]As mentioned earlier, we overlook the fact that any concern can be mapped onto code.

code to be generated from DSms. He may be forced away from our approach by the absence of a Statechart-to-Ruby transformation. To circumvent this obstacle, he may choose the traditional approach to artifact generation from DSms and manually program a DSm-to-Ruby code generator. Alternatively, he might choose to enhance an ST-enabled tool with a Statechart-to-Ruby transformation. Both options will fulfil his needs. However, the latter option has the potential to benefit a much wider audience. Thus, we argue that the reuse of past work and leveraged expertise that our approach enables make it a more than viable alternative to traditional DSML specification techniques.

Perhaps the most fragile aspect of our approach is its dependency on thorough documentation. Indeed, the implications of a ST must be very well documented and understood for it to be usable. Given that the numerous layers of abstraction between DSML specification and target platform artifact introduced by our approaches to artifact synthesis and DSML design each hide and automate translation steps, it may be difficult to ascertain the validity and/or meaning of a DSm or DSML other than by trial and error. This problem is exponentially worsened when STs are ambiguous and even their translation to the next closest layer of abstraction (i.e., DSML abstract syntax and semantics) is unclear. Thus, a mature ST-enabled tool should provide clear descriptions of the implications of each ST in (ideally) layman's terms, and of the relationship(s) between various STs, if any. Once again, this limitation is not strictly related to our approach, but rather another side-effect of automating partial design tasks. Similarly, an undocumented Java API with cryptic method and property names can be unusable despite the enormous success of code-centric engineering and APIs.

An issue that is especially relevant to our approach is that of composing artifacts from heterogeneous models of computation. In the provided examples, no overly complex interactions were encountered. For PhoneApp$^{2.0}$, all artifacts were rooted in the discrete model of computation. For SimpleTraffic, although there is no notion of time in Petri Net models, the blocking of Petri Net `Transitions` by the Statechart and ActionCode artifacts imposed some notion of time by periodically halting and then re-enabling the flow of tokens. Thus, for the provided examples, our central event manager solution to the problem of artifact composition sufficed. However, consider the example of a "hybrid" model where one of the artifacts is based on the Causal Block Diagram formalism, where time is continuous, while another is based on the discrete-time Statechart formalism, such as the modelling of a bouncing ball. It is arguably unrealistic for the first artifact to emit thousands or millions of events each second to report changes in the input and output values of its entities (e.g., changes in velocity and position). The problem of composing such seemingly irreconcilably different systems is often solved, in *co-simulation* tools, via thresholding or boundary conditions. Concretely, the continuous artifact from the bouncing ball example might be made to report velocity only when it changes orientation (e.g., from going up to going down and vice versa), thereby vastly reducing and effectively discretizing its output. Such thresholding behaviour is trivially achieved within our approach but would admittedly require revisiting

Steps	Traditional	Proposed
Domain analysis, resulting in lists of concerns and requirements	√	√
Choice of base formalisms	√ or *n/a*	√
DSML definition (as collection of instantiated STs)	*n/a*	√
Meta-model definition (e.g., as UML Class Diagram augmented with constraints)	√	*automated*
Semantic mapping definition (e.g., as coded text generator or as model transformations)	√	*automated*
Debugging/testing of semantic mapping	√	*obsolete*
DSm creation, artifact synthesis, etc.	√	√

Table 4.4: The traditional and proposed workflows of DSML design and engineering shown side-by-side.

the events exposed by the Causal Block Diagram formalism.

Finally, the issue of artifact composition, or rather the solutions we proposed to enable artifact composition, hint at a broad limit of our approaches applicability. The assumptions that base formalisms either inherently input and output events or can easily (and sensibly) be made to do so limits our approach to scenarios where those assumptions hold. In the arguably vast context of software development, where time, data and control flow are all discrete, these assumptions do indeed hold. However, our approach may prove inadequate for the modelling of certain physical systems, for instance, where discretization for event-based artifact composition purposes may be neither easy nor sensible given the modelled system's intended semantics.

4.6.3 Evaluation

Throughout this work, we have argued that the approach we propose improves upon traditional DSML design and engineering practices on numerous fronts. A brief summary of these claimed improvements is that our approach brings the benefits often attributed to DSM to DSM itself. Indeed, the provided case studies – specifically the first – demonstrate the vast difference between the amount of user-input required by the manual specification of a DSML's abstract syntax and semantics, and the specification of the same DSML as a collection of STs. Table 4.4 furthers this demonstration by illustrating the traditional and proposed workflows of DSML creation side-by-side, with automated and obsolete tasks clearly marked. Beyond the already claimed raises in abstraction and productivity, the automation of the repetitive and non-trivial tasks that characterize traditional DSML design and engineering also eliminates the need for numerous debugging activities as many bug sources are effectively absorbed within the ST framework.

At first glance, the sheer difference in the length of required user-input seems to indicate that our approach should significantly increase productivity. Certainly, it has been our experience that developing DSMLs using STs is incomparably easier and expedient. Such empirical evidence, based mostly on author testimonials, has been reported by Kelly and Tolvanen in [KT08], as confirmation of the promised benefits of DSM (versus code-centric development) in industrial settings. Safa is slightly more formal in [Saf07], where development times of experienced programmers are compared to those of a new employee trained only to use a DSM environment. However, a better and stronger demonstration of the benefits of our approach would be to conduct a large scale study, akin to that performed by the Middleware Company in their evaluation of MDA-based development [Mid03]. DSM experts and novices could be asked to work in parallel – and in a controlled environment – on the same problem (e.g., the full specification of a non-trivial DSML). Highly insightful quantitative and qualitative metrics would include time to task completion, number of bugs, number of clicks and keystrokes, user perception of task difficulty and overall user appreciation. Intuitively, DSM novices should prefer and be more productive with a ST approach to DSML design. Ideally, this would also be the case for DSM experts. Without overlooking its enormous importance to the validation of our claims, we defer such a study to future work.

4.7 Comparison with Related Work

This section reviews and situates the introduced approach with respect to relevant work by others surveyed in Section 4.2.

First and foremost, ST DSML specifications are at a higher level of abstraction than traditional DSML abstract syntax and semantics specifications. While these are still very much present, they are demoted from first class artifacts to synthesized internal representations. Thus, it is indeed more relevant to compare our approach to other efforts that targeted formalism combination and the automation of syntax and semantics specification.

It turns out Vallecillo's work on the merging of interrelated models and meta-models that capture different views of a given system bares little resemblance to our own. Indeed, despite arguably similar vernacular, our motivations and problem context are very different. The focus of Vallecillo's is on resolving correspondences between models and meta-models. These are trivially known in the context of our work. Instead, we attempt to combine (ideally) unrelated formalisms.

As for authors that tackled the problem of DSML combination from an engineering perspective, studying how recurring structures can be turned into generic building blocks, their work is intimately related to our own. *Template instantiation* by Emerson and Sztipanovits bears a strong resemblance to what is achieved by our base formalism *genAS* transformations. However, by providing only isolated templates and targeting only the generation of abstract syntax patterns, their approach offers little more than syntactic sugar. The level of abstraction of abstract syntax specification is left unchanged (as is that of the ignored specification of semantics), most of which is (ostensibly) still performed manually. Similar reasoning applies to Pedro's work, who additionally provides templates for specifying arbitrary model transformations. In contrast to theirs, our approach *entirely* automates the definition of abstract syntax models and of semantics model transformations, requiring input only at a higher level of abstraction.

At first glance, White et al.'s work on specifying DSML combination via Feature Diagrams seems very similar to our own. However, a key difference is that their motivations (i.e., creating many *similar* meta-models) leads them to take a "reductive" approach whereas we take a more "constructive" approach. Indeed, in White et al.'s work, all possible meta-models are known and one must prune and choose features to reach one of them. In contrast, we propose that base formalisms be combined to produce one of an infinite set of DSMLs. We also don't restrict our attention to meta-models.

Despite a significant body of research and experience, our own included, on organizing and structuring the specification of DSML semantics, most modern approaches still rely on the ad hoc manual input of semantic mappings. With *semantic anchoring*, Chen et al. were the first to propose the use of pre-defined semantic and syntactic building blocks to assist in DSML specification in a somewhat unified manner. Unfortunately, their work fails to truly elevate the level of abstraction of syntax or semantics specifications as only partial specifications are generated and manual input is required to complete their generated forms. Furthermore, their approach places a heavy burden on tool developers (often DSM experts) as they must specify the aforementioned building blocks and their implications. Incidentally, the authors do not report on how this is achieved. Finally, due to the fact that the building blocks are chosen arbitrarily, more theoretical claims (e.g., pertaining to the current expressiveness of the framework) become difficult to assert. In comparison, our approach truly does raise the level of abstraction of DSML specification by completely shielding DSML designers from generated abstract syntax and semantics definitions. Moreover, it does so in a theoretically sound manner that is amenable to some form of analysis.

In summary, the approach presented in this chapter is a combination and extension of past and current work on template instantiation and semantic anchoring with the specific aim of *fully* generating complete DSML abstract syntax *and* semantics specifications from higher-level models of DSMLs.

4.8 Conclusion and Future Work

The work presented in this chapter was motivated by the fact that the driving principles behind DSM have yet to be incorporated into the workflow of DSML design and engineering. Indeed, the guidelines of leveraged expertise and of automating the generation of complex yet repetitive artifacts (through the use of higher-level abstractions) are often disregarded in modern DSML design and engineering efforts. Consequently, DSM experts find themselves forced to repeatedly and manually specify possibly similar and even overlapping meta-models and semantic mappings from DSms to target platforms. The latter, especially, are often non-trivial and require a non-negligible level of familiarity with targeted platforms.

The approach we propose in this work provides DSM experts with the full power of DSM. We introduce a template-based technique that enables the construction of new DSMLs by combining lower-level *base formalisms* that capture commonly recurring concepts in DSMLs. In our approach, the DSML designer creates unified representations of abstract syntax and semantics at a much higher-level of abstraction than either of their traditional specifications. From these unified representations, complete abstract syntax and semantics are generated. This synthesis is enabled by the information carried within the templates in our framework, namely, the means for DSms to be projected onto semantically equivalent base formalism instances. Our approach is both extensible and widely applicable as it supports the addition of arbitrary new base formalisms and templates. Moreover, although it remains bound by limitations of previous attempts at raising the level of abstraction of development efforts, we argue that its focus on reuse and leveraged expertise make it a more than viable alternative to traditional DSML specification techniques.

We demonstrated our approach in two non-trivial case studies in which we showed how syntax and semantics could be generated from very concise and high-level unified representations of DSMLs for modelling mobile device applications and traffic networks. However, we recognize that a more objective validation is required. As such, we suggest that a user study be conducted to formally compare several metrics, including productivity and amount of user-input, between traditional DSML design and engineering techniques and our proposed approach.

Chapter 5

AToMPM: A Tool for Multi-Paradigm Modelling

This chapter introduces AToMPM, A Tool for Multi-Paradigm Modelling developed as a platform for the implementation of the contributions of this thesis. It replaces AToM^3, A Tool for Multi-formalism and Meta-Modelling. This chapter details the objectives and requirements that guided AToMPM's development, as well as its architecture, features, and technical and scientific innovations.

5.1 Context, Objectives and Requirements

There exists a number of DSM tools. Some are stand-alone applications, such as the commercial MetaEdit+ and the academic AToM^3, GME, Fujaba [NZ99] and VMTS. Others are plug-ins grafted on the Eclipse IDE, such as GEMS [WSNW07], Actifsource [act] and Fujaba4Eclipse [HT08]. Finally, the Eclipse Foundation have themselves enhanced their IDE with the Eclipse Modelling Framework (EMF) and Graphical Modelling Framework (GMF) [EFc].

The foundations of DSM are meta-modelling and model transformation. Most of the tools listed above have integrated or add-on support for them. Features, flexibility, interfaces and implementations vary from one tool to the other. Though we do not intend to provide an exhaustive, feature-based comparison and review of all existing DSM tools, our knowledge of them allows us to extract a number of shared characteristics which we deem limiting to their usability, accessibility and/or scientific appeal. First and foremost, they each require the installation of one or more applications on the user's computer. In the case of Eclipse-based tools, this includes the installation of the resource-intensive Eclipse IDE and a non-negligible learning curve. For GME, this restricts the user to the Microsoft Windows operating system. Other limitations are more tool-specific. For instance, as mentioned in Chapter 3, VMTS is the only tool we know of that supports pausing ongoing model transformations. Other tools, such as AToM^3, do not support specifying multiple concrete syntaxes for a given abstract syntax element, which can often be crucial to avoid duplication. Finally, model transformation models and models of their rules are often treated as "special" artifacts. That is, tools provide special means of specifying them outside of the traditional "meta-model and conforming model" workflow. One grave consequence of this is that most tools do not and can not (without significant alterations) support HOTs. In developing AToMPM, a key objective was to produce a modelling and meta-modelling tool that avoided the limitations encountered in existing tools while combining their individual strengths. Feedback from researchers at Honeywell, a diversified technology and manufacturing leader, was also taken into consideration to produce a tool that would be appealing to both academic and industrial DSM experts and practitioners.

The first of the requirements that guided AToMPM's development – not including its implied support for basic modelling, meta-modelling and model transformation tasks – is that it run in popular Web browsers and store user data in the cloud (i.e., on the remote server hosting AToMPM, not on the user's machine). This implies that no installation is required on the user's machine. Merely navigating to a URL that serves the AToMPM client (e.g., *http://orac.cs.mcgill.ca:8124/mpm/*) should suffice to perform the full array of supported DSM activities from anywhere in the world, on any machine. This indeed lowers the threshold and makes AToMPM extremely accessible for anyone from expert developer to curious novice. It also enables otherwise impossible on-the-fly and impromptu demonstrations. Finally, this design choice is both compatible and consistent with the current tendency towards cloud computing that is guiding modern software development in the direction of online services and away from standalone applications.

