Domain-Specific Engineering of Domain-Specific Languages

[†]Raphael Mannadiar
† McGill University
3480 University Street
Montreal, Quebec, Canada
rmanna@cs.mcgill.ca

[†,‡]Hans Vangheluwe
‡ University of Antwerp
Middelheimlaan 1
2020 Antwerpen, Belgium
hv@cs.mcgill.ca

ABSTRACT

Domain-specific modelling (DSM) enables experts of arbitrary domains to perform modelling tasks using familiar constructs. This contrasts with common code-centric development approaches where programmers deal with object-oriented approximations of higher level concepts. Domain-specific concepts and their relationships are captured by domain-specific languages (DSLs). Unfortunately, it is common practice for DSLs to be specified within the object-oriented mindsets of classes and associations. This approach not only contradicts the model-driven engineering (MDE) philosophy of development using domain-specific concepts – in this case, the domain and concepts of DSLs –, it is also faced with the same obstacle as past UML-to-code generation efforts; namely, that UML models are too generic to enable complete program synthesis. In the context of DSL engineering, this obstacle translates to the necessity for DSL designers to explicitly define DSL semantics manually (e.g., via coded generators and/or model transformations). In this work, we propose a novel approach to DSL design where low level modelling formalisms are seamlessly woven together to form new DSLs whose semantics are fully automatically generated.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Software Architectures; D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures, information hiding, languages*

General Terms

Design, Languages, Standardization

Keywords

Language design, Language weaving, Metamodelling templates, Fully generated semantics, Multi-paradigm modelling, Google Android

1. INTRODUCTION

Domain-specific modelling (DSM) allows domain-experts to play active roles in development efforts. It provides them with means to manipulate constructs they are familiar with and automate the many error-prone and time consuming translation steps that characterize code-centric development efforts – most notably, the manual mapping between the (often far away) problem and solution domains –. This automated transformation of domain-specific models (DSms[1]) to complete artifacts (e.g., executable programs, other models) is enabled by their tightly constrained nature – as opposed to the general purpose nature of UML models, for instance, which are used to model programs from any domain using object-oriented constructs –. Empirical evidence suggests increases in productivity of up to one order of magnitude when using DSM as opposed to traditional code-driven development approaches [15, 12, 14].

Despite the stated merits of DSM, the guiding principles that enable full artifact generation from DSms have yet to be incorporated into the work flow of domain-specific language (DSL) 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. Modern DSL engineering techniques are still rooted in the object-oriented mindsets of entities and relations and thus, the means, constructs and techniques used to describe DSLs are more akin to UML modelling than to DSM. A direct consequence of this is that, much like common UML models do not hold sufficient information to generate complete programs, those of DSLs do not hold sufficient information for the full semantics of the modelled languages to be automatically synthesized. Hence, it falls upon the DSL engineer to explicitly specify language semantics manually using coded generators and/or model transformations.

Not only is the task of manually defining DSL semantics non-trivial, it is also repetitive: it is conceivable that numerous languages share some semantics. Consider, for instance, the subset of all DSLs where the notions of state and transition exist. It is sensible 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 prime targets for automation via DSM. Given a number of low level formalisms that encompass commonly recurring concepts in DSLs (e.g., Statecharts for states and transitions [11]) and the means to translate these formalisms to low-level artifacts (e.g., a Statechart to code transformation), it would be possible to automatically generate the full semantics of new DSLs constructed exclusively in terms of these "domain-specific" concepts . In other words, if DSL engineers were to adopt a higher level and more domain-specific approach to DSL specification, the

[1]Note that we refer to domain-specific modelling as DS*M* and to a domain-specific model as a DS*m*.

need for them to manually define DSL semantics would be entirely eliminated.

The rest of this paper is structured as follows. In Section 2, we survey the current state-of-the-art of DSL and semantics specification and engineering techniques. In Section 3, we propose a set of low level modelling formalisms that form a "basis" for (re-)constructing – ideally – any conceivable DSL. In Section 4, we detail how new DSLs can be related to the "base formalisms" from Section 3 such that their full semantics become implicit. In Section 5, we discuss how instance models of these DSLs are turned into complete and meaningful artifacts. In Section 6, we detail how a non-trivial formalism for modelling mobile device applications can be defined under our approach. Finally, in Section 7, we discuss future work and provide some closing remarks.

