Modular Synthesis of Mobile Device Applications from Domain-Specific Models

Raphael Mannadiar, Hans Vangheluwe
School of Computer Science
McGill University
rmanna@cs.mcgill.ca, hv@cs.mcgill.ca

Abstract

Domain-specific modelling enables modelling using constructs familiar to experts of a specific-domain. Domain-specific models (DSms) can be automatically transformed to various lower-level artifacts such as configuration files, documentation and executable programs. Although many researchers have tackled the formalization of various aspects of model-driven development such as model versioning, debugging and transformation, very little attention has been focused on formalizing how programs are actually synthesized from DSms. State-of-the-art approaches rely on ad hoc coded generators which essentially use modelling tool APIs to programmatically iterate through model entities and produce the final artifacts. In this work, we propose a more structured approach to artifact generation where layered model transformations are used to modularly isolate, compile and re-combine various aspects of DSms.

1 Introduction

Domain-specific languages (DSLs) allow non-programmers to play an active role in the development of applications. This makes obsolete 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 –. Furthermore, due to 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 concepts –, domain-specific models (DSms) can be automatically transformed to complete executable programs. This truly raises the level of abstraction above that of code. Empirical evidence suggests increases in productivity of up to an order of magnitude when using domain-specific modelling (DSM) and automatic program synthesis as opposed to traditional code-driven development approaches [15, 12, 14].

Due to the very central part played by automatic code synthesis in DSM, we argue that structuring how models are transformed into code is both beneficial and necessary. Previous work realizes the said transformation by means of ad hoc hand-coded generators that manipulate tool APIs, regular expressions and dictionaries [15, 12, 17]. In constrast, our approach employs modular and layered visual graph transformations whose results can readily be interpreted. Section 2 briefly overviews related work. Section 3 introduces a DSL we developed for modelling mobile device applications as well as the transformations and lower level formalisms that make up the layers between the DSms and the generated applications. In Section 4, we present a non-trivial instance model of our DSL, every stage of the transformation to code and the synthesized application running on a Google Android [2] device. Finally, in Section 5, we discuss future work and provide some closing remarks.

2 Related Work

Most of current research in the general area of model-driven engineering focuses on enabling modellers with development facilities equivalent to those from the programming world. Most notably, these include designing/editing [9, 5, 4], differencing [3, 7, 13], transforming [8, 16], evolving [6] and debugging models [17]. However, more hands-on research has explored the complete DSM development process; from the design and creation of a DSL suited to a specific problem in a specific domain to the synthesis of the required artifacts [15, 12, 14]. In these works, DSms are systematically transformed to lower level artifacts by means of hand-coded generators [15, 12]. In [17], Wu et al. enhance such a code generator with provisions to instrument generated code and build a mapping between models and artifacts to enable debugging at the DSm level.

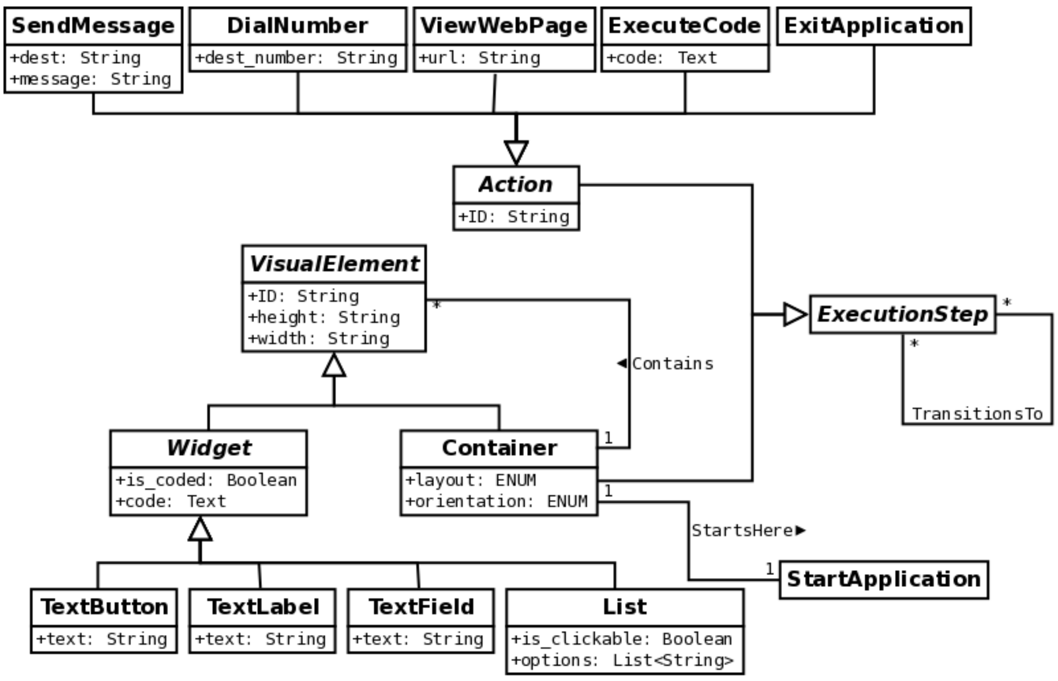


Figure 1: The PhoneApps meta-model (as a Class Diagram).

Several academic and non-academic works have investigated, in some form or another, the modelling and synthesis of mobile device applications. In [14], a meta-model for modelling mobile device applications is introduced where behaviour and user interface elements are intertwined. In [15], a meta-model for home automation device interfaces with provisions for escape semantics[1] is described. The meta-model we introduce here was inspired by these two formalisms and is in fact a combination and enhancement. In [1], a web-based applet enables the graphical modelling of the visual components of Google Android application screens and the generation of the equivalent XML code required to render the modelled screen. Although this tool can not be used to model application behaviour, it is a good starting reference as to which features are required for a DSL which aims at specifying both the appearance and behaviour of a mobile device application.