The next set of requirements pertains to the specification and execution of model transformations. For the former, we focused on supporting model transformations (and HOTs) in a natural and principled manner and making no distinction between rules and their schedules and any other model. For the latter, we targeted continuous and step-by-step rule execution, a debugging mode, and the ability to pause and resume an ongoing model transformation. Due to the central role that model transformations (should) play in DSM, powerful and theoretically sound facilities are bare necessities in a modern DSM tool.

The next requirement pertains to usability. Undoing and redoing changes, as well as copying and pasting model entities are often not or poorly supported in DSM tools we have encountered. The prevalence of such facilities in the vast majority of modern applications makes this omission particularly noticeable and inconvenient during model development. Thus, to enhance usability and reduce overall (repeated) labour, the aforementioned features should all be supported.

Another requirement is that it be possible to associate several concrete syntaxes to a given abstract syntax. Scenarios where this is useful include the specification of construction schematics where European and American units and/or symbols differ. Indeed, creating and maintaining several models that are conceptually and semantically identical and differ only in graphical representation minutia is clearly impractical, non-scalable and error-prone. In practice, it should be possible to render a given abstract model with any one of its associated icon sets (i.e., concrete syntaxes). More generally, this requirement also extends to textual concrete syntaxes.

The following two requirements were formulated by researchers at Honeywell. First, it should be possible to perform version control on models. More specifically, it should be possible to compute (and store) the difference between two models, and to merge parallel branches of a given model. Second, several developers should be able to simultaneously edit the *same* model and be made aware of the others' changes in real-time. This, as opposed to editing their own local copies of a model and merging them back together at a later time. Thus, real-time, distributed, collaborative modelling should be supported. The appeal of such facilities in an industrial, multi-developer context is clear. However, collaboration in particular can also be extremely useful for didactic purposes. Consider, for instance, a scenario where team mates or even an entire class and professor work together to build a model or a meta-model, each from their own collaborating instance of AToMPM.

Finally, the last high-level requirement that guided AToMPM's development is that it be aesthetically pleasing. Although this may seem somewhat superficial, the appeal of a modern look-and-feel, particularly to novices, is undeniable. Furthermore, demonstrating the *future* of software development to an unfriendly and/or sceptic audience is that much more challenging when one's tooling looks and feels like it was developed over a decade ago.

To our knowledge, no existing DSM tool supports all of the listed requirements. The rest of this chapter is structured as follows. In Section 5.2, we give an overview of AToMPM and explore how it (at least partially) satisfies every single one of the above requirements. In Section 5.3, we peer beneath the surface and examine the architecture and implementation of AToMPM from a developer standpoint. Finally, in Section 5.4, we identify our scientific and technical contributions and discuss remaining implementation work.

5.2 Features and Usage

This section describes AToMPM from a user's perspective and explores how it satisfies each of its requirements. Note that more details are provided in AToMPM's User's Manual [Man].

5.2.1 Client

The AToMPM client (i.e., the component with which the user interacts) is shown in Figure 5.1 running in the Google Chrome Web browser [Gooc]. Two *button toolbars*[1] are loaded: the main menu and the transformation controller menu. The former enables basic editing commands such as loading and unloading (button and meta-model) toolbars and models, saving models and undoing and redoing changes. The latter is used to load transformations, control their execution and toggle the transformation debugging mode. More details pertaining to the implementation of button toolbars and of their associated functionality are given in the next section. At the bottom-right of the interface are a username input field and collaboration links. The former enables one to access and manipulate personal data stored within the cloud (i.e., on the machine that serves the AToMPM client). The latter are used to initiate one of two types of collaboration sessions, which we explore in Section 5.2.6. Of primary interest in this section are the facts that the AToMPM client is running in the Google Chrome Web browser[2], is accessed through an appropriate URL, stores user data in the cloud, and the aesthetics of the graphical user interface. Aesthetics are of course a very subjective matter. Therefore, we took inspiration for style and color palette from popular and/or relevant websites, applications and services such as Google [Gooa], Google Docs [Good], Google Chrome and Facebook [Fac].

[1]Two types of toolbars exist within AToMPM. *Button toolbars* are akin to traditional toolbars. They provide a collection of buttons that perform arbitrarily complex functions. *Meta-model toolbars*, on the other hand, display a meta-model's instantiable types and enable the creation of model entities.

[2]At this time, AToMPM has only been tested on Webkit- (e.g., Apple Safari [Appb], Google Chrome) and Gecko-powered (e.g., Mozilla Firefox [Mozb]) browsers [Moza, web].

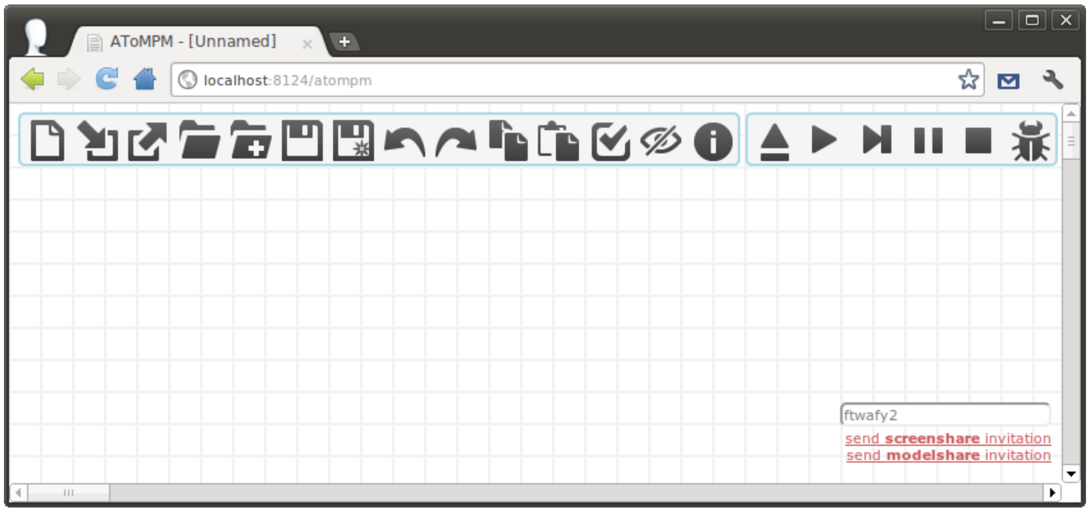


Figure 5.1: The AToMPM client, with the main menu and transformation controller menu loaded, running in the Google Chrome Web browser.

5.2.2 Modelling and Meta-Modelling

AToMPM is first and foremost a modelling and meta-modelling tool. As such, it defines a meta-meta-model to which all meta-models defined within AToMPM conform, and that conforms to itself. This meta-meta-model is the meta-model of the Entity Relationship Model, shown in Figure 5.2. For convenience, more elaborate, alternative meta-meta-models are also provided within AToMPM, such as a simplified version of UML Class Diagrams. The template-based approach presented in Chapter 4 can also be seen as *somewhat* of a meta-meta-model. While we have shown that traditional meta-models can be automatically synthesized from ST models, the relationship between ST models and synthesized meta-models is arguably not one of instantiation. Furthermore, the bootstrapped architecture of AToMPM restricts meta-meta-model status to self-conforming meta-models, which ST meta-models are not.

Meta-models defined in AToMPM (e.g., as Entity Relationship or Class Diagram models) are automatically compiled and can be loaded as *meta-model toolbars*. Before doing so, however, one must associate concrete syntax to abstract syntax concepts (i.e., to concepts defined by the meta-model) such that these may be rendered. In AToM^3, `Class` and `Entity` constructs that make up meta-models have an editable `Graphical_Appearance` attribute. Editing this attribute launches a built-in drawing tool that one can use to describe the relevant abstract syntax concept's concrete representation. This approach is flawed in a number of ways. First, concrete syntax has no place as an abstract syntax attribute. Second, supporting several representations per concept would further pollute abstract syntax representations. Third and worst, defining concrete syntax is treated as a "special" activity that sits outside of the "meta-model and conforming model" workflow. Amongst other things, this implies that the meta-modelling infrastructure that supports that workflow can not be reused to support the specification of concrete syntax. A clear consequence of this is the existence of the aforementioned drawing tool. In AToMPM, ensuring that all activities adhere to the "meta-model and conforming model" workflow is a top priority. To this end, in AToMPM, concrete syntax specifications are full-fledged models, disjoint from abstract syntax specifications. The meta-model to which they conform is shown in Figure 5.3. Essentially, `Icons` contain an arbitrary number of graphical constructs (e.g., `Circle`, `Text`) optionally inter-related to each other via layout constraints. For example, the rectangle describing a class' icon should always be large enough to fully contain its name and list of attributes. Furthermore, all graphical constructs have `style`, `mapper` and `parser` attributes. The `style` attribute provides control over color, font, opacity, etc. The two others respectively provide means to reflect abstract syntax values within the concrete syntax and vice versa. Thus, concrete syntax models are essentially a collection of enhanced attributed drawings contained within appropriately labelled `Icons`. These labels enable the resolution of correspondences between an icon and its associated abstract syntax concept. Figure 5.4 shows an example meta-model and *two* associated concrete syntax specifications. Any non-zero[3] number of concrete syntax specifications are supported, and modellers can easily cycle between them to alter the appearance of loaded models. Last but not least, meta-model actions and constraints are supported (e.g., to validate a model, to react to user input) and make use of an API that exposes modelling constructs and their attributes. Further details are provided in the AToMPM User's Manual.

5.2.3 Usability Enhancements

AToMPM robustly supports undoing and redoing changes and copying and pasting model entities. For the former, a critical point is that within a meta-modelling context, a single change may trigger several changes (which may in turn trigger other changes and so on and so forth). For instance, the post-editing action of one entity can cause attributes of connected entities to be altered. This is taken into account in AToMPM which provides robust facilities to atomically undo and redo changes *and* their side-effects. As for the copying and pasting of entities, they present two key concerns. First, on a more technical note, entities can be copied from one instance of AToMPM and pasted into that same instance or into any other (running on the same machine). Second, on a more theoretical note, it should arguably not be possible to create impossible models (i.e., models that do not conform to their meta-model(s)'s constraints) via the pasting of entities. For instance, if a meta-model constraint limits the number of instances of a specific concept to n, it should not be possible to exceed that number by copying m instances and pasting them repeatedly. In other words, the pasting of entities should be subject to the same conformance restrictions and consequences as the manual specification of the said entities. This is the implemented behaviour in AToMPM.

[3]Note that we are currently exploring "headless" (or "scripted") modelling scenarios in which concrete syntax representations could be omitted.

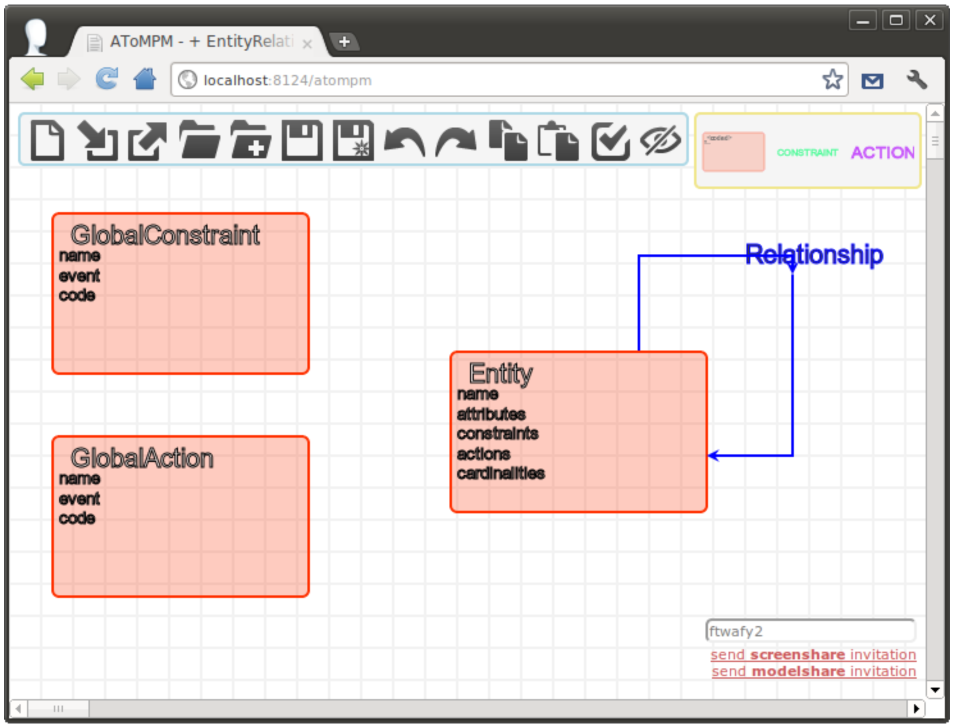


Figure 5.2: The meta-meta-model at the core of AToMPM's meta-modelling kernel.