2. BACKGROUND AND RELATED WORK

Domain-specific languages can essentially be broken down into three components. Their *abstract syntax* describes language concepts and the relationships between them, as well as constraints that encode domain rules. Their *concrete syntaxes*[2] provide graphical and/or textual representations of abstract syntax elements. Finally, their meaning or semantics are commonly defined *operationally* or *denotationally*[3]. 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 (e.g., for Petri Nets, one such rule could describe the model before and after the firing of an enabled *transition*). As for denotational semantics, they essentially define the meaning of a DSL by mapping its concepts onto formalisms for which operational or denotational semantics are well defined (e.g., code, mathematics, Petri Nets).

The most popular means of DSL abstract syntax specification and communication today are UML class diagrams[4] and human readable textual notations (HUTNs). The former are strongly privileged by visual metamodelling tools and are the preferred means of DSL graphical representation in modern publications [8, 14, 5, 15, 2, 12, 9, 13]. The latter are privileged by textual metamodelling tools; example HUTNs include TXL [6], Stratego/XT [3] and MetaDepth [7].

Recently, several researchers have studied the problem of DSL combination. On the one hand, DSL combination is desired for merging distinct views of a single system (e.g., a UML class diagram describing a system's structure with Statecharts describing its behaviour). In [16], Vallecillo argues that it is unrealistic to model large and complex systems with a single instance model of a single DSL, and that instead it is preferable to model different facets of such systems with distinct instance models of distinct DSLs. Thus, the broad survey of model and metamodel combination approaches Vallecillo provides, as well as his own *viewpoint unification* technique are mostly aimed at the merging of interrelated models and metamodels. On the other hand, other authors have tackled the problem of DSL combination from an engineering perspective, studying how recurring structures can be turned into generic building blocks. In [10], Emerson and Sztipanovits target the reuse of parts or all of existing metamodels to address the repeated redefinition of popular metamodelling patterns. Their technique, *template instantiation*, consists in presenting the metamodeller 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 metamodeller can then instantiate these templates with his own domain-specific concepts yielding appropriately customized metamodel patterns in a timely and more standardized manner.

Finally, although modern DSL syntax engineering does lack in formality, the most ad hoc step of any DSM project remains the definition of a DSL's semantics. The common approach – at least, for DSms of executable systems – is to encode the denotational semantics of a language within a hand-crafted code generator[5] [14, 15, 12, 17]. In [13], we argue and demonstrate that modelling – as opposed to coding – the mapping onto lower level modelling formalisms (via layered model transformations) is more modular, adheres more closely to the multi-paradigm modelling (MPM) principles [8], and considerably facilitates advanced tasks like model and transformation debugging. Nevertheless, despite improving upon coded generators, the approach we propose still relies on the ad hoc manual specification of semantic mappings. In [4], the notion of *semantic anchoring* is explored in the context of formalizing and semi-automating DSL semantics specification. First, library-like reusable *semantic units* are defined to capture commonly occurring semantic patterns. Then, syntactic templates – much like those from [10] – are mapped onto appropriate *semantic units* by tool developers. The result is a language engineering environment where *partial* semantics can be generated.

The approach we present in this work is in fact a combination and extension of past and current work on template instantiation and semantic anchoring with the specific aim of *fully* automating DSL semantics specification while reinforcing the process with deeper and more structured roots.

3. A BASIS FOR DSL DESIGN

In past work, we demonstrated how separate concerns (e.g., layout and behaviour) within a single DSm could be isolated and mapped onto concern-specific lower level formalisms (e.g., Statecharts for behaviour), and later woven back together into complete artifacts [13]. Our current claims are that (1) any conceivable DSL is in fact nothing more than a combination of a finite set of lower level formalisms, and (2) that the knowledge of how these formalisms are combined to form a given DSL is sufficient to infer that DSL's full semantics. These claims introduce several questions:

1. Which modelling formalisms form this *basis for DSL design*?

[2]A single DSL may have more than one concrete syntax.

[3]Denotational semantics are commonly referred to as *translational* semantics.

[4]For our purposes, Entity-Relationship diagrams can be encompassed within UML class diagrams.