3 Meta-Models and Transformations

3.1 PhoneApps

Mobile device applications often require high levels of user interaction. It can thus be argued that behaviour *and* visual structure make up the domain of such applications. The *PhoneApps* DSL encompasses both of these aspects at an appropriate level of abstraction (see Figure 1). 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 final application modelled as a *top-level* `Container` (i.e. a `Container` contained in no other).

With a series of graph transformations, PhoneApps models are translated to increasingly lower level formalisms until a complete Google Android application is synthesized. Figure 2 gives an overview of the hierarchical relationships between the meta-models in play. The following subsections detail each transformation.

[1]Means to extend the modelling language's expressiveness are built into the language itself.

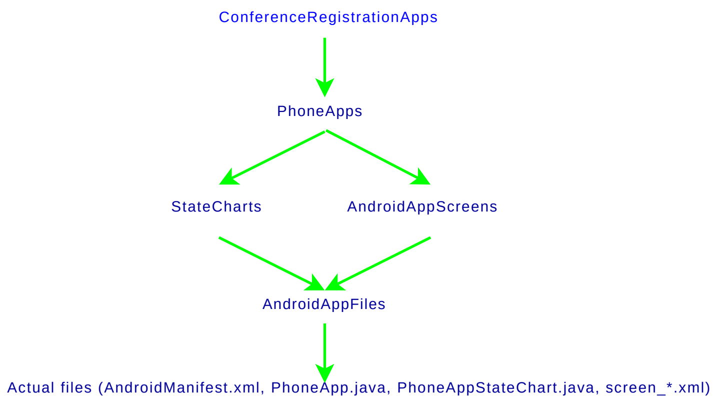

ConferenceRegistrationApps
PhoneApps
StateCharts
AndroidAppScreens
AndroidAppFiles
Actual files (AndroidManifest.xml, PhoneApp.java, PhoneAppStateChart.java, screen_*.xml)

Figure 2: A Formalism Transformation Graph [9] for PhoneApps.

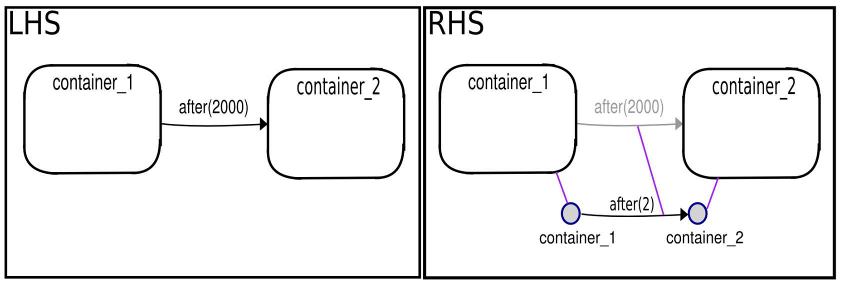


Figure 3: A PhoneApps timeout mapped to a Statechart timeout. The grayed out transition between `Containers` illustrates the marking of that transition as "*seen*" by the rule's action code.

3.2 PhoneApps-to-Statecharts

Rather than attempt to invent a novel way of transforming and generating code for complex behaviour, we used the extensively proven and studied formalism of Statecharts [11] as the target formalism for the behavioural components of PhoneApps models. These components are fully encompassed in the edges between `Container`s and `Action`s and can readily be mapped onto Statecharts. Existing tools for Statechart compilation, simulation, analysis, etc. can be exploited to produce efficient and correct code. Figure 3 shows an example graph transformation rule[2] from a subgraph of a PhoneApps model to its equivalent in the Statechart formalism. Essentially, when the *PhoneApps-to-Statechart* transformation (see Table 1) has run its course, every `Container` and `Action` has a corresponding state and these states are connected and customized according to the edges that connect their corresponding `Container`s and `Action`s. Note that the behaviour described by the current PhoneApps formalism requires only the expressiveness of timed automata. Future work will extend possible behaviour such that more powerful Statechart features such as orthogonality and nesting will be required.

3.3 PhoneApps-to-AndroidAppScreens

After isolating and transforming the behavioural components of PhoneApps models to Statecharts, another transformation is required to isolate and transform their user-interface and Google Android related components. The formalism we propose to encompass this information is *AndroidAppScreens* (see Figure 4). Essentially, top-level

[2]A rule is parameterized by a left-hand side (LHS), a right-hand side (RHS) and optionally a negative application condition (NAC) pattern, condition code and action code. The LHS and NAC patterns respectively describe what sub-graphs should and shouldn't be present in the source model for the rule to be applicable. The RHS pattern describes how the LHS pattern should be transformed by the application of the rule. 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.

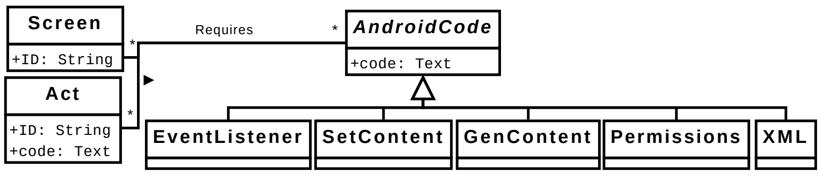


Figure 4: The AndroidAppScreens meta-model.