5.2.4 Model Transformation

AToMPM's implementation of model transformation specification also adheres to the "meta-model and conforming model" workflow. This again contrasts with other tools and with its predecessor, AToM^3, where "special" editors are provided for the editing of rule schedules and of rules themselves. As demonstrated throughout this thesis, rules and their schedules in AToMPM are ordinary models that conform to ordinary meta-models. These are shown in Figures 5.5 and 5.6 and are described in detail in the AToMPM User's Manual. For the purpose of the discussion at hand, however, their usage and the consequences of their existence are of greater interest. In their usage lies what is possibly AToMPM's most significant scientific contribution, namely, the first full implementation of Khüne et al.'s *RAM* (Relaxation, Augmentation, Modification) process [KMS+10, Syr11] within a DSM tool. The RAM process consists of altering a meta-model to allow the specification of partial models in rule patterns. For instance, while the Statechart meta-model may require that at least one starting state exist, a LHS pattern describing two connected states should clearly not be invalid if neither of them is a starting state. The RAM process is divided into three steps. First, validity and multiplicity constraints are *relaxed*. In other words, any constraints on minimum numbers of instances or connections are removed. As for validity constraints, an often acceptable approach is to ignore them all, while supporting manual intervention to select those that should be kept, if any. The final relaxation step is to "concretize" abstract classes, i.e., provide default concrete syntax for them and allow their instantiation. Second, abstract

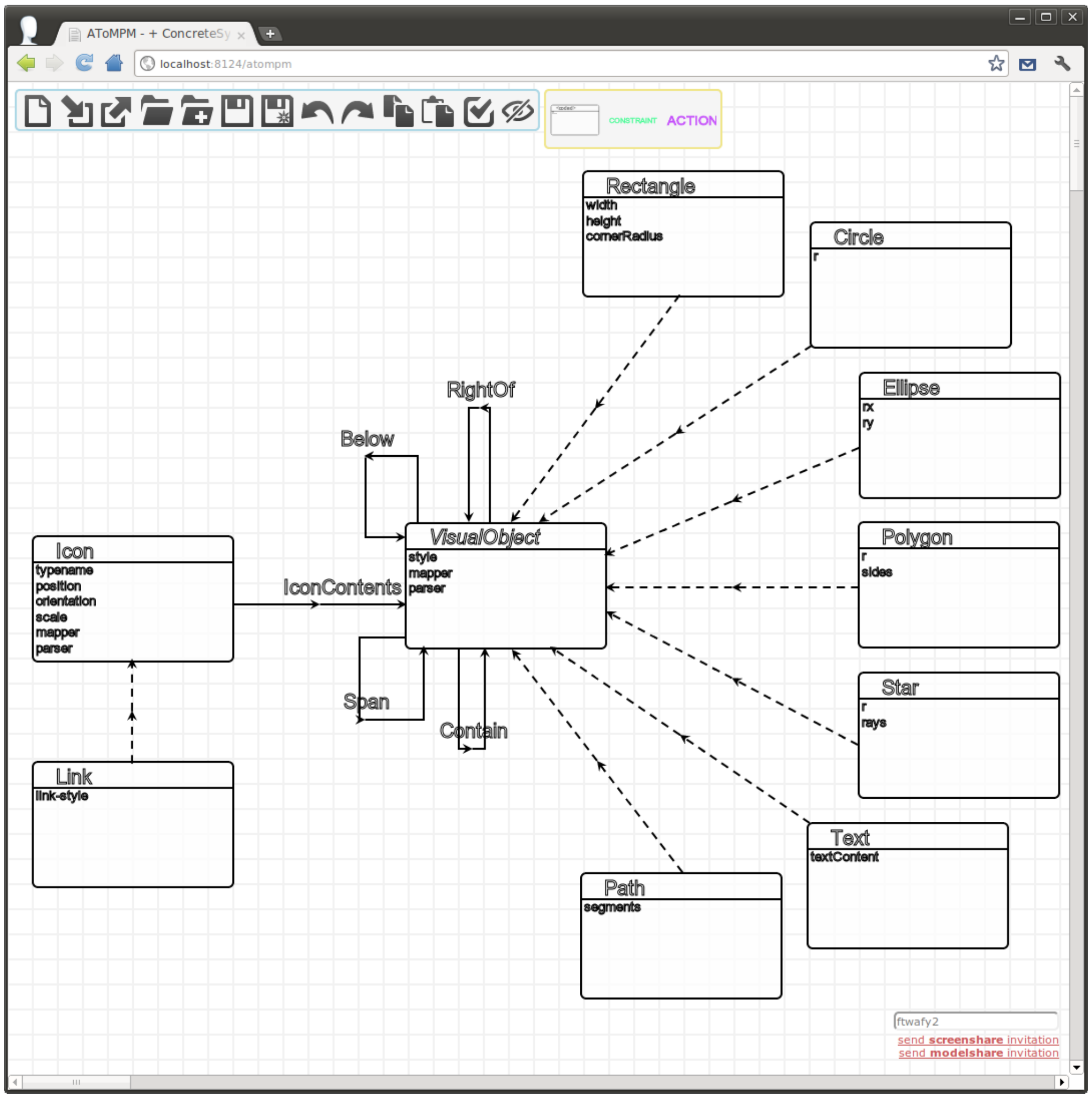


Figure 5.3: The meta-model of concrete syntax representations in AToMPM.

and concrete syntax elements are *augmented* with transformation pertinent attributes, such as pattern labels and subtype matching flags. These added attributes are of paramount importance to the expressiveness and proper functioning of the underlying transformation execution engine. Third and last, attribute types are *modified* to account for the fact that transformation engines expect their values to be condition or action code. Indeed, during the matching process, the transformation engine evaluates conditions that determine if the attribute values of candidate matches in the source model are acceptable (e.g., does `name` start with “red”, is `nbTokens` greater than five). In practice, these conditions are specified in the attribute values of entities that form LHS and NAC patterns. Analogous reasoning

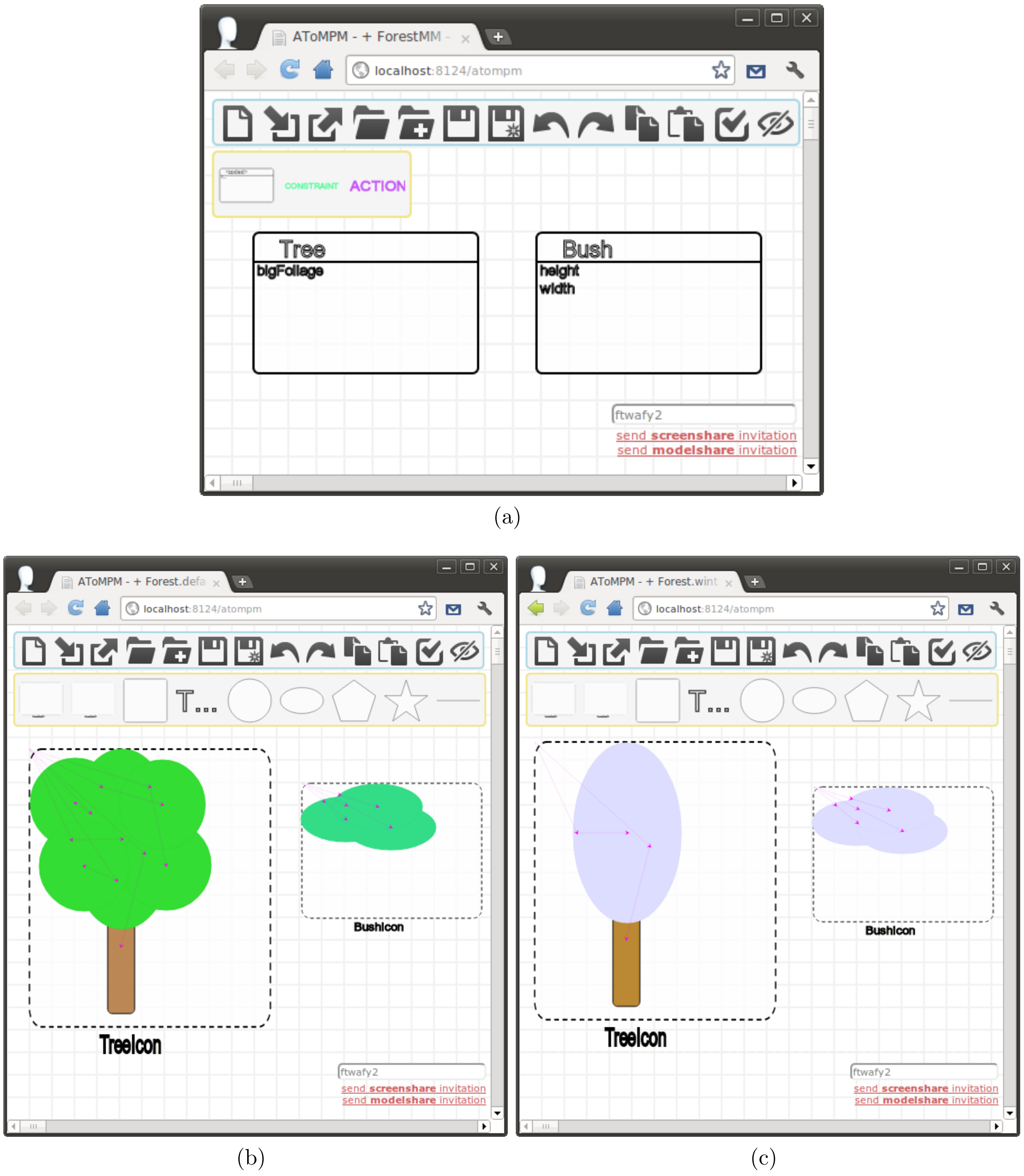


Figure 5.4: (a) A very simple meta-model for a forest DSML, and (b-c) two associated concrete syntax specifications.

applies to action code and RHS pattern entity attributes. Once the RAM process has completed, the resulting *RAMified* meta-model is output and stored. In AToMPM, these are called *pattern meta-models*, and only constructs conforming to pattern meta-models can be contained within rule patterns. As for the consequences of meta-modelling transformations and their rules, beyond providing a consistent and principled editing environment devoid of "special" scenarios that complicate usage and implementation, it enables HOTs. Specifying HOT rules in AToMPM is not only possible, it is in fact no different than specifying regular rules. To match one or more rule in another rule R's pre-condition pattern(s) or to produce one or more rule in R's post-condition pattern, one needs only populate R's patterns with entities from the RAMified *Transformation Rule* meta-model (shown in its non-RAMified form in Figure 5.5). Similarly, entire transformations can be matched and produced by populating R's patterns with entities from the RAMified *Transformation* meta-model (shown in its non-RAMified form in Figure 5.6, and in its RAMified form in Figure 5.7).

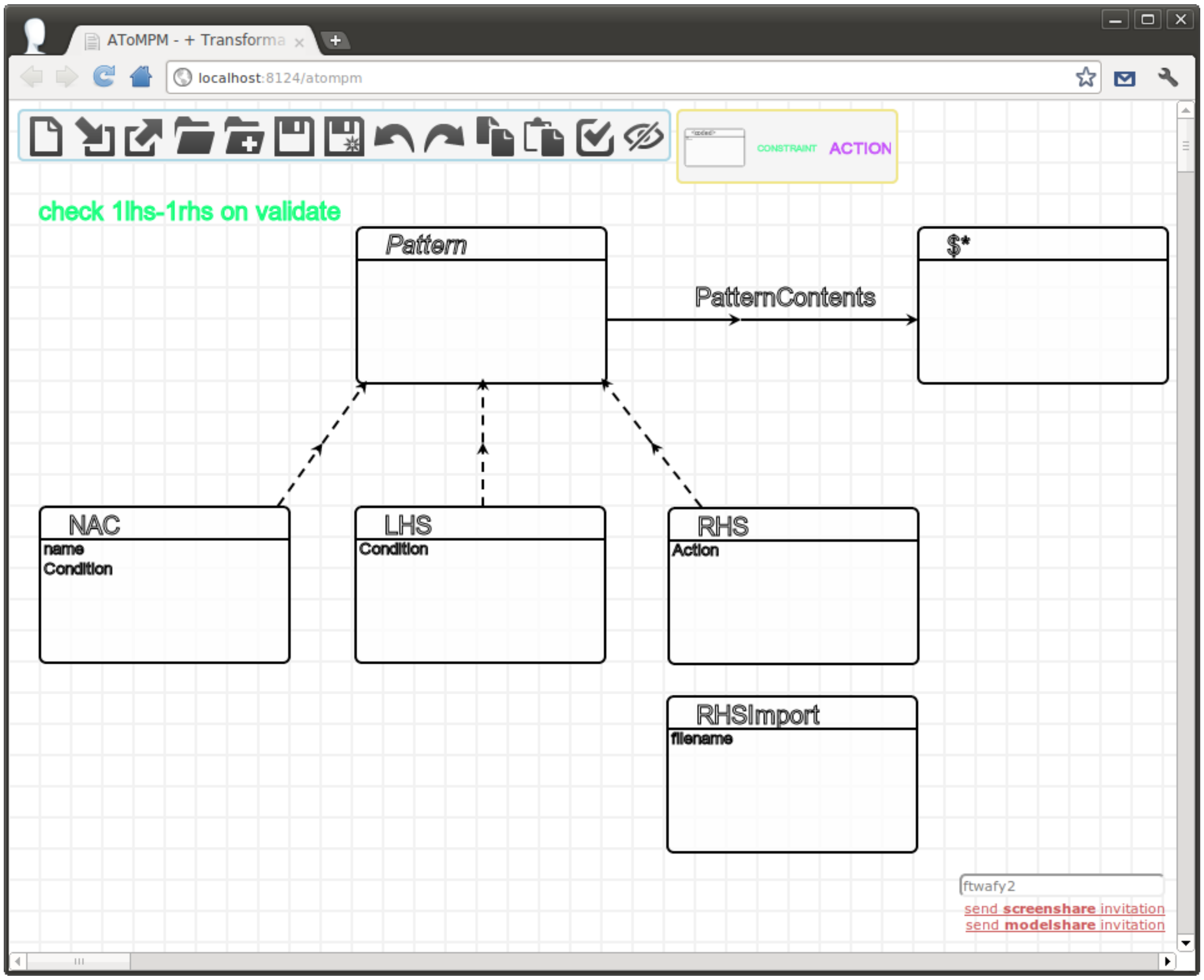


Figure 5.5: The *Transformation Rule* meta-model.