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

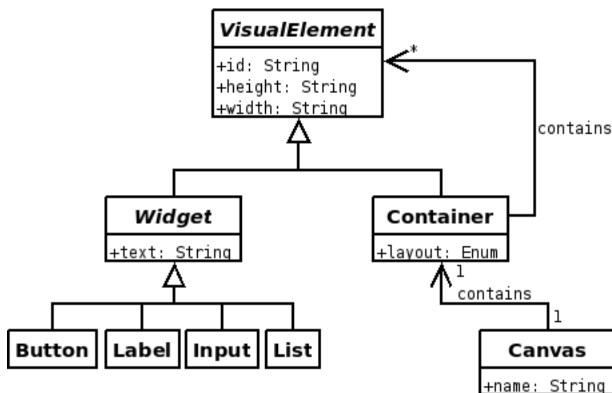

Figure 1: The *Layout* metamodel (as a UML class diagram).

2. How can new DSLs be defined in terms of these *base formalisms*?

3. How can complete artifacts be generated from instance models of these DSLs (without the need to manually define DSL semantics)?

This section will explore the first of these questions. The remainder will be discussed in Sections 4 and 5.

From our experience designing metamodels and their semantic mappings to lower level formalisms, and keeping with the traditional mathematical interpretation of what is a basis (i.e., a minimal set of d-dimensional vectors that can be added to produce any vector in $\mathbb{R}^d$), we are able to list a few formalisms that a basis for DSL design could hardly do without.

Statecharts are useful for the modelling of reactive behaviour. They provide notions of event- and timeout- triggered transitions between possibly composite states with entry and exit actions. Furthermore, they are able to capture concurrency through *orthogonal components*.

Petri Nets are useful for modelling distributed processes and their synchronization. They provide notions of concurrent and non-deterministic resource consumption and production.

Causal Block Diagrams (CBDs) are useful for modelling physical systems. They provide notions of continuous flow of data between primitive mathematical operators. Furthermore, they can be easily integrated into discrete systems by making use of their zero-crossing detection constructs to implement thresholding.

Layout is a formalism we introduce here for modelling user interfaces. Its metamodel is presented in Figure 1. Essentially, a `Canvas` can contain `Container`s which can in turn contain other `Container`s and/or `Widget`s. This simple metamodel could of course be extended with additional features to gain in expressiveness.

Action Code is a formalism we introduce here for modelling API method calls and user-provided code; its metamodel is presented in Figure 2.

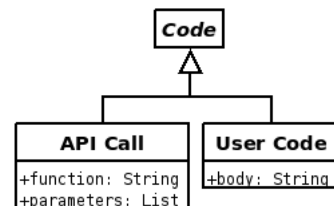

Figure 2: The *Action Code* metamodel (as a UML class diagram).

The five formalisms above enable the modelling of determinism and non-determinism, states and transitions, discrete and continuous systems, user interfaces, APIs and code-based escape semantics. Appropriately combined, they can thus produce a wide variety of expressive languages. Means of achieving this combination are discussed in the following section.

4. RELATING DSL AND BASE FORMALISM

Given a set of base formalisms, the next question to answer is how they can aid in defining new DSLs. Keeping in mind our initial goal of fully automating the specification of DSLs semantics, this question can be rephrased to: "how can domain-specific concepts be related to base formalisms tightly enough to enable the automatic generation of *DSL-to-base-formalism* transformations"?

We propose an approach that builds on Emerson and Sztipanovits' template instantiation technique from [10]. In their work, templates are little more than syntactic sugar that enable the reuse of common metamodelling patterns. We instead propose to promote syntactically and semantically rich templates to the foreground of DSL design. These *semantic templates* (STs) are no longer placeholders for isolated structures but instead "interfaces" to base formalisms. Each base formalism "exposes" a set of STs that encode the unambiguous mapping of arbitrary domain-specific concepts onto concepts from the given base formalism.