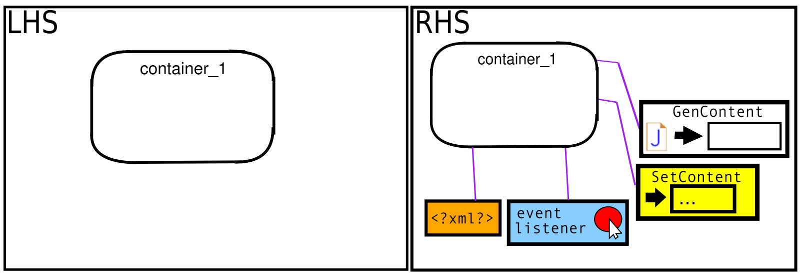


Figure 5: Extracting information from a `Container` into new AndroidAppScreens constructs.

`Containers` and `Actions` are replaced by `Screens` and `Acts` which are appropriately connected to some number of constructs that represent snippets of Google Android-specific code. For instance, `XML` constructs contain the XML code required to render the screen (with its specified layout and widgets) on a Google Android enabled device; `Permission` constructs contain code required to indicate which restricted actions an `Action` might accomplish (e.g., access the address book, access the internet); `GenContent` constructs contain modeller specified Java code, if any, that will compute information to display at runtime (e.g., dynamic `List` elements). Figure 5 shows an example translation rule from a subgraph of a PhoneApps model to its equivalent in the AndroidAppScreens formalism. See Table 2 for an overview of the full transformation.

The PhoneApps-to-Statecharts and PhoneApps-to-AndroidAppScreens transformations clearly demonstrate the modular and layered nature of our approach to code synthesis. Our approach improves upon the traditional ad-hoc hand-coded generator approach on numerous fronts.

First, although in a finished product, the inner workings that convert PhoneApps models to Google Android applications should be hidden from the modeller, it is useful – at the very least for educational purposes – to see how higher- and lower-level models are related. This fact is demonstrated in Figures 12 and 13.

Second, the recurringly stated goals of simulation and debugging at the DSm level can be achieved by instrumenting the generated code with appropriate callbacks as in [17]. Unfortunately, in doing so, the code generator is polluted by considerable added complexity. In our approach, the complex task of maintaining backward links between models and synthesized artifacts is accomplished by connecting higher-level entities to their corresponding lower-level entities in our transformation rules via *generic edges*[3]. These have a minimal impact on the readability of our rules and their specification is amenable to semi-automation. The resulting chains of generic edges can be used to seamlessly animate and update DSms (or any intermediate models) during execution of the synthesized application [4]. Furthermore, these links can aid in the debugging of the "code synthesis engines" themselves. For instance, complex tasks such as determining what was generated from which model entity become trivial and don't require any further instrumentation.

Third, the step-by-step execution of transformation rules provides a free debugging environment where one can very easily observe (and modify) the effect of every single rule in isolation. Once again, this is considerably easier,

[3]Notice the purple, undirected edges between constructs of different formalisms in the figures describing rules
[4]Future work will explore extending these animation capabilities to more advanced debugging activities such as altering the execution flow

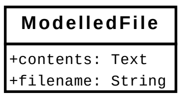


Figure 6: The AbstractFiles meta-model.

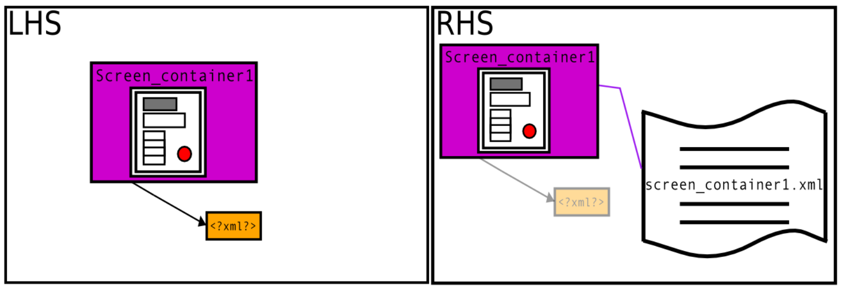


Figure 7: Creating one `ModelledFile` per `Screen` to hold its XML layout specification.

more modular and more elegant that lacing a coded generator with print statements or breakpoints.

Finally, the most important advantage of our approach is perhaps that it raises the level of abstraction of the design of code synthesis engines. Rather than interacting with tool APIs and writing complex code, the task of implementing a code generator is reduced to specifying relatively simple graphical model transformations rules using domain-specific constructs. In a research community that ardently encourages the use of models and modelling and more generally development at a proper level of abstraction, our approach to code synthesis seems like a natural and logical evolution.

3.4 AndroidAppScreens/Statecharts-to-AbstractFiles

Certain benefits of keeping backward links between models and generated artifacts were discussed in the previous subsections. To accomplish this with the same level of modularity and elegance as was achieved by our previous transformations, we introduce the *AbstractFiles* formalism. This trivially simple formalism (see Figure 6 for its meta-model) serves as an abstraction of the actual generated files i.e. a model element exists for each generated file. Hence, rather than compile and output the previously generated AndroidAppScreens and Statechart models directly to files on disk, their compilation results in an instance model of the AbstractFiles formalism. An added benefit of this design choice is that the generated output for each file can be reviewed from within the modelling environment as part of the debugging process: there is no more need to locate files on disk and open them in a separate editor. Figures 7 and 8 show two example transformation rules from subgraphs of an AndroidAppScreens model to their equivalent in the AbstractFiles formalism. See Table 3 for an overview of the full transformation and Table 4 for the trivial transformation between AbstractFiles to files on disk. As for the transformation of the Statechart constructs to the AbstractFiles formalism, the output of a Statechart compiler is directed towards an AbstractFiles model element to be later output to a Java file on disk.