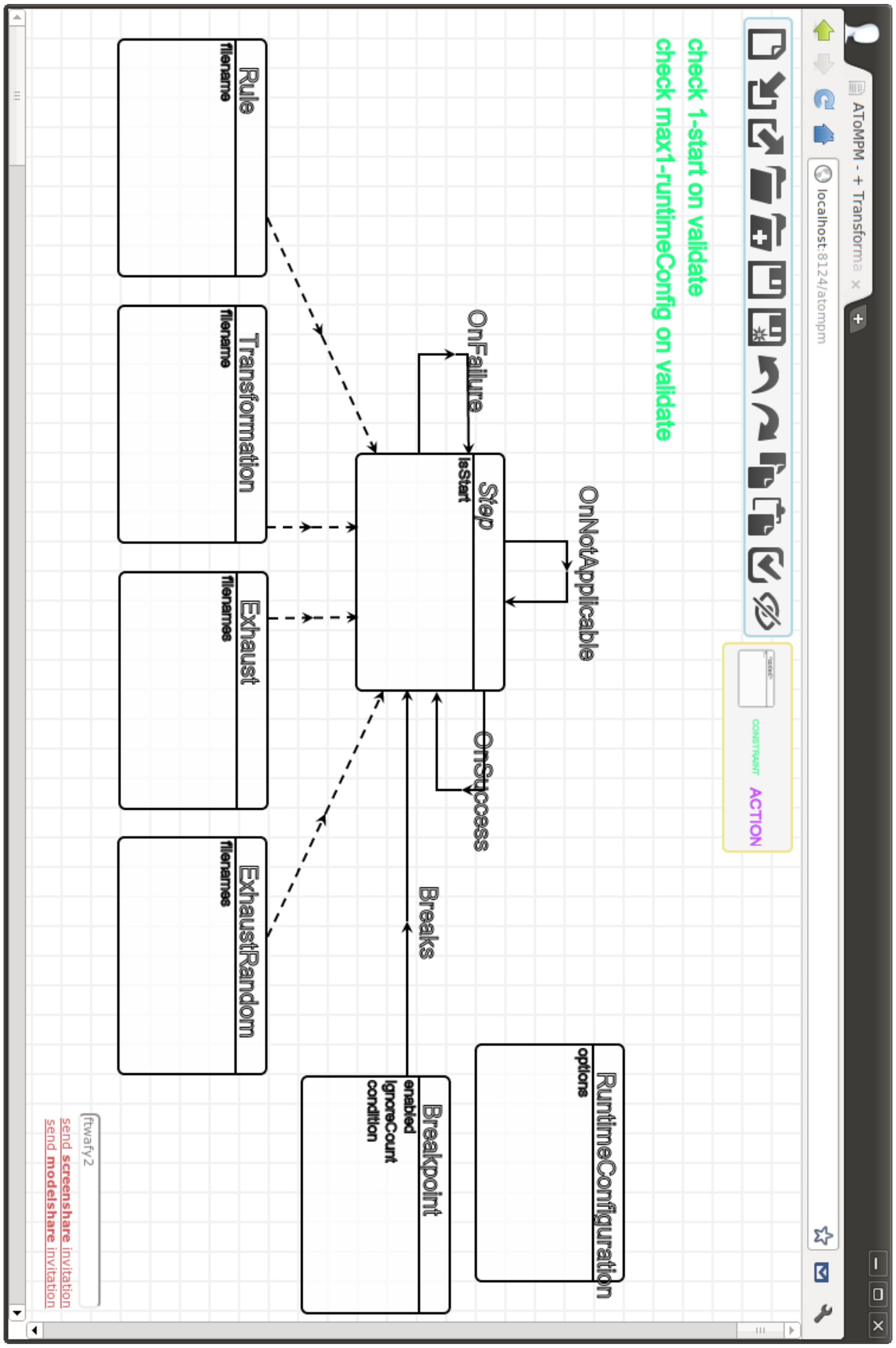


Figure 5.6: The *Transformation* meta-model.

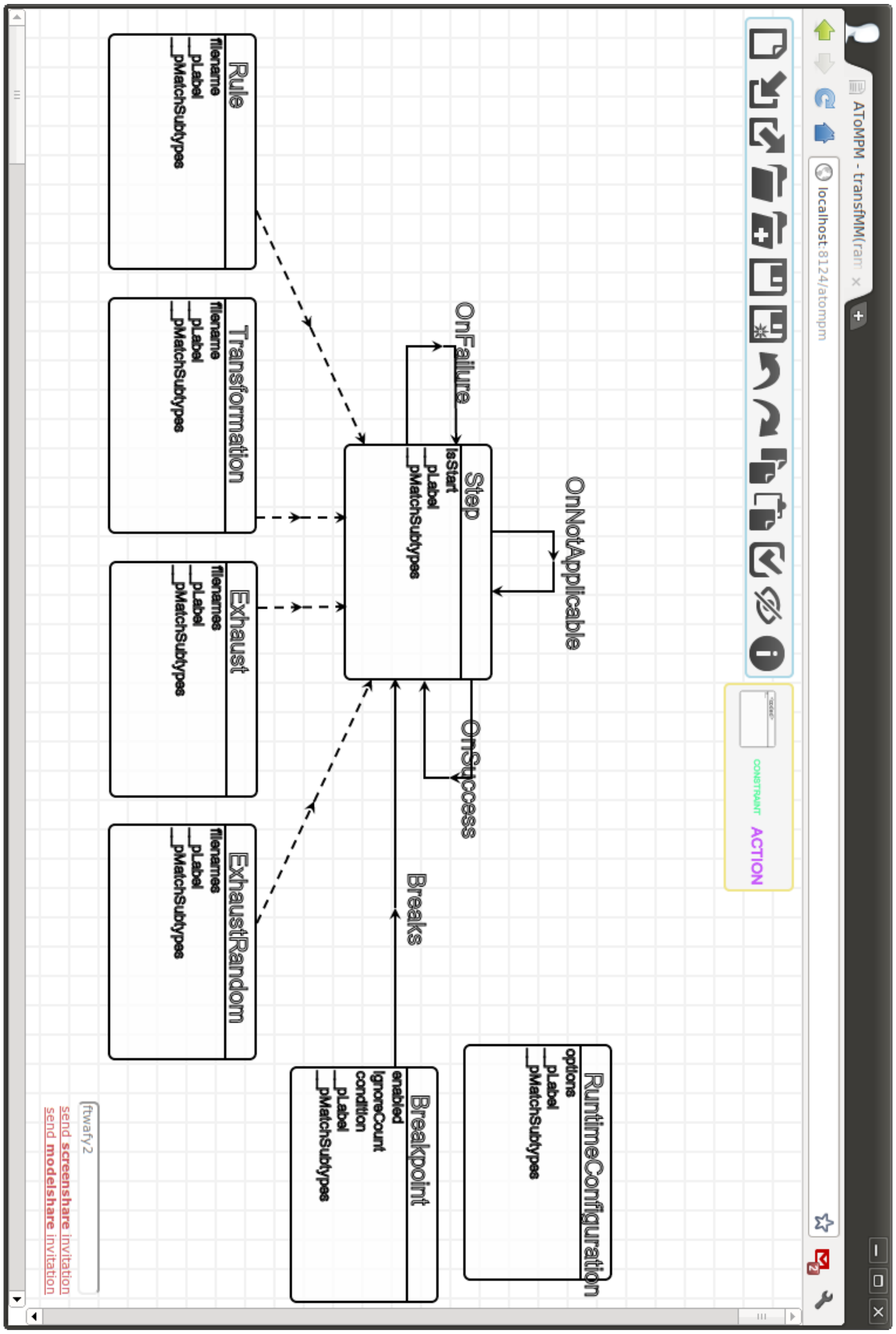


Figure 5.7: The RAMified *Transformation* meta-model.

As of now, we do not know of any tool that has more elaborate transformation execution facilities than AToMPM. Step-by-step and continuous execution modes, "triple-outcome" (i.e., success, not applicable, failure) rule and transformation control flow, pausing of ongoing transformations and a debugging mode are all supported. Furthermore, breakpointing and full stepping functionality (recall that only "step into" functionality is currently available) will soon be included.

5.2.5 Model Versioning

Model versioning has not yet been fully implemented within AToMPM. However, AToMPM is built on a journalling system (i.e., all operations are logged) and as such, numerous complex issues related to model versioning disappear. More specifically, the difference between two subsequent versions of an AToMPM model is entirely captured by the series of logged operations (also known as the *edit script* [AP03]) that begins after the first version was committed and ends after the second version was finalized. Thus, model versioning in a single-development branch environment can be very simply (and naively) achieved by storing a model's first version and all subsequent operation logs. Restoring a particular version then reduces to applying relevant logs to the first version. Optimizations can also be applied to reduce the size of operation logs. A trivial example is shown below:

<u>Original operation log</u>:

1. Set attribute a of instance #4[4] to 7;
2. Set attribute b of instance #5 to 8;
3. Set attribute a of instance #4 to 5;
4. Delete instance #5.

<u>Optimized operation log</u>:

1. Set attribute a of instance #4 to 5;
2. Delete instance #5.

A more complex issue is the question of differencing models that aren't subsequent versions of the same model, such as parallel branches of a model. This problem is closely related to the question of optimizing operation logs. A combination of semi-automatic inference rules and optimizing reductions are required to determine which operations can produce one model from the other. Merging model branches is an active research field in its own right and is beyond the scope of this thesis. Nevertheless, existing results from the model differencing and versioning community will be applied to enhance AToMPM's versioning (and specifically development branch merging) capabilities.

[4]In AToMPM, instance identifiers are preserved during serialization.

5.2.6 Collaborative Editing

AToMPM supports two modes of real-time distributed collaboration, namely, *screenshare* and *modelshare*. In the former, all collaborating developers share the same concrete and abstract syntax. This implies that if one developer moves an entity or cycles to another concrete syntax representation, the change will be replicated for all collaborators. The screenshare mode is thus especially useful for didactic purposes. In contrast, in the modelshare mode, only abstract syntax is shared. This means that all collaborators can have distinct concrete syntax representations and distinct layouts (provided layout and abstract syntax are not intricately related), and are only affected by others' abstract syntax changes (e.g., modifying attribute values). This is certainly a more flexible mode where individual modeller tastes (e.g., for model layout) are not penalized by collaboration. Note that the availability of the modelshare mode is a direct consequence of the principled separation of abstract and concrete syntax discussed earlier.

5.3 Architecture

This section examines the inner workings of AToMPM with a particular focus on the implementation of the critical features discussed in previous sections. AToMPM is built according to the client-server model. Its high-level components and their associations are shown in Figure 5.8. Each component and its associations are described below.

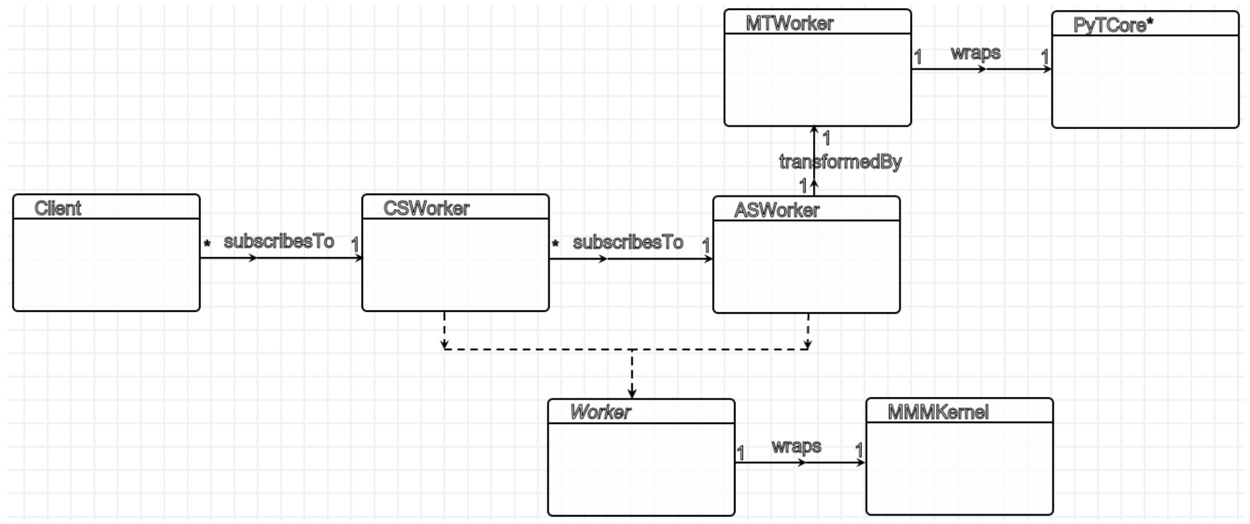


Figure 5.8: The high-level architecture of AToMPM.

5.3.1 Client

The part of AToMPM we refer to as its *client* is a JavaScript application that runs in Webkit- and Gecko-powered Web browsers. A side-effect of the strict application of the "meta-model and conforming model" workflow to the requirements of AToMPM is that its client is relatively simple, supporting barely more than what is necessary to enable the said workflow. The client's capabilities are essentially: sending requests to its associated *CSWorker*, handling its responses, and implementing GUI behaviour.