Instantiating a ST accomplishes two tasks. First, it creates appropriate classes and associations in an internal UML class diagram representation of the DSL at hand. Second, it creates relationships between classes representing domain-specific entities and those from the internal UML class diagram representation of the given base formalism's metamodel. In practice, model transformation facilities can be exploited to implement these tasks. The STs themselves are matched by transformation rule pre-conditions while the implied classes and associations – examples of which are given in Table 1 – are produced by rule post-conditions. Although UML class diagrams still have a role to play in metamodel representation, they are no longer first class artifacts but are rather relegated to internal representations meant as input for metamodelling tools (e.g., to generate domain-specific model editors) and to facilitate later model transformations. In the limit, the DSL engineer need not be aware of the content or even of the existence of the underlying UML class diagram. Table 1 overviews example STs for the formalisms

in our proposed basis as well as the syntactic and semantic implications they carry.

The question of how STs should be presented to the DSL engineer remains. Two options come to mind. The first is to present the list of all base formalisms to the user and have him choose which ones he wishes to build his language on. For instance, a DSL engineer could indicate he wishes to base his language on Petri Nets and *Layout*. He would then be presented with generic STs and STs specific to both selected base formalisms. The second option is to group STs by feature. In this case, the user could select required features (e.g., non-determinism, escape semantics) from a list[6] and then be presented with possible formalisms, or even directly with STs.

Past attempts at using templates, especially in the context of programming, were met with much reserve because of how restrictive the resulting development environments were. Given appropriate tool support, our method need not be subject to this limitation: providing DSL engineers with means to specify STs for their own languages would enable these new languages to themselves be used as base formalisms thereby enabling the potentially unlimited increase in expressiveness and flexibility of the available collection of STs. It is our hope that future tools will be bundled with a wide array of available *semantically-templated base formalisms* along with facilities to define new ones.

Although we have explained how DSLs can be constructed by combining specific low level formalisms, the manner in which their semantics can be automatically derived remains unaddressed. We shed light on this point in the following section.

5. FROM MODELS TO ARTIFACTS

We have discussed which formalisms constitute our basis for DSL design and how new DSLs can be built by instantiating STs associated to base formalisms. We now focus on the question of how to generate complete artifacts from instance models of DSLs built using our approach without the need to manually define DSL semantics. We faced a similar challenge in [13] where layout and behavioural concerns were tangled in DSms of mobile device applications. The essence of the solution we presented there was two-fold[7]. The first task was to iteratively project portions of DSms onto appropriate lower level formalisms and then onto code. The second was to automatically weave message passing facilities into the generated code to enable communication between the synthesized artifacts corresponding to each of the earlier projections.

The solution we propose here is nearly identical, with the sole difference that it no longer falls upon the DSL designer to manually define coded generators and/or model transformations to map hand-picked portions of DSms onto lower level formalisms. Indeed, the fact that DSLs are completely defined in terms of base formalisms provides the necessary information for this mapping to be fully generated: the lower level formalisms to project onto, which parts of DSms to project and how to project them are implicitly encoded in the STs that define their DSL (see Section 6 for concrete examples). Thus, the artifact synthesis pipeline now consists in automatically projecting implicitly specified (and possibly non-disjoint) portions p_i of DSms onto appropriate base formalisms f_i from which the desired artifacts a_i (e.g., Java code) can be generated. A simplistic yet effective approach is to generate instances of the appropriate base formalisms from DSms using the information contained within the STs – which by definition is sufficient to do so –. We achieve this using automatically synthesized model transformations that reflect the STs. This approach has the added advantage that it enables the maintenance of traceability links between entities at various levels of abstraction. In [13], we argue this facilitates tasks such as model animation and debugging.

Our approach shares the caveat of the reviewed template instantiation and semantic anchoring techniques: tool support is paramount. The array of possible targets for artifact synthesis from DSms is only as wide as the array of available targets for artifact synthesis from instance models of base formalisms. Luckily, these were selected among very popular and longstanding languages which have received considerable attention by tool developers and researchers alike and for which numerous compilers to various targets already exist. Reusing these however might well be prevented by difficulties commonly encountered during tool integration endeavors; namely, incompatible data formats and insufficient APIs. Nevertheless, these formalisms and their compilation remain well understood and well documented. Reimplemented versions of the said base formalism compilers within tools that support our approach to DSL design will be transparently usable for any DSM effort with no further attention paid to tool integration. Thus, the potential for effort reuse and raise of abstraction under our approach is enormous.