3.5 ConferenceRegistrationApps-to-PhoneApps

This meta-model is even more domain-specific than PhoneApps: it is specific to conference registration applications on mobile devices. It was designed as a proof of concept to show that higher level and more tightly constrained models can be built on top of the PhoneApps formalism. See [10] for a more advanced example of this where models of privacy-preserving eServices are used to generate Google Android applications by first translating them to equivalent PhoneApps models. With *ConferenceRegistrationApps*, the tasks of assigning IDs to `VisualElements`, of explicitly defining screens (via `Containers`) and containment edges between `Containers` and their contents, and of textually specifying user event triggers is removed at the cost of flexibility in screen and application design. Model elements are in fact barely customizable screengrabs of a synthesized Google Android conference registration application that

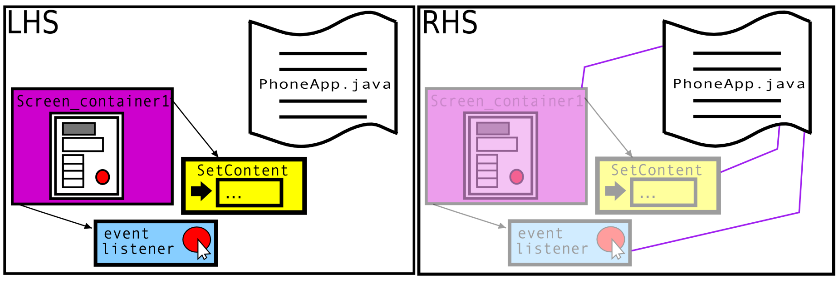


Figure 8: Appending event listener and content initialization code to a `ModelledFile` of the main Java artifact "PhoneApp.java".

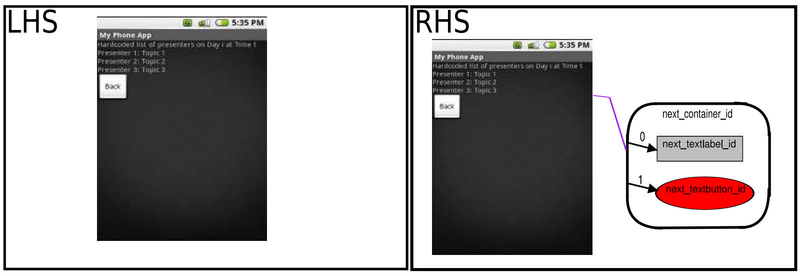


Figure 9: A content display screen is mapped to equivalent PhoneApps constructs.

was first modelled and generated via the PhoneApps formalism. A conceptually similar meta-model is presented in [15] for designing home automation device interfaces. Figures 9 and 10 show example transformation rules from subgraphs of a ConferenceRegistrationApps model to their equivalent in the PhoneApps formalism. See Table 5 for an overview of the full transformation.

4 Case study: Conference Registration in PhoneApps

We now present a hands-on example to give more insight on the full scale model transformations and intermediate representations involved in synthesizing a Google Android conference registration application from a PhoneApps model. Of particular interest is the intuitive mapping between each model and its counterpart(s) in the lower-level formalisms. The following will explicit the above-mentioned advantages of easier debugging and readability of the code synthesis engine that result from our approach.

Figure 11 shows the modelled conference registration application CR in the PhoneApps formalism.There are 3

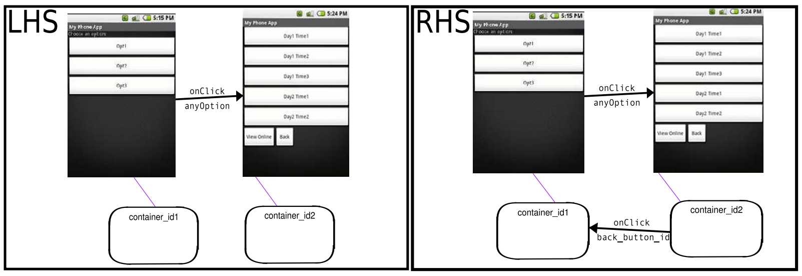


Figure 10: A user prompted PhoneApps transition is created to model the click of a screen's "Back" button.