During client initialization, a WebSocket [W3C] request triggers the spawning of a new CSWorker (or the connection to an existing one in screenshare mode) which in turn triggers the spawning of a new *ASWorker* (or the connection to an existing one in either collaboration mode). The main advantage of establishing WebSocket connections between AToMPM's components is that WebSockets enable elegant and simple means for downstream components to send unsolicited data upstream. For instance, in the traditional HTTP request and response model, a (downstream) server can respond to requests from an (upstream) client, but it can not send it any additional data until the next request. Other means than WebSockets exist to enable this, such as Long Polling [Wika], but they are mostly creative adaptations of unsuited facilities, which the recently introduced WebSockets are rendering obsolete. Upon completion of client initialization, the new client is said to be "subscribed" to its CSWorker which in turn is said to be subscribed to its ASWorker. All future requests from client to CSWorker, all of which pertain to modelling and meta-modelling activities (e.g., loading a new meta-model, creating a new entity, moving an entity), are HTTP requests. The vast majority of these are answered by virtually nothing more than a success or error HTTP status code that reflects the CSWorker's proper reception of the request. Only when the CSWorker has processed the request, which often involves forwarding it to its associated ASWorker and asynchronously waiting for its feedback, is a *changelog* encoding relevant concrete syntax changes (as a list of primitive operations) produced and sent (via the WebSocket connection) to subscribed clients. This shows two aspects of AToMPM's architecture. First, most actions the user can carry out (e.g., loading a new meta-model, creating a new entity, moving an entity) have no immediate impact within the client. Indeed, when a user drags an entity on the canvas, all that happens is that an HTTP request describing the desired change is sent to the CSWorker. If the requested operation is legal (i.e., does not violate any meta-model constraints), the resulting CSWorker changelog will cause the client to actually move the entity. Second and most importantly, all collaborators are equals. Given that client behaviour is "changelog-driven", the instigating user (and his client) is treated no differently that any other. Although these design choices lead to a perceptible lag on very high-latency networks, they considerably reduce client model editing implementation complexity. All (possibly complex and/or illegal) changes to a model are computed beyond the client, with the client only required to properly understand and reflect incoming changelogs. This task involves little more than updating rendered concrete syntax elements.

The main purpose of the AToMPM client is to display and enable interaction with models. The former task is captured by its ability to translate concrete syntax representations (i.e., compositions of Figure 5.3's `VisualObjects`) into Scalable Vector Graphics (SVG) drawings.

This translation is actually quite straightforward due to the near one-to-one equivalence between supported `VisualObject`s and existing SVG primitives. Major advantages of SVG over basic browser raster graphics facilities are that the quality of SVG drawings does not decrease during scaling, that complex geometric transformations (e.g., rotations) can be applied to them very easily, and that they can be edited outside of the AToMPM client with dedicated SVG manipulation tools. As for actual interaction with models, the client's GUI behaviour is entirely modelled using Statechart models, i.e., high-level models rather than tangled and complex code describe the effects of clicks, motions, keystrokes and more. The advantages of using Statechart models to capture complex, reactive behaviour have been studied extensively in past work [BV09, DBV09]. They include dramatically reduced development times and eased maintenance. Figure 5.9 shows the Statechart model that describes the behaviour of the client canvas. It shows that the canvas can be in a number of different states where it reacts to various events differently. Not shown are `Transition` guards (i.e., conditions that enable or disable them), `Transition` actions and `State` entry actions. A textual summary of the Statechart from Figure 5.9 (*including* its guards and actions), targeted at end-users, is presented in Table 5.1.

Last but not least, a note on button toolbars. Despite the fact that they arguably sit outside of concrete modelling activities, their specification also adheres to the "meta-model and conforming model" workflow. In AToMPM, button toolbars are instance models of the *Buttons* formalism. The model that defines the main menu toolbar is shown in Figure 5.10a. The x, y position of each `Button` instance defines the corresponding button's position on the loaded toolbar. The buttons' tooltips, their names (used to bind them to an icon) and the code that implements their behaviour are specified as `Button` abstract syntax properties, as shown in Figure 5.10b. The behaviour code makes use of AToMPM's client API, which is thoroughly detailed its User's Manual.

5.3.2 MMMKernel

The modelling and meta-modelling kernel, or *MMMKernel*, is at the core of AToMPM. Each model lives within an MMMKernel that performs all modifications on it, with other components merely (re-)routing requests to it and reacting to its output. Virtually every operation a user may carry out (from the client) will eventually reach an instance of the MMMKernel, where appropriate steps will be performed and changelogs produced. For instance, consider the sequence of events that begins when a user requests that a new instance of a concept C be created, illustrated in Figure 5.11. First, an appropriate HTTP request is sent to the client's associated CSWorker which immediately responds with an HTTP success status code indicating the request has been accepted and is being processed. Then, the CSWorker adjusts the HTTP request and sends it to its associated ASWorker. At this point, the ASWorker makes an appropriately parametrized function call to its associated MMMKernel instance requesting a new C. The MMMKernel will then verify all pre-creation constraints, execute all pre-creation actions, create a new instance given the provided parameters, execute all post-creation actions and verify all post-creation constraints. If any of these steps

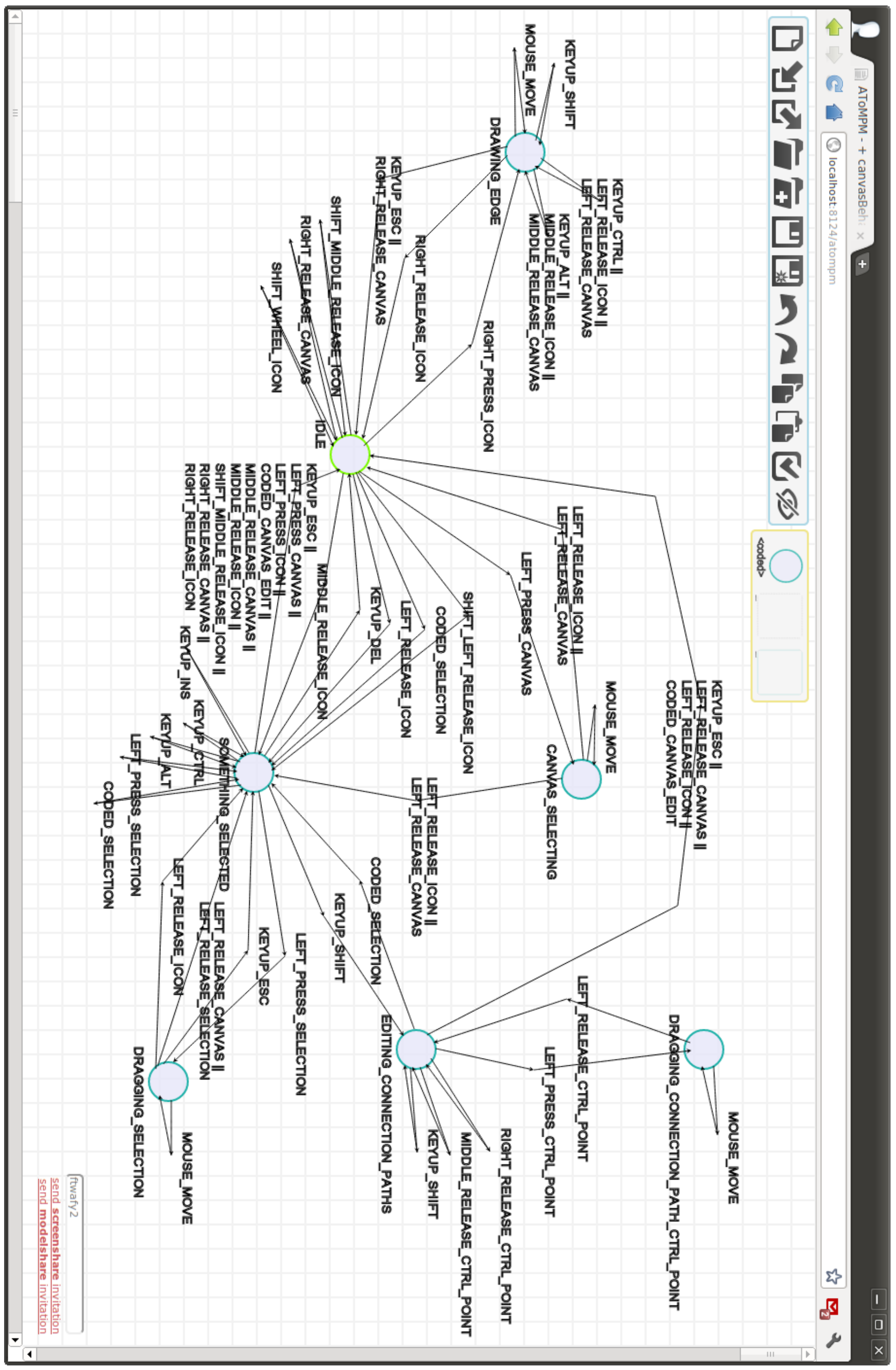


Figure 5.9: The Statechart model that describes the behaviour of the AToMPM client canvas.

Action	Shortcut(s)
Choose an entity type to create	Left-click on desired type from a loaded formalism toolbar.
Create an entity	Right-click anywhere on the Canvas.
Select an entity	Left-click any entity.
Select one or more entity	Left-press anywhere on Canvas, drag selection box around desired entity or entities and release.
Connect entities	Right-press an entity, drag to-be edge to target entity and release.
Edit icon text	CTRL-SHIFT-Middle-click any text from any icon on the Canvas (this will display a very simple text editor).

(a) When in the **IDLE** (or default) state.

Action	Shortcut(s)
Unselect selection	Right-/Left-/Middle-click anywhere on the Canvas, or click ESC.
Move selection	Left-press selection, drag preview overlay to desired position and release.
Delete selection	Click DELETE.
Edit first entity in selection	Middle-click selection, or click INSERT (this will display the abstract attribute editor).
Enter geometry editing mode	Click CTRL (this will display geometry controls).
Enter edge editing mode	Click SHIFT (this will display editable edge control points).

(b) When in the **SOMETHING SELECTED** state (i.e., when one or more entity is selected).

Action	Shortcut(s)
Make current line horizontal/vertical	Click SHIFT.
Create control point	Left-click anywhere, or click CTRL.
Delete last control point	Middle-click anywhere, or click ALT.
Cancel current edge	Left-release anywhere on the Canvas.

(c) When in the **DRAWING EDGE** state (i.e., when dragging to-be edge from source to target entities).

Table 5.1: Actions available in each of the client canvas' states and their corresponding shortcut(s) *(continued)*.

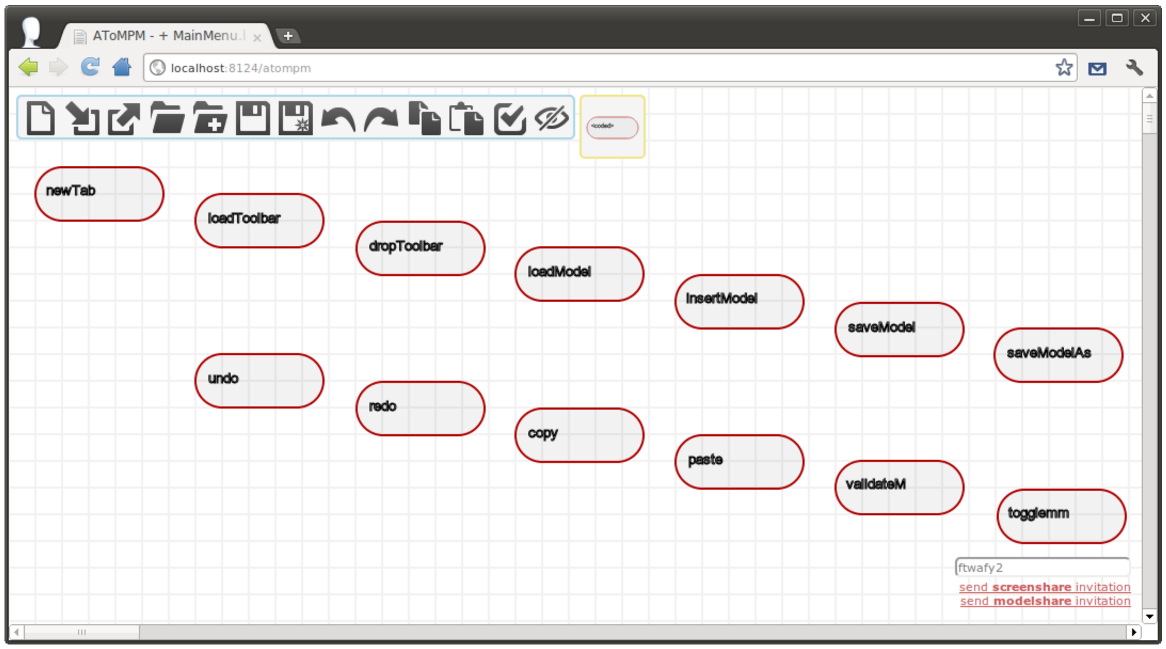
Action	**Shortcut(s)**
Move control point	Left-press any control point, drag it to desired position and release.
Vertically/Horizontally align control point to previous control point	Left-click any control point and click SHIFT.
Clone control point	Right-click any control point.
Delete control point	Middle-click any control point (extremities and the central control point are not removable).

(d) When in the **EDGE EDITING** state.