In explaining our solution to artifact synthesis, we mentioned that for each projection p_i of a DSm onto a base formalism f_i, artifacts a_i are generated, and that these artifacts are instrumented with message passing facilities to communicate with each other. This presents further questions. How and when can artifacts communicate with each other? What type of information and/or commands will given artifacts be responsive too? To address these challenges, we push the idea of "STs as interfaces to base formalisms" into a new and orthogonal direction. Base formalisms are no longer described solely by a set of STs that enable mapping higher level concepts onto them, they are also characterized by a collection of input and output *events*. Table 2 shows a tentative list of events produced and consumed by each base formalism. A single additional generic template now suffices to capture how synthesized artifacts should emit and respond to these events: *On event `e1`, produce event `e2`*, where both events need not be associated to the same artifact. Modellers and metamodellers can now describe how artifacts generated from the behavioural and layout components of a DSm should communicate (as shown in Section 6). The final piece of the puzzle is for events emitted by one artifact to be properly received by its intended

[6]This assumes that base formalisms are tagged with the features they provide and that means to visualize these tags exist.

[7]Note that though we isolate both components of our solution here, their practical implementations were entangled.

Formalism	Semantic Template	Impact
Statecharts	`A` transition to `B`	Creates classes for `A` and `B` if they don't already exist. "Imports" the class diagram for Statecharts and creates inheritance relationships between `A` and `Statecharts.State` and between `B` and `Statecharts.State` if they don't already exist. Among many other things, the latter implies that instances of classes `A` and `B` in DSms can be connected via `Statecharts.Transition`s.
Petri Nets	`A` have `(finite | infinite)` capacity `(k)`	Creates a class for `A` if it doesn't already exist. Imports the class diagram for Petri Nets and creates an inheritance relationship between `A` and `PetriNets.Place`. The inherited `PetriNets.Place.capacity` attribute is either set to *infinite* or to *k*.
<generic>	`A` are types of `B`	Creates classes for `A` and `B` if they don't already exist. Creates inheritance relationships between `A` and `B`.

Table 1: Sample base formalism and generic semantic templates.

Formalism	Produces	Consumes
Statecharts	`enteredState:s`, `exitedState:s`	`handleEvent:e`
Petri Nets	`transitionFired:t`	`fireTransition:t|*`
CBDs	`outputChanged:[o,v]`	`setInput:[i,v]`
Layout	`guiEventFired:e`	`drawCanvas:c`
Action Code	`codeReturned:r`	`runCode:c`

Table 2: A tentative list of events produced and consumed by each base formalism.

recipient(s). We addressed this issue in the past by weaving a message passing infrastructure into generated artifacts. This infrastructure is essentially composed of an *event manager* which is aware of the artifacts and whom the artifacts are aware of. In short, at execution time, events are sent to this event manager who then dispatches them appropriately.

We have explained which modelling formalisms form a basis for DSL design, how to define new DSLs in terms of the formalisms in that basis and how complete artifacts can be generated from instance models of these DSLs without their designers having to manually define their semantics. The following section demonstrates our approach in a concrete example.

6. CASE STUDY

In [13], we introduced *PhoneApps*, a DSL for modelling mobile device applications that captures both the visual interface and behavioural concerns of such applications[8]. Figure 3 shows its metamodel. Essentially, timed, conditional and user-prompted transitions describe the flow of control between `Container`s – that can contain other `Container`s and `Widget`s – and `Action`s – mobile device specific features (e.g., sending text messages, dialing numbers) – with each screen in the synthesized application modelled as a *top-level* `Container` (i.e., a `Container` contained in no other). With a series of manually defined graph transformations, PhoneApps models were translated to increasingly lower level formalisms until a complete Google Android [1] application was synthesized. In this section, we describe how the *PhoneApps* metamodel can be redefined in terms of base formalism STs.

[8]Our DSL is strongly inspired by that presented in [12].