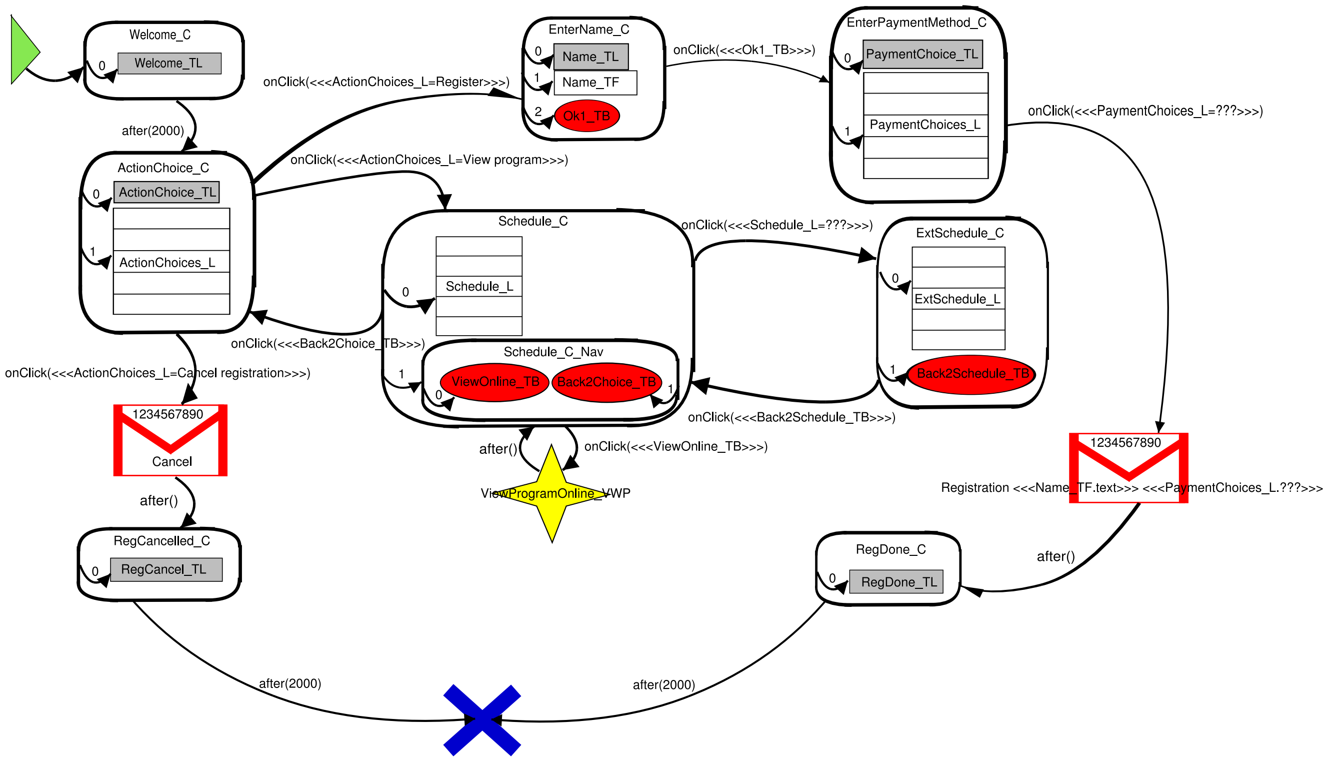


Figure 11: Conference registration in PhoneApps.

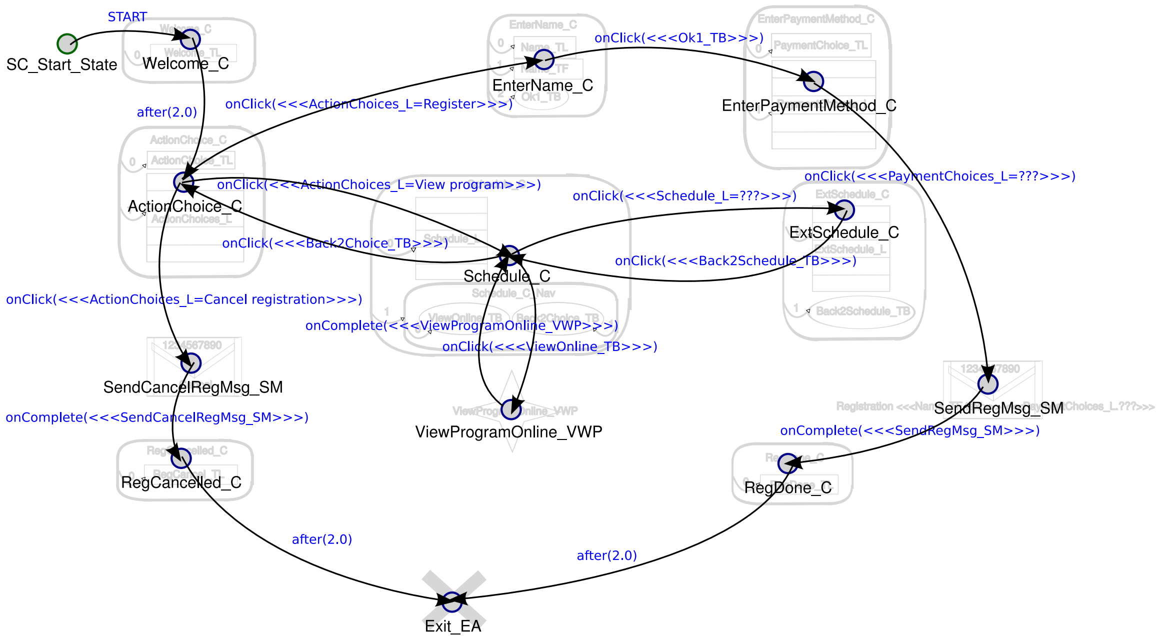
main use cases: (1) registering, (2) viewing the programmed schedule and (3) canceling a registration. The first is explored below. The two non-detailed use cases are as intuitive as the first which seems to indicate that the proposed meta-model is not only easy to learn but also at the appropriate level of abstraction for this sort of application.

1. The user sees the *Welcome* screen for 2 seconds and is then taken to the *ActionChoice* screen;
2. The user clicks on "Register" on the *ActionChoice* screen and is taken to the *EnterName* screen;
3. The user enters a name, clicks "OK" 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 hardcoded phone number after which the user is taken to the *RegistrationDone* screen;
5. The user sees the *RegistrationDone* 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.

Figure 12 shows the model after the PhoneApps-to-Statecharts transformation has run its course[5]. Essentially, every transition in the PhoneApps model, user or delay prompted, has been translated to a Statechart transition between states that represent the original `Actions` and `Containers`. Not visible are the entry actions of each state which dictate the application's behaviour through function calls to generated methods that carry out tasks on the mobile device such as loading a screen, sending a text message, etc.

Figure 13 shows the model after the PhoneApps-to-AndroidAppsScreens transformation has run its course. Essentially, every `Container` and `Action` has been translated to appropriate elements of the AndroidAppsScreens formalism. The Java code that will carry out the `Actions` from the PhoneApps model is encapsulated in `Act` entities

[5]For clarity, we refrain from reproducing the entire *CR* model and generic edges between it and generated constructs in Figures 12, 13 and 14. Instead, we overlay corresponding constructs.