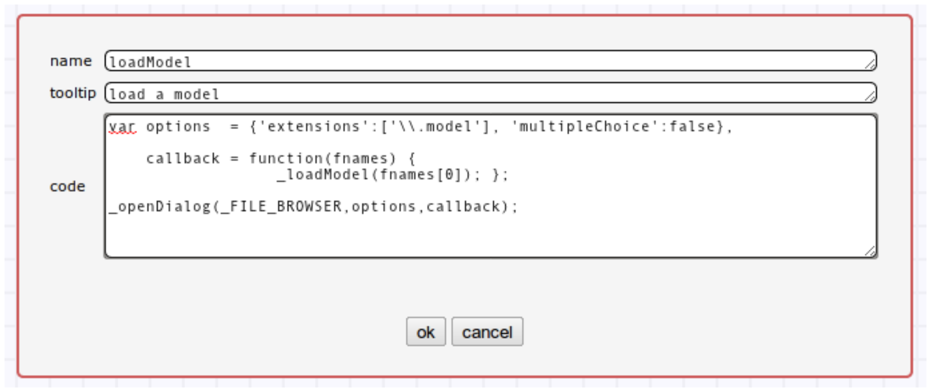
Action	**Shortcut(s)**
Scale	Mouse-wheel up/down on scale icon until preview overlay reaches desired shape.
Scale vertically only	Mouse-wheel up/down on vertical scale icon until preview overlay reaches desired shape.
Scale horizontally only	Mouse-wheel up/down on horizontal scale icon until preview overlay reaches desired shape.
Rotate	Mouse-wheel up/down on rotation icon until preview overlay reaches desired shape.
Cancel changes	Right-/Left-/Middle-click anywhere on the Canvas, or click ESC.
Confirm changes	Left-click confirmation icon.

(e) When in the **GEOMETRY EDITING** state.

Table 5.1: Actions available in each of the client canvas' states and their corresponding shortcut(s).



(a) A model of the main menu toolbar.



name: loadModel

tooltip: load a model

code:

```
var options  = {'extensions':['\\.model'], 'multipleChoice':false},

    callback = function(fnames) {
                    _loadModel(fnames[0]); };

_openDialog(_FILE_BROWSER,options,callback);
```

(b) The property dialog of the main menu *loadModel* `Button`.

Figure 5.10: Button toolbars in AToMPM.

fail, the MMMKernel will return an error and rollback any changes. Otherwise, it will return a changelog detailing the modifications the model underwent. In both cases, the ASWorker will notify the requesting CSWorker of failure or success (via a standard HTTP response), wrap the returned error or changelog in communication protocol boilerplate[5] and broadcast it to all subscribed CSWorkers (via their WebSocket connections). These will in turn react by making an appropriate function call to their own MMMKernel instance requesting the creation of a new *CIcon* instance. The resulting changelog will then itself be wrapped and broadcast to subscribed clients, including the one that instigated the request. Regardless of

[5]Such boilerplate includes sequence numbers that provide robustness against various network issues that can lead to changelogs being received out of order by subscribers.

Logged Data	**Operation Description**
`MKNODE:`$id, node$	Created a node with given identifier and data.
`RMNODE:`$id, node$	Deleted a node with given identifier and data.
`MKEDGE:`id, id	Created an edge between given nodes.
`RMEDGE:`id, id	Deleted an edge between given nodes.
`CHATTR:`$id, attr, new\_val, old\_val$	Updated given attribute of given node from *old_val* to *new_val*.
`LOADMM:`$name, mm$	Loaded given meta-model.
`DUMPMM:`$name, mm$	Unloaded given meta-model.
`RESETM:`$new\_name, new\_model,$ $old\_name, old\_model, insert$	Loaded given model preserving or overwriting current model.

Table 5.2: The set of primitive operations supported by the MMMKernel.

the requested operation, the above sequence of steps or a close variation (e.g., requests that do not involve abstract syntax modifications are not forwarded to the ASWorker) of it hold true. At the lowest level, pre- and post- meta-modelling constraints and actions are verified and performed before and after each model modification. No other component of AToMPM is aware of these constraints and actions or of how to manipulate models. Conversely, the MMMKernel is completely oblivious of the rest of AToMPM and even of anything outside of its internal state (e.g., it never performs any disk I/O, it is never exposed to communication protocol details). It knows no more and no less than how to perform model CRUD (Create, Read, Update, Delete) operations within a meta-modelling context. The MMMKernel is thus a critical yet relatively simple component.

Finally, the changelogs returned by MMMKernel operations reflect the journalling system that AToMPM is built on. All model modifications can be reduced to a small number of primitive invertible operations, described in Table 5.2. In practice, the primitive operations that constitute each model modification are logged in a journal that sits at the heart of the MMMKernel. Subsets of this journal form the returned changelogs while the entire journal will eventually serve to add model versioning capabilities to AToMPM. The fact that all primitive operations are invertible is also of critical importance for another requirement: it is the key enabler for the undoing and redoing of changes. Undoing a change simply involves inverting each of its constituting operations.

5.3.3 CSWorker and ASWorker

Each *Worker* encapsulates exactly one MMMKernel, and by extension, exactly one model. That it to say that CRUD operations on a model must go through its Worker. Concrete syntax workers, or CSWorkers, encapsulate concrete syntax models. Abstract syntax workers, or ASWorkers, encapsulate abstract syntax models. In practice, each CSWorker and ASWorker is a distinct process, neither of which needs to be running on the same machine

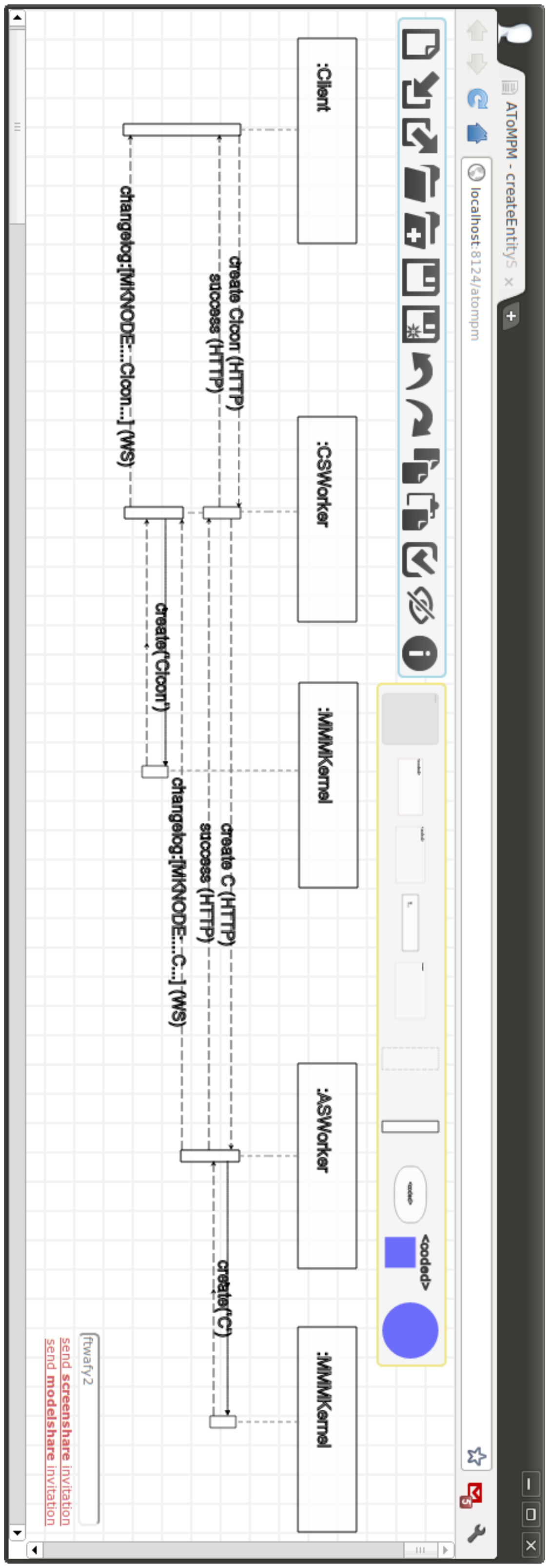


Figure 5.11: A Sequence Diagram [Obj09] describing the creation of an instance of type C.

(or on the same machine as the client). Important interactions between client, CSWorker, ASWorker and MMMKernels have already been discussed. Here, we focus on the distinction between CSWorkers and ASWorkers.

From the user's perspective, a model is the transparent union of an abstract and a concrete component. The fact that each of these components is in reality a model in itself is entirely hidden. In practice, the abstract syntax model conforms to a user-defined meta-model and captures abstract instances, their attributes and connections between them. The concrete syntax model conforms to a meta-model synthesized from a concrete syntax specification (such as those shown in Figure 5.4) and captures icons (that represent abstract instances) and their attributes, namely, position, size, orientation and style. Each abstract syntax model has one or more associated concrete syntax model (each belonging to a subscribed CSWorker). The need for a distinction between concrete and abstract models and workers arises from the fact that an abstract syntax model is entirely disjoint from its concrete syntax representation(s). Indeed, if concrete syntax were to be stored as (hidden or special) abstract arguments, as was the case in AToM^3, there would be no need for CSWorkers. The negative consequences of such a design choice with respect to AToMPM's requirements were discussed in the previous section.

The separation of abstract and concrete syntax concerns also has a significant impact on request handling and propagation. When a user-requested operation is relevant to the abstract syntax model, it is propagated to the appropriate ASWorker and concrete syntax impacts, if any, are carried out only as a side-effect of reported abstract syntax changes. This scenario was described in the earlier example on instance creation. For changes that only concern the concrete syntax, however, such as the rotation of an icon, requests are not propagated to the ASWorker and are instead entirely handled by the CSWorker (and its MMMKernel). In practice, requests that are seemingly only relevant to the concrete syntax model may impact abstract syntax. For instance, in a formalism that describes city maps, constructs' abstract attributes should arguably include and reflect their associated icon's position. Thus, moving icons should trigger appropriate abstract syntax attribute updates. AToMPM provides means for specifying how concrete syntax variables map onto abstract syntax variables (via the `parser` attribute of `VisualObjects`), as well as appropriate internal infrastructure for supporting such deferred abstract syntax modifications.

From the above discussion, one of the properties of AToMPM's architecture emerges, namely, the dual application of the Model/View/Controller design pattern [Bur87] that promotes modularity and decoupling by isolating system logic (Model) from user interface details (View) via an intermediate translator (Controller). On the one hand, the <client/Worker/MMMKernel> hierarchy naturally reflects the said pattern. However, looking further into AToMPM model representation reveals that the <client canvas SVG/concrete syntax model/abstract syntax model> hierarchy does as well. The dual application of this three-tiered architectural paradigm is made especially explicit by the numerous translation and propagation steps that characterize AToMPM's responses to user requests. From a developer's perspective, while these additional steps do entail a slight performance penalty, their advantages in terms of eased development, maintainability, extensibility and (most of all) accessibility significantly outweigh the costs, as is often the case when the Model/View/Controller design pattern is applied.

5.3.4 MTWorker and PyTCore*

The final two components from Figure 5.8, namely, *MTWorker* and *PyTCore**, capture AToMPM's model transformation engine. Each instance of the MTWorker process is associated to exactly one ASWorker and is responsible for carrying out model transformations on its encapsulated model. The MTWorker component is in fact little more than a wrapper around PyTCore*, an enhanced and journalling-based adaptation of Syriani's Py-T-Core [Syr11]. PyTCore* (and Py-T-Core for that matter) is a Python implementation of Syriani's T-Core[6]. For our purposes, it is sufficient to describe it as a means of applying rules to source models. In other words, PyTCore* can be used to identify LHS patterns that satisfy certain conditions within a model, to ensure that they do not satisfy any NACs, and to transform the matched patterns into RHS patterns. What is needed then to use PyTCore* from AToMPM are means to compile AToMPM rules and models into the PyTCore* internal format, means to sequence PyTCore* function calls such as to reflect user-specified rule schedules, means to appropriately route and react to user commands (e.g., enable debugging, pause), and means to translate rule effects back into the AToMPM format. All these tasks and more and carried out by MTWorkers. User commands are HTTP requests that are routed from the client through its CSWorker and ASWorker to its MTWorker, which translates them appropriately into the internal context of the executing transformation engine. As for rule schedules, specified as *Transformation* models in AToMPM, these are interpreted on the fly, meaning that their traversal and the choice of the next rule to read, compile and execute are performed one rule at a time, as a result of the application status (e.g., success, failure) of the previous rule. This design choice privileges lowering initialization overhead, which could potentially be very high, over lowering inter-rule processing time, which remains negligible. Finally, the translation of rule effects back to the AToMPM format is captured by an HTTP query from MTWorker to ASWorker that requests that the relevant changes, described by collections of logged PyTCore* primitive operations (as was the case with MMMKernel), be made. On the one hand, this implies that from the ASWorker's perspective, changes requested by the user through the client are indistinguishable from changes requested by the model transformation engine. On the other, it implies that negative feedback from the ASWorker (e.g., due to rule effects violating meta-model constraints) must trigger MTWorker rollbacks.

The choice of PyTCore* as AToMPM's model transformation engine is mostly due to the numerous strengths of T-Core. One such strength involves the ability to specify rules that are more complex than traditional (NAC-)LHS-RHS rules. For instance, rules with nested

[6]For a complete description of T-Core, we refer the reader to Syriani et. al's work in [SV10, Syr11].