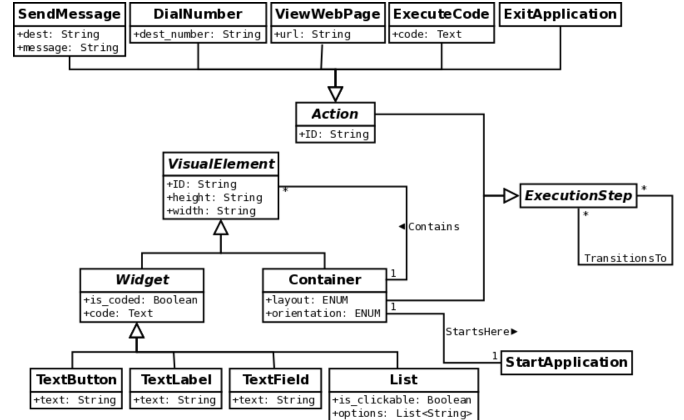

Figure 3: The *PhoneApps* metamodel (as a UML class diagram).

The *PhoneApps* language combines three basic components: visual interface, behaviour and mobile device functions. These can be respectively captured using the *Layout*, Statecharts and *Action Code* base formalisms. A more domain-specific approach – which more closely reflects the version of the DSL presented in [13] – would be to model mobile device features at a higher-level of abstraction than coded function calls. To this end, we introduce a new formalism, *Google Android API*. We view this formalism as a third-party (e.g., Google) provided semantically-templated base formalism. It also serves the purpose of demonstrating how arbitrary new languages can graft themselves onto the list of available base formalisms. Table 3 shows a possible reconstruction of the *PhoneApps* DSL using STs.

A class diagram very similar to that shown in Figure 3 can trivially be generated from Table 3 using rules similar to those from Table 1. We refrain from showing it to reassert the fact that a set of STs can and does fully capture a DSL (and more). Indeed, Table 3 defines the abstract syntax *and* the semantics of the *PhoneApps* metamodel. The STs contain all the required information for communicating instances of the four base formalisms in play to be generated. Their semantics transformations being defined – as they are instances of base formalisms –, the semantics of any *PhoneApps* model is also defined. Thus, the manual definition of projections and mappings of PhoneApps models onto lower level formalisms is no longer necessary.

Formalism	Template
<*generic*>	`Actions` and `Screens` are types of `Steps`.
Statecharts	`Steps` transition to `Steps`.
Layout	`Screens` are canvases.
<*generic*>	`Dials`, `SMSs`, `Browses` and `UserCodes` are types of `Actions`.
G. Android API	`Dials` make phone calls.
G. Android API	`SMSs` send text messages.
G. Android API	`Browses` open browsers.
Action Code	`UserCodes` are coded.
<*generic*>	On event `enteredState:s`, produce event `drawCanvas:s`.
<*generic*>	On event `enteredState:s`, produce event `runCode:s`.

Table 3: The *PhoneApps* metamodel (as a set of semantic templates).

7. CONCLUSION AND FUTURE WORK

We proposed a novel approach to defining domain-specific languages based on the combination of low level formalisms that capture commonly recurring DSL features. These *base formalisms* expose a set of *semantic templates* and *events* that enable the full synthesis of communicating base formalism instances from domain-specific models. Given the existence of "base formalism to target artifact" transformations, DSms can be transformed to the said target artifacts without the DSL designer having to manually define DSL semantics (e.g., via coded generators and/or model transformations). Our approach to DSL engineering adheres closely to the model-driven engineering philosophy of development using domain-specific concepts by privileging the use of elementary language constructs over that of generic object-oriented constructs. Furthermore, we improve on related techniques by enabling the *full* generation of DSL semantics – as opposed to only their partial generation – and by rooting language design in well understood and studied formalisms (e.g., Statecharts, Petri Nets).

The *basis* for language design we propose might not be complete. A wise first step towards completing it might be to categorize – ideally all – existing formalisms into a minimal set of distinct "classes of formalisms". It is our hope that future work will address any lackings our basis may have as the array of expressible DSLs under our approach is only as wide as the expressiveness of the said basis. We have prototyped the concepts described in this paper in our tool for multi-formalism and meta-modelling, AToM^3 [8] (which natively supports the Statecharts, Petri Nets and Causal Block Diagrams formalisms as well as many others). We will fully integrate our proposed ST-based approach to DSL design in AToM^3's successor, AToMPM.

8. REFERENCES

[1] Google android. http://code.google.com/android/.

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

[3] 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.

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] Raphael Mannadiar and Hans Vangheluwe. Modular synthesis of mobile device applications from domain-specific models. In *The 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, 2010.

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

[15] 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.

[16] 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, 2010.

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