START
SC_Start_State
Welcome_C
after(2.0)
onClick(<<<ActionChoices_L=Register>>>)
EnterName_C
onClick(<<<Ok1_TB>>>)
EnterPaymentMethod_C
ActionChoice_C
onClick(<<<ActionChoices_L=View program>>>)
onClick(<<<Schedule_L=???>>>)
onClick(<<<PaymentChoices_L=???>>>)
ExtSchedule_C
onClick(<<<Back2Choice_TB>>>)
Schedule_C
onClick(<<<Back2Schedule_TB>>>)
onClick(<<<ActionChoices_L=Cancel registration>>>)
onComplete(<<<ViewProgramOnline_VWP>>>)
onClick(<<<ViewOnline_TB>>>)
SendCancelRegMsg_SM
SendRegMsg_SM
onComplete(<<<SendCancelRegMsg_SM>>>)
ViewProgramOnline_VWP
onComplete(<<<SendRegMsg_SM>>>)
RegCancelled_C
RegDone_C
after(2.0)
after(2.0)
Exit_EA

Figure 12: The *CR* model after applying the PhoneApps-to-Statecharts transformation.

while the code that will carry out the appropriate display of a screen and its contents is encapsulated in `Screen`, `XML`, `EventListener`, `SetContent` and `GenContent` entities. Instances of these entities are comparable to visual representations of the coded lists and dictionaries found in other works.

Figure 14 shows the model after the AndroidAppsScreens-to-AbstractFiles and the Statecharts-to-AbstractFiles transformations have completed[6]. These two transformations output to a disjoint set of AndroidAppsFiles model elements and can thus be run in parallel. Their results are presented together to illustrate the merging of the previously isolated conceptual components into a single, final target formalism. The former transformation iteratively moves through the various instances of AndroidAppsScreens model elements outputting and appending their contents more or less directly to the `ModelledFile`s that will become PhoneApp.java, AndroidManifest.xml and screen_*.xml. The latter compiles the Statechart into the `ModelledFile` that will become PhoneAppStateChart.java.

The final step is the trivial transformation of the `ModelledFile`s to real files on disk. The end result of this series of transformations is a fully functional Google Android application that perfectly reflects the original PhoneApps model; see Figure 15.

The obvious similarities between the synthesized application and its domain specific model are a testament to the fact that the `PhoneApps` meta-model is indeed at the proper level of abstraction for the development of mobile device applications. However, if one wanted to exclusively model conference registration applications, the aforementioned ConferenceRegistrationApps formalism might be at an even more appropriate level. Figure 16 displays the same example application modelled using ConferenceRegistrationApps.

5 Conclusion and Future Work

We proposed a structured approach to code generation where layered model transformations are used to modularly isolate, compile and re-combine various aspects of DSms. We argued that our approach improves upon the traditional

[6]Remember that though they are hidden here, numerous generic edges connect the `ModelledFile`s to Statechart and AndroidAppsScreens constructs

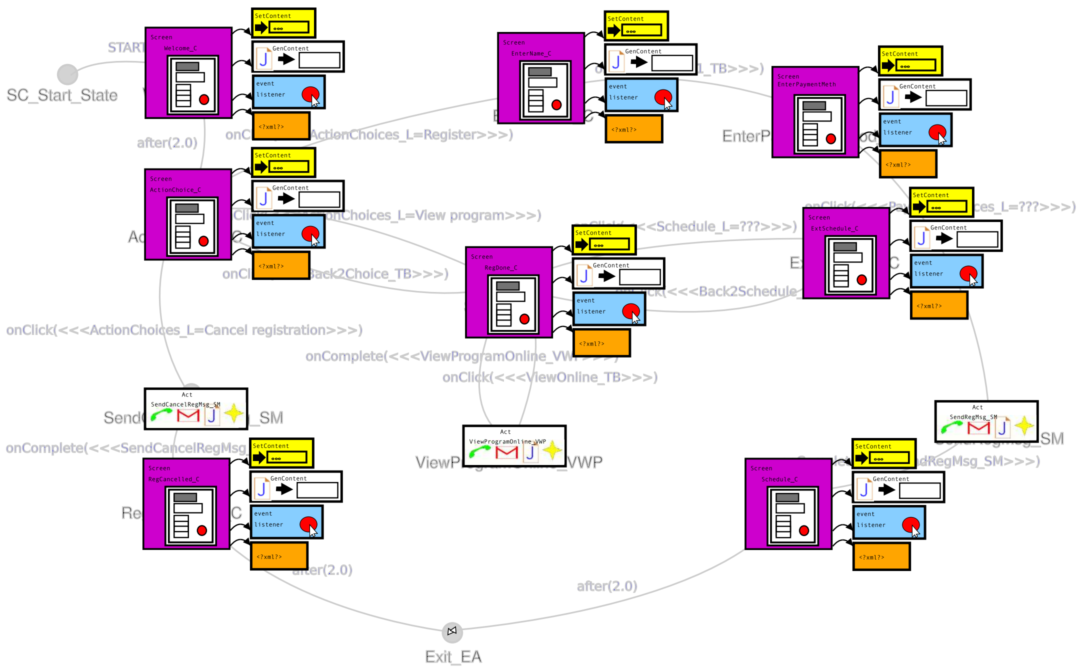


Figure 13: The *CR* model after applying the PhoneApps-to-AndroidAppsScreens transformation.