LHSs, or several *matchers* in T-Core terminology, can be used to easily match patterns that would otherwise require very complex LHS and NAC condition code. Other advantages involve its flexibility in terms of rule scheduling. Any scheduling expressible via the Python language is valid and supported. In its current state, AToMPM only harnesses a small fraction of PyTCore*'s full potential. Future enhancements to AToMPM's *Transformation* and *TransformationRule* formalisms and to the MTWorker AToMPM-to-PyTCore* compiler will overcome this limitation. In particular, to fully leverage the power of PyTCore*, means of arbitrarily combining and sequencing T-Core primitives (e.g., *matchers*, *iterators*, *rewriters*), rather than atomic rules, should be provided within AToMPM.

5.4 Technical and Scientific Contribution Review and Remaining Work

The work presented in this chapter was motivated by what we deemed to be usability, accessibility and/or scientific limitations common to popular DSM tools. These included the unprincipled "special" handling of concrete syntax and model transformation specification, the lack of portability inherent to standalone applications, limited model transformation execution controls, and the absence of collaboration and versioning facilities. The means by which AToMPM (partially) addresses each of these shortcomings were discussed from the user's and the developer's perspective.

AToMPM's contributions are both technical and scientific. On the one hand, it is the first (and only) DSM tool that runs in a Web browser (and thus requires no local installation), that supports real-time distributed collaborative modelling, that provides fully integrated copy-paste and undo-redo facilities, that supports pausing ongoing transformations and stepping into composite transformations, *and* whose GUI behaviour is modelled via and synthesized from Statechart models. On the other hand, it is the first DSM tool to implement Khüne et al.'s RAM process, to make use of Syriani's T-Core as its model transformation kernel, and to rigorously apply the "meta-model and conforming model" workflow to all facets of DSM.

At this time, ongoing implementation and research work focuses mostly on the completion of the transformation debugging mode and the implementation of fully integrated model versioning facilities.

Conclusions

Summary

Domain-specific modelling makes a list of ambitious claims at the top of which is that it can significantly shorten and often completely close the notoriously large conceptual gap between problems and their realizations in the solution domain, a feat that no other computer-assisted development technique has achieved. While these claims have been substantiated in several documented industrial and academic efforts, more focus is often given to the benefits of DSM than to the lower-level implementation details of how exactly the aforementioned conceptual gap is tackled, or to the supporting tooling (or lack thereof). These unsung implementation details often comprise the specification of complex code-generators that make little use of reusable modules and that adhere to no standardized development guidelines.

Guided by the principles of Multi-Paradigm Modelling, this thesis targeted the foundations of DSM, proposing more modular, structured, scalable and most of all principled means of realizing the promises of DSM. The contributions of each of its chapters are reviewed below.

Structuring the Synthesis of Artifacts from Models *(Chapter 2)*

In this chapter, we proposed a structured approach to artifact synthesis from DSms that addresses the numerous flaws of widely adopted (within the DSM community) artifact synthesis techniques through the application of MPM principles. Layered model transformations are used to modularly isolate, compile and re-combine various concerns within DSms, while maintaining traceability links between corresponding constructs at different levels of abstraction. A thorough study of the approach revealed a number of its wide-ranging benefits, one of which is its simplifying effect on the notoriously difficult problem of addressing non-functional requirements (e.g., timing and resource utilization constraints). We demonstrated the approach through the synthesis of fully functional Google Android applications from DSms of mobile phone applications.

Debugging in Domain-Specific Modelling *(Chapter 3)*

In this chapter, we proposed a mapping between common debugging concepts such as breakpoints and assertions from the code-centric development realm to the DSM realm. The meaning of these concepts was explored from both the modeller's and the DSM expert's points of view, where the tasks and debugging requirements of the former are akin to those of programmers, while those of the latter are akin to those of compiler builders. Guidelines, caveats and examples are provided, many of which are implemented and demonstrated, as blueprints for future DSM debuggers. They also serve to demystify the amount of effort required to produce DSM debuggers.

Domain-Specific Engineering of Domain-Specific Languages *(Chapter 4)*

In this chapter, we proposed a DSML engineering approach that automates much of the complex tasks traditionally associated with the specification of DSML syntax and semantics. The basic "domain constructs" of the domain of DSML engineering are identified as being portions of lower-level modelling formalisms that capture commonly occurring syntactic and semantic concepts and structures in DSMLs. Then, a template-based approach for composing these DSML building blocks into new DSMLs is proposed. The approach is demonstrated on two very different DSMLs and studied to clearly identify its benefits and limitations, and the scope of its applicability.

AToMPM: A Tool for Multi-Paradigm Modelling *(Chapter 5)*

In this chapter, we presented AToMPM, a new tool for MPM, that addresses usability, accessibility and/or scientific limitations common to popular DSM tools. In addition to several technical innovations and improvements, which include a Web-based client and support for real-time, distributed collaboration, its main scientific interest lies in its theoretically sound approach towards the specification of modelling language syntax and semantics and of model transformation rules and pre- and post-condition pattern languages, and in its implementation and integration of recent research work by ourselves and others. Its features and architecture were reviewed in the context of its objectives and requirements.

Outlook

This thesis makes a number of contributions to the field of DSM. Nevertheless, many challenges still stand between DSM and its adoption in industry as a viable and scalable development discipline. Those we deem most relevant are reviewed below.

Model Versioning

Researchers in the area of model versioning wish to provide modellers with the same version control utilities than those programmers have been using over the past decades. As modelling efforts are growing in scale, the need to work with several sequential and parallel versions of models is increasing. Storing all these versions is an impractical solution due to space and bandwidth concerns. Under this premise, the manipulation of model differences becomes key. Thus the problems of computing, visualizing and merging model differences arise. A significant body of research tackles each of these challenges and means to achieve them (semi-)automatically have been proposed [AP03, OWK03, XS05, CRP07, TELW12].

Brosch et al. go one step further and propose collaborative conflict resolution [BSWW09]. Their rationale is that merging different model versions may sometimes be very complex and that the authors of each version should participate in the merging effort. Though their work has been prototyped within the EMF, it may be interesting, especially in the context of *distributed* collaboration, to implement it within AToMPM.

Other researchers have proposed even more advanced uses for model differencing. For instance, in [LGJ07], Lin et al. suggest that model differences be used to compare the results of model transformations with generated or specified desired results, thereby enabling model transformation testing and verification through model differencing.

Model (Co-)Evolution

Unlike GPLs, which are specifically designed to be useful in the implementation of applications from an ever growing pool of domains, DSMLs are specifically designed to be tightly coupled with a single domain. This is desirable because it enables domain experts to readily create, understand and manipulate DSms. However, this coupling makes DSMLs very sensitive to changes in their associated domains. Thus, means of evolving all aspects of DSMLs (i.e., their abstract and concrete syntaxes as well as their semantics) and of conforming DSms are especially crucial. Other evolution-prompting events include the need for more convenient syntactic constructs and changes to target solution domains (i.e., the target domains of DSm compilers). Hence, means to co-evolve meta-models, model transformations and models as a result of external evolution must be available. Research in this field revolves around automating co-evolution in various scenarios and identifying those scenarios where human intervention is unavoidable [SK04, ZGL04, CREP08, MV11c].

Model and Model Transformation Debugging

We have already made the case of how critically important complete debugging facilities are to DSM's widespread adoption, and have proposed guidelines for future debuggers. A next step is that a debate surrounding our proposals take place within the DSM community to further specify the requirements of debuggers for all facets of the DSM development process. Further study of what is the meaning of certain debugging concepts, breakpoints in particular, in the context of formalisms where the notion of state is implicit is also necessary.

Semantic Template-based DSML Design

Despite our demonstrations and argumentation in favour of using semantic templates, when appropriate, to alleviate the burden placed on DSM experts and facilitate and automate much of the DSML engineering process, we recognize that a more objective, empirical validation is required and suggest that a user study be conducted to formally compare several metrics, including productivity and amount of user-input, between traditional DSML design and engineering techniques and our proposed approach. Given positive results, future research surrounding ST-based DSML design might explore extending our proposed set of base formalisms to increase its *usefulness*. For instance, performance models may be a useful base formalism. Instance DSms of DSMLs built on top of a performance model base formalism could be mapped onto performance models and more (e.g., performance measurement artifacts) without any modeller or designer input. Moreover, relevant performance notions would be elevated to the DSm level, making measurement reporting at that level possible and sensible.

An implicit and pressing need is of course for results from all of the above research areas to be integrated into widely accessible DSM tools, such as AToMPM.

List of Publications

Structuring the Synthesis of Artifacts from Models

[1] Bart De Decker, Jorn Lapon, Mohamed Layouni, Raphael Mannadiar, Vincent Naessens, Hans Vangheluwe, Pieter Verhaeghe, and Kristof Verslype (Ed.). Advanced Applications for e-ID Cards in Flanders. adapID Deliverable D12. Technical Report, KU Leuven, 2009.

[2] Raphael Mannadiar and Hans Vangheluwe. Modular Synthesis of Mobile Device Applications from Domain-Specific Models. Technical Report SOCS-TR-2010.5, McGill University, 2010.

[3] Raphael Mannadiar and Hans Vangheluwe. Modular Synthesis of Mobile Device Applications from Domain-Specific Models. In *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES). Part of the International Conference on Automated Software Engineering (ASE).*, MOMPES '10, pages 21–28, 2010.

[4] Raphael Mannadiar and Hans Vangheluwe. Modular Artifact Synthesis from Domain-Specifc Models. *Journal of Innovations in Systems and Software Engineering (ISSE)*, 8(1):65–77, 2011.

Debugging in Domain-Specific Modelling

[5] Raphael Mannadiar and Hans Vangheluwe. Debugging in Domain-Specific Modelling. In *Proceedings of the 3rd International Conference on Software Language Engineering (SLE)*, volume 6563 of Lecture Notes in Computer Science (LNCS), pages 276–285. Springer, 2011.

Domain-Specific Engineering of Domain-Specific Languages

[6] Raphael Mannadiar and Hans Vangheluwe. Domain-Specific Engineering of Domain-Specific Languages. In *Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM). Part of Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)*, DSM '10, pages 11:1–11:6, 2010.

[7] Raphael Mannadiar and Hans Vangheluwe. Modular Design of Domain-Specific Languages. *Journal of Software and Systems Modeling (SoSym)*, (pending 2nd review), 2012.

Appendix A

Acronyms

API	Application Programming Interface
AToM^3	A Tool for Multi-formalism and Meta-Modelling
AToMPM	A Tool for Multi-Paradigm Modelling
CORBA	Common Object Request Broker Architecture
CRUD	Create Read Update Delete
DEVS	Discrete EVent system Specification
DSm	Domain-Specific model
DSM	Domain-Specific Modelling
DSML	Domain-Specific Modelling Language
EMF	Eclipse Modelling Framework
GME	Generic Modeling Environment
GMF	Graphical Modelling Framework
GReAT	Graph Rewriting And Transformation
GPL	General Purpose Language
GUI	Graphical User Interface

HOT	Higher-Order Transformation
HUTN	Human Readable Textual Notations
IDE	Integrated Development Environment
I/O	Input/Output
LHS	Left-Hand Side
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MOF	Meta-Object Facility
MTL	Model Transformation Language
MPM	Multi-Paradigm Modelling
MTBD	Model Transformation By Demonstration
NAC	Negative Application Condition
OCL	Object Constraint Language
OMG	Object Management Group
OOP	Object-Oriented Programming
QVT	Query/View/Transformation
RAM	Relaxation Augmentation Modification
RHS	Right-Hand Side
SOAP	Simple Object Access Protocol
SoC	Separation of Concerns
SPL	Software Product Line
ST	Semantic Template
SVG	Scalable Vector Graphics

UML	Unified Modelling Language
VMTS	Visual Modelling and Transformation System
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Bibliography

[act] actifsource GmbH. Actifsource. `http://www.actifsource.com/`.

[AK08] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software and Systems Modeling (SoSym)*, 7:345–359, 2008.

[AKN$^+$06] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software and Systems Modeling (SoSym)*, 5:261–288, 2006.

[AP03] Marcus Alanen and Ivan Porres. Difference and union of models. In *Unified Modeling Language (UML)*, volume LNCS 2863, pages 2–17, 2003.

[Appa] Apple. Apple iOS. `http://www.apple.com/ios/`.

[Appb] Apple. Apple Safari. `http://www.apple.com/safari/`.

[B́05] Jean Bézivin. On the unification power of models. *Software and Systems Modeling (SoSym)*, 4:171–188, 2005.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72:52–70, 2008.

[Bro87] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[Bro04] Alan W. Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling (SoSym)*, 3:314–327, 2004.

[BSWW09] Petra Brosch, Martina Seidl, Konrad Wieland, and Manuel Wimmer. We can work it out: Collaborative conict resolution in model versioning. In *Proceedings of the 11th European Conference on Computer-Supported Cooperative Work (ECSCW)*, pages 207–214, 2009.

[Bur87] Steve Burbeck. Applications programming in smalltalk-80tm: How to use model-view-controller mvc, 1987. http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

[BV09] Jacob Beard and Hans Vangheluwe. Modelling the reactive behaviour of svg based scoped user interfaces with hierarchically linked statecharts. In *SVG Open*, 2009. http://svgopen.org/2009/papers/.

[CCG$^+$09] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *7th joint meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, 2009.

[CD09] Shane Conder and Lauren Darcey. *Android Wireless Application Development*. Addison-Wesley Professional, 1st edition, 2009.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

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

[CLR00] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 2000.

[Cor06] James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61:190–210, 2006.

[Cox87] Brad Cox. *Object-oriented Programming*. Addison Wesley Longman Publishing Co, March 1986 edition, 1987.

[CREP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing (EDOC)*, pages 222–231, 2008.

[CRP07] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. A metamodel independent approach to difference representation. *Journal of Object Technology (JOT)*, 6:165–185, 2007.

[CSN05] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *International Conference On Embedded Software*, pages 35–43, 2005.

[Dah02] Ole-Johan Dahl. The roots of object orientation: the Simula language. *Software pioneers*, pages 78–90, 2002.

[DBV09] Denis Dubé, Jacob Beard, and Hans Vangheluwe. Rapid development of scoped user interfaces. In *Human Computer Interaction (HCII)*, volume 5610 of *Lecture Notes in Computer Science (LNCS)*, pages 816–825, 2009.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

[dLG10] Juan de Lara and Esther Guerra. Deep meta-modelling with MetaDepth. In *TOOLS Europe 2010: 48th International Conference on Objects, Models, Components, Patterns*, volume LNCS 6141, pages 1–20, 2010.

[DLL$^+$09] Bart De Decker, Jorn Lapon, Mohamed Layouni, Raphael Mannadiar, Vincent Naessens, Hans Vangheluwe, Pieter Verhaeghe, and Kristof Verslype (Ed.). Advanced applications for e-ID cards in flanders. adapid deliverable D12. Technical report, KU Leuven, 2009.

[dLV02] Juan de Lara and Hans Vangheluwe. AToM^3: A tool for multi-formalism modelling and meta-modelling. *Lecture Notes in Computer Science*, 2306:174–188, 2002.

[dLVA04] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modeling (SoSym)*, 3:194–209, 2004.

[DMN70] Ole-Johan Dahl, Bjørm Myhrhaug, and Kristen Nygaard. *Common Base Language*. Publication No. S-22 / Norwegian Computing Center. Norwegian Computing Center, 1970.

[EB04] Claudia Ermel and Roswitha Bardohl. Scenario animation for visual behavior models: A generic approach. *Software and Systems Modeling (SoSym)*, 3:164–177, 2004.

[EFa] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`.

[EFb] The Eclipse Foundation. Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`.

[EFc] The Eclipse Foundation. Graphical Modelling Framework. `http://www.eclipse.org/modeling/gmp/`.

[Eis97] Marc Eisenstadt. “My Hairiest Bug” war stories. *Communications of the ACM (CACM)*, 40:30–37, 1997.

[ES06] Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *6th Workshop on Domain Specific Modeling at OOPSLA*, pages 123–139, 2006.

[Fac] Facebook. Facebook. `http://www.facebook.com/`.

[Fav06] Jean-Marie Favre. Megamodelling and etymology. In James R. Cordy, Ralf Lämmel, and Andreas Winter, editors, *Transformation Techniques in Software Engineering*, volume 427, 2006.

[Gooa] Google. Google. `http://www.google.com/`.

[Goob] Google. Google Android. `http://code.google.com/android/`.

[Gooc] Google. Google Chrome. `https://www.google.com/chrome`.

[Good] Google. Google Docs. `https://docs.google.com/`.

[GTK$^+$07] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, and Jonathan Sprinkle. *Domain-Specific Modeling*, chapter 7, pages 1–20. CRC Press, 2007.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *The Science of Computer Programming*, 8:231–274, 1987.

[HK04] David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). *Integration of Software Specification Techniques forApplications in Engineering*, LNCS 3147:325–354, 2004.

[HKGV10] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code generation by model transformation: a case study in transformation modularity. *Software and Systems Modeling (SoSym)*, 9:375–402, 2010.

[Hoa85] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HT08] Jörg Holtmann and Matthias Tichy. Component story diagrams in fujaba4eclipse. In *6th International Fujaba Days*, 2008.

[Jen86] Kurt Jensen. Coloured petri nets. In *Advances in Petri Nets*, pages 248–299, 1986.

[Jet] JetBrains. Meta programming system – language oriented programming environment and dsl creation tool. `http://www.jetbrains.com/mps/`.

[JK06] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *MTiP'05*, volume 3844 of *LNCS*, pages 128–138, 2006.

[K$\ddot{0}$6] Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling (SoSym)*, 5:369–385, 2006.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineereing Institute, Carnegie Mellon University, 1990.

[KGHR10] Lucia Kapova, Thomas Goldschmidt, Jens Happe, and Ralf H. Reussner. Domain-specific templates for refinement transformations. In *in 1st Workshop on Model Driven Interoperability (MDI)*, 2010.

[KKMK11] Tomaž Kos, Tomaž Kosar, Marjan Mernik, and Jure Knez. Ladybird: debugging support in the sequencer. In *Proceedings of the 2011 American conference on applied mathematics and the 5th WSEAS international conference on Computer engineering and applications*, pages 135–139, 2011.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin1. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, volume LNCS 1241, 1997.

[KMS+10] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 240–255. Springer Berlin / Heidelberg, 2010.

[KR10] Lucia Kapova and Ralf Reussner. Application of advanced model-driven techniques in performance engineering. In *7th European Performance Engineering Workshop (EPEW)*, volume LNCS 6342, pages 17–36, 2010.

[KSWR09] Angelika Kusel, Wieland Schwinger, Manuel Wimmer, and Werner Retschitzegger. Common Pitfalls of Using QVT Relations - Graphical Debugging as Remedy. In *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 329–334, 2009.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling : Enabling Full Code Generation*. Wiley-Interscience, 2008.

[LGJ07] Yuehua Lin, Jeff Gray, and Frederic Jouault. DSMDiff: A differentiation tool for domain-specific models. *European Journal of Information Systems (EJIS)*, 16:349–361, 2007.

[LLMC06] László Lengyel, Tihamér Levendovszky, Gergely Mezei, and Hassan Charaf. Model transformation with a visual control flow language. *International Journal of Computer Science (IJCS)*, 1:45–53, 2006.

[LLMM08] Tihamér Levendovszky, László Lengyel, Gergely Mezei, and Tamás Mészáros. Introducing the VMTS mobile toolkit. In *Applications of Graph Transformations* *with Industrial Relevance*, volume LNCS 5088, pages 587–592. Springer Berlin / Heidelberg, 2008.

[LMB$^+$01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstroma, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Proceedings of the IEEE International Workshop on Intelligent Signal Processing (WISP)*, 2001.

[Man] Raphael Mannadiar. *AToMPM User's Manual.*

[Met] MetaCase. Domain-specific modeling with MetaEdit+: 10 times faster than UML. http://www.metacase.com/resources.html; June 2009.

[Mid03] Middleware Company. Model driven development for J2EE utilizing a model driven architecture (MDA) approach : Productivity analysis. Report by the Middleware Company, 2003.

[Moza] Mozilla Corporation. Gecko. `https://developer.mozilla.org/en/Gecko`.

[Mozb] Mozilla Corporation. Mozilla Firefox. `www.mozilla.org/en-US/firefox/`.

[MSMG10] Daniel A. Menascé, Joao P. Sousa, Sam Malek, and Hassan Gomaa. QoS Architectural Patterns for Self-Architecting Software Systems. In *7th IEEE International Conference on Autonomic Computing and Communication*, 2010.

[MV04] Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation: Transactions of The Society for Modeling and Simulation International*, 80(9):433–450, 2004.

[MV10] Raphael Mannadiar and Hans Vangheluwe. Domain-specific engineering of domain-specific languages. In *10th Workshop on Domain-Specific Modeling (DSM). Part of Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)*, DSM '10, pages 11:1–11:6. HSE-Press, B-120, 2010.

[MV11a] Raphael Mannadiar and Hans Vangheluwe. Debugging in domain-specific modelling. In *3rd International Conference on Software Language Engineering (SLE)*, volume LNCS 6563, pages 276–285. Springer, 2011.

[MV11b] Raphael Mannadiar and Hans Vangheluwe. Modular artifact synthesis from domain-specifc models. *Journal of Innovations in Systems and Software Engineering (ISSE)*, 8(1):65–77, 2011.

[MV11c] Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. *Science of Computer Programming*, 76(12):1223–1246, 2011.

[NR69] Peter Naur and Brian Randell. *Software Engineering: Report of a conference sponsored by the NATO Science Committee.* Scientific Affairs Division, NATO, 1969.

[NZ99] Jörg Niere and Albert Zündorf. Using fujaba for the development of production control systems. In *International Workshop and Symposium on Applications Of Graph Transformations With Industrial Relevance (AGTIVE)*, 1999.

[Obja] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)*. OMG. `http://www.omg.org/spec/QVT/`.

[Objb] Object Management Group. *Model-Driven Architecture (MDA)*. OMG. `http://www.omg.org/mda/specs.htm`.

[Obj06] Object Management Group. *Meta Object Facility (MOF) Core Specification Version 2.0*. OMG, January 2006. `http://www.omg.org/spec/MOF/`.

[Obj09] Object Management Group. *Unified Modeling Language (UML) Superstructure*. OMG, February 2009. `http://www.omg.org/spec/UML/`.

[Obj10] Object Management Group. *Object Constraint Language Version 2.2*. OMG, 2010. `http://www.omg.org/spec/OCL/`.

[OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of UML diagrams. In *European Software Engineering Conference / Foundations of Software Engineering (ESEC/FSE)*, pages 227–236, 2003.

[Ped09] Luis Miguel Venceslau Pedro. *A Systematic Language Engineering Approach for Prototyping Domain Specific Modelling Languages*. PhD thesis, Université de Genève, 2009.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[Por05] Ivan Porres. Rule-based update transformations and their application to model refactorings. *Software and Systems Modeling (SoSym)*, 4:368–385, 2005.

[Saf07] Laurent Safa. The making of user-interface designer a proprietary DSM tool. In *7th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, page 14, http://www.dsmforum.org/events/DSM07/papers.html, 2007.

[sC76] Peter Pin shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[Sch06] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39:25–31, 2006.

[SK04] Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing (JVLC)*, 15:291–307, 2004.

[SKV10a] Eugene Syriani, Jörg Kienzle, and Hans Vangheluwe. Exceptional transformations. Technical Report SOCS-TR-2010.2, McGill University, 2010.

[SKV10b] Eugene Syriani, Jörg Kienzle, and Hans Vangheluwe. Exceptional transformations. In *International Conference on Model Transformation (ICMT)*, volume LNCS 6142, pages 199–214. Springer, 2010.

[Sun11] Yu Sun. *Model Transformation By Demonstration: A User-Centric Approach To Support Model Evolution*. PhD thesis, University of Alabama at Birmingham, 2011.

[SV09] Eugene Syriani and Hans Vangheluwe. *Discrete-Event Modeling and Simulation: Theory and Applications.*, chapter DEVS as a Semantic Domain for Programmed Graph Transformation. CRC Press, 2009.

[SV10] Eugene Syriani and Hans Vangheluwe. De-/re-constructing model transformation languages. In *9th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT)*, 2010.

[SWG09] Yu Sun, Jules White, and Jeff Gray. Model transformation by demonstration. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages & Systems (MODELS)*, volume LNCS 5795, pages 712–726. Springer, 2009.

[Syr11] Eugene Syriani. *A Multi-Paradigm Foundation for Model Transformation Language Engineering*. PhD thesis, School of Computer Science, McGill University, 2011.

[TELW12] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: from theory to implementation. *Software and Systems Modeling (SoSym)*, TBD:1–34, 2012.

[The] The MathWorks. Simulink. `http://www.mathworks.com/products/simulink/`.

[TP08] Rasha Tawhid and Dorina Petriu. Integrating performance analysis in the model driven development of software product line. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 490–504, 2008.

[Tra05] Laurence Tratt. Model transformations and tool integration. *Software and Systems Modeling (SoSym)*, 4:112–122, 2005.

[Val10] Antonio Vallecillo. On the combination of domain specific modeling languages. In *European Conference on Modeling Foundations and Applications (ECMFA)*, volume LNCS 6138, pages 305–320. Springer, 2010.

[VB07] Dániel Varró and András Balogh. The model transformation language of the viatra2 framework. *Science of Computer Programming*, 68:214–234, 2007.

[VK05] Mahesh Viswanathan and Moonzoo Kim. Foundations for the run-time monitoring of reactive systems - fundamentals of the mac language. In *First International Colloquium on Theoretical Aspects of Computing (ICTAC)*, volume LNCS 3407, pages 543–556. Springer, 2005.

[W3C] W3C. The WebSocket API. `http://dev.w3.org/html5/websockets/`.

[web] The webkit open source project. `http://www.webkit.org/`.

[WGM08] Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Software : Practice and Experience*, 38:1073–1103, 2008.

[WHT$^+$09] Jules White, James H. Hill, Sumant Tambe, Aniruddha Gokhale, Douglas C. Schmidt, and Jeff Gray. Improving domain-specific language reuse with software product line techniques. *IEEE Software*, 26:47–53, 2009.

[Wika] Wikipedia. Comet. `http://en.wikipedia.org/wiki/Comet_(programming)`.

[Wikb] Wikipedia. Java version history. `http://en.wikipedia.org/wiki/Java_version_history`.

[WSNW07] Jules White, Douglas C. Schmidt, Andrey Nechypurenko, and Egon Wuchner. Introduction to the generic eclipse modeling system. *Eclipse Magazine*, 6:11–18, 2007.

[XS05] Zhenchang Xing and Eleni Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Automated Software Engineering*, pages 54–65, 2005.

[YCDW10] Andrés Yie, Rubby Casallas, Dirk Deridder, and Dennis Wagelaar. Realizing model transformation chain interoperability. *Software and Systems Modeling (SoSym)*, 9:1–21, 2010.

[Zel09] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, second edition, 2009.

[ZGL04] Jing Zhang, Jeff Gray, and Yuehua Lin. A generative approach to model interpreter evolution. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 121–129, 2004.

[ZKP00] Bernard Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation - Integrating Discrete Event and Continous Complex Dynamic Systems.* Academic Press, 2nd edition, 2000.