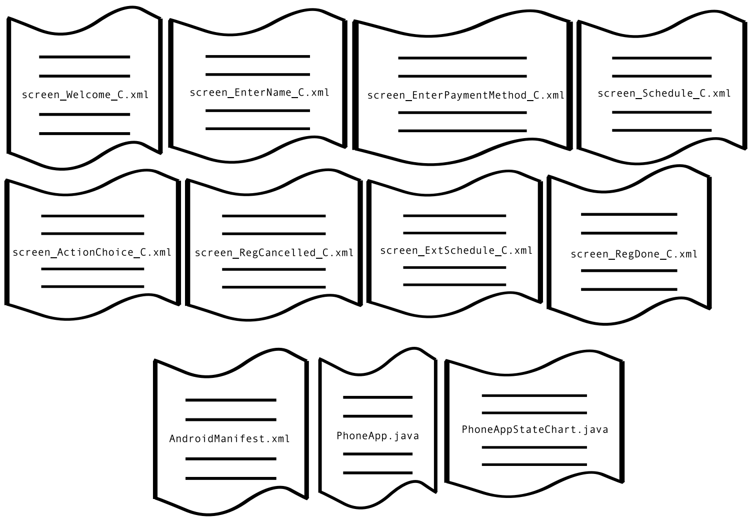
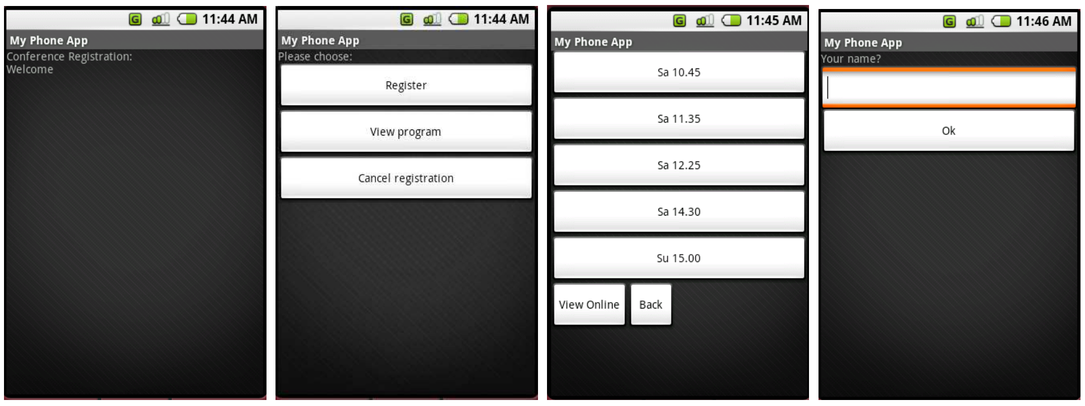


Figure 14: The *CR* model after applying the AndroidAppsScreens-to-AbstractFiles and Statecharts-to-AbstractFiles transformations.



(a) *Welcome* (b) *ActionChoice* (c) *ProgramSchedule* (d) *EnterName*

Figure 15: Screens of the synthesized application running on a Google Android emulator.

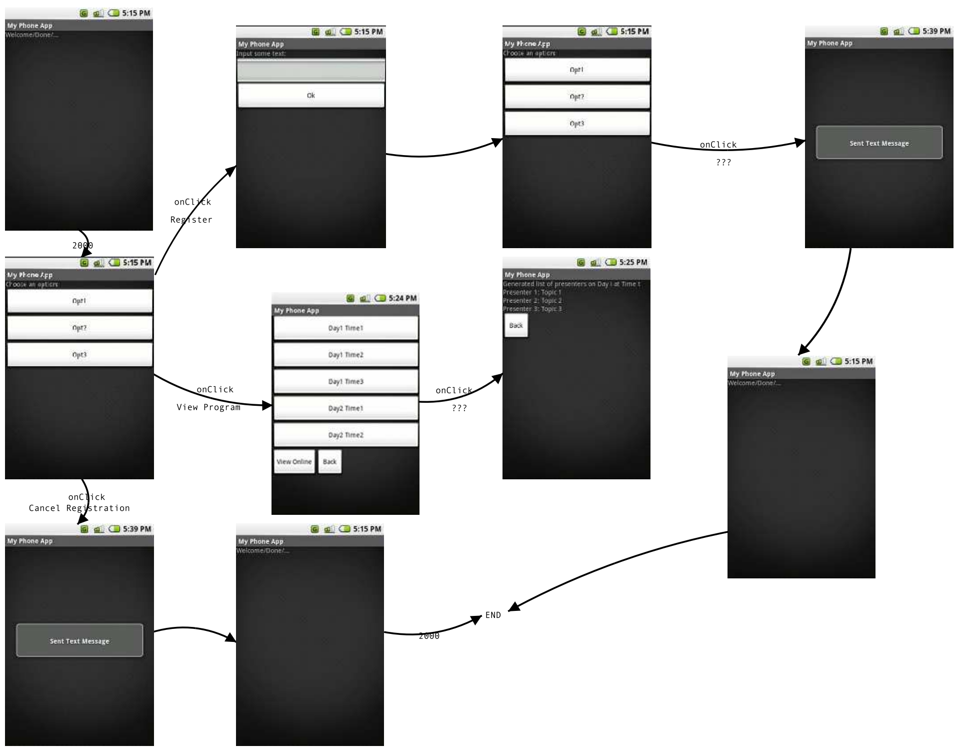


Figure 16: *CR* modelled in the ConferenceRegistrationApps formalism.

ad-hoc coded generator approach to synthesizing applications from DSms by easing debugging of the code synthesis engine, providing a clear picture of the real and conceptual links between constructs at different levels of abstraction, and easing and improving the modularity of the implementation of advanced facilities such as DSm simulators and debuggers. Finally, we argued that visually modelling code synthesis and moving away from code and tool APIs adheres well to the model-driven engineering philosophy.

The DSLs, transformations and case study we presented in this work provided some empirical evidence to back our claims that (graphical) model transformations are a better means of generating code from DSms than coded generators. However, our approach still requires some formalization. Since we essentially *ripped* and *weaved* DSLs with our transformations, we believe that the first step towards this formalization is the study of the broader ideas and theory of DSL *weaving* and *ripping* specifically during DSL design. For instance, combining some form of explicit DSL concept generalization relationship (e.g., `PhoneApps.ExecutionStep` *is a* `Statechart.State` and `PhoneApps.TransitionsTo` *is a* `Statechart.Edge`) with higher-order transformations[7] could enable the automation of much of the above work (e.g., part or all of the PhoneApps-to-Statechart transformation could be generated automatically). As a final benefit to our approach, although this idea is very promising in the context of (semi-)automatically generating transformation rules, generating equivalent parts of a coded generator would require considerably more effort and likely produce a complex and incomplete program that would be difficult to understand let alone complete and maintain.

References

[1] Droiddraw. http://www.droiddraw.org/.

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

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

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

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

[6] 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.

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

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

[9] 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.

[10] 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.

[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. 427 pages.

[13] 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.

[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.

[7]Transformations that take other transformations as input and/or outputs.

[16] Yu Sun, Jules Whit, and Jeff Gray. Model transformation by demonstration. In *MODELS*, volume LNCS 5795, pages 712–726, 2009.

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

Appendix

Priority	Description
1	Match a "not yet visited" `TransitionsTo` edge between two PhoneApps `ExecutionSteps` $N1$ and $N2$. Create two states named $N1.ID$ and $N2.ID$, connect them via edge E and set appropriate trigger, action and guard values for E.
2	Match two states with identical names Merge them and redirect edges appropriately.
3	Match the unique instance of `StartApplication` in the PhoneApps model and the `Container` C to which it is connected. Create a state named "SC_Start_State" that transits to the state named $C.ID$ on reception of event "START".

Table 1: An overview of the rules that make up the *PhoneApps-to-Statechart* transformation along with their execution priorities. In the AToM^3 [9] transformation execution environment, rules with lower priority run first and a rule will execute repeatedly until it is no longer applicable (i.e. its LHS pattern can not be matched or its firing condition fails).

Priority	Description
1	Match a not yet visited `Widget` W. Connect W to new instances of `EventListener`, `SetContent`, `GenContent` and `XML` and populate these according to W's parameters.
1	Match a not yet visited `Action` A. Connect A to a new instance of `Act` (connected, if necessary, to an appropriately filled instance of `Permissions`) and populate it with the Google Android Java code to carry out the `Action`.
2	Match a not yet visited `Container` C. Connect C to new instances of `EventListener`, `SetContent`, `GenContent` and `XML` and populate these according to C's parameters *and* according to the contents of the `EventListener`, `SetContent`, `GenContent` and `XML` instances associated with C's contained `VisualElements`.
3	Match a not yet visited top-level `Container` C. Connect C to a new instance of `Screen` S and redirect the edges between C and its instances of `EventListener`, `SetContent`, `GenContent` and `XML` towards S.

Table 2: An overview of the steps that make up the *PhoneApps-to-AndroidAppScreens* transformation.

Priority	Description
1	Initialize two `ModelledFile`s with filenames “PhoneApp.java” and “AndroidManifest.xml”. This initialization entails setting the static portion of their content e.g., for the “PhoneApp.java” `ModelledFile`, the content attribute is initialized with package information as well as import statements and so on.
2	Match a not yet visited `XML` connected to `Screen` S. Output its contents to a new instance of `ModelledFile` with filename attribute set to $S.ID$.
2	Match a not yet visited `Permissions`. Append its content to that of the “AndroidManifest.xml” `ModelledFile`.
2	Match a not yet visited `Act`. Append its content to the “PhoneApp.java” `ModelledFile`.
3	Match a not yet visited `Screen` S, its `EventListener` E and its `SetContent` T. Generate a Java function that contains the contents of E and T and more and append it to the “PhoneApp.java” `ModelledFile`. This function will be called from within the synthesized application to set the mobile device’s display to the Google Android equivalent of S.
4	Match a not yet visited `GenContent`. Append its content to that of the “PhoneApp.java” `ModelledFile`.
5	Finalize the “PhoneApp.java” and “AndroidManifest.xml” `ModelledFiles`. This finalization entails closing brackets and tags.

Table 3: An overview of the steps that make up the AndroidAppScreens to AbstractFiles transformation.

Priority	Description
1	Match a not yet visited `ModelledFile` MF. Output $MF.contents$ to a file on disk named $MF.filename$.

Table 4: An overview of the steps that make up the transformation that convert AbstractFiles to actual files.

Priority	Description
1	Connect a ConferenceRegistrationApps model element instance (visually represented by a screengrab of a synthesized Google Android application) to its equivalent representation in the PhoneApps formalism.
1	Connect an `End` to an `ExitApplication`.
2	Match two ConferenceRegistrationApps model element instances, $V1$ and $V2$, that represent screens that are semantically connected by a “Back” button. Create a `TransitionsTo` edge between the `Container`s representing $V1$ and $V2$ with a trigger corresponding to a click on the “Back” button.
2	Add a `StartApplication` and connect it to the `Container` that represents the ConferenceRegistrationApps model element instance that was marked by the modeller as the initial screen.
3	Match two ConferenceRegistrationApps model element instances, $V1$ and $V2$ that are connected by an edge E representing some user or delay prompted event. Create a `TransitionsTo` edge between the `Containers/Action`s representing $V1$ and $V2$ with a trigger corresponding to the event encoded in E.

Table 5: An overview of the steps that make up the ConferenceRegistrationApps-to-PhoneApps transformation.