Modular Synthesis of Mobile Device Applications from Domain-Specific Models

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

Hans Vangheluwe
McGill University
3480 University Street
Montreal, Quebec, Canada
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, executable programs and performance models. 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 artifacts 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. We demonstrate our technique by detailing the synthesis of running Google Android applications from DSms, and discuss how it may be applied in addressing the characteristic non-functional requirements (e.g. timing constraints, resource utilization) of modern embedded systems.

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

Multi-paradigm modelling, Model transformations, Language ripping and weaving, Application synthesis, Performance metric synthesis, Google Android

MOMPES '10, September 20, 2010, Antwerp, Belgium

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[1]) and automatic program synthesis as opposed to traditional code-driven development approaches [15, 11, 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 code generators that manipulate tool APIs, regular expressions and dictionaries [15, 11, 18]. 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 have developed for modelling mobile device applications as well as the lower level formalisms and transformations that make up and produce 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 [1] device. In Section 5, we compare the traditional approach to artifact synthesis to our own in the context of generating and presenting non-functional requirement related performance information from DSms. Finally, in Section 6, we discuss future work and provide some closing remarks.

[1]Note that we refer to domain-specific modelling as DSM and to a domain-specific model as a DSm.

2. RELATED WORK

In this section, we briefly discuss related research on three topics: artifact synthesis from DSms, modelling mobile device applications and integrating performance concepts in early development phases.

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, 4, 3], differencing [2, 6, 12], transforming [8, 16], evolving [5] and debugging models [18]. More hands-on research has explored the complete DSM development process starting from the design of DSLs to the synthesis of required artifacts from instance models. In these works, DSms are systematically transformed to lower level artifacts by means of ad hoc hand-coded generators [15, 11, 14]. However, no allusions are made regarding model debugging, simulation, tracing or any other activity where it might be desirable to establish links between model and artifact. Wu et al. [18] recognized this need for the context of DSm debugging and proposed a generic grammar-based technique for generating DSL debuggers that reuse existing integrated-development environment facilities. Unlike previous work, mapping information from model to code is computed and stored during code generation. This information is then used to enable common debugging activities such as setting and clearing breakpoints and stepping into statements at the DSm level. Limitations of this work are that the mapping construction inevitably increases the complexity of the code generator and that the said mapping is not readily presentable to the modeller.

Several academic and non-academic efforts have investigated 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[2] is described. The meta-model we introduce in Section 3 is inspired by these two formalisms and is in fact a combination and enhancement.

Finally, in [17], 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) as well as propose means to realize the said elevation. Their technique consists in annotating high-level models which are later transformed to performance models that lend themselves to analysis. The reasoning behind integrating low-level non-functional requirement related concepts so early on is that it is best not to realize that these requirements can not be met under the current design once implementation is well under way. The mindset of SPLs (where parameterizing high-level domain-specific concepts enables application synthesis) is of course very similar to that of DSM. Thus, means to reason about performance related concerns such as resource utilization and application response time at the DSm level are desirable.

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

3. META-MODELS & 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 1a). Timed, conditional and user-prompted transitions describe the flow of control between `Containers` – that can contain other `Containers` and `Widgets` – and `Actions` – 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 1b gives an overview of the hierarchical relationships between the meta-models in play. The following subsections overview each transformation[3].

3.2 PhoneApps-to-Statecharts

The first step in the synthesis of executable applications from PhoneApps models is the *PhoneApps-to-Statecharts* transformation which extracts the models' behavioural components. Rather than attempt to invent a novel way of transforming and generating code for complex behaviour, we use the extensively proven and studied formalism of Statecharts [10] as our target formalism. The behavioural semantics of PhoneApps models are fully encompassed in the edges between `Containers` and `Actions` 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 2 shows an example graph transformation rule[4] from a subgraph of a PhoneApps model to its equivalent in the Statechart formalism. When the full PhoneApps-to-Statechart transformation has run its course, every `Container` and `Action` has a corresponding *state*. These are connected via customized *transitions* according to the edges that connect their respective `Containers` and `Actions`. Note that the current range of possible behaviours of PhoneApps models requires only the expressiveness of timed automata. Future work will extend the formalism with notions of hierarchy and concurrency such that more powerful Statechart features (e.g., orthogonality, nesting) become required.

3.3 PhoneApps-to-AndroidAppScreens

After isolating and transforming the behavioural components of PhoneApps models to Statecharts, another trans-

[3]See [13] for more detailed descriptions of the steps that make up these transformations.

[4]Rules are the basic building blocks of rule-based graph transformations. They are 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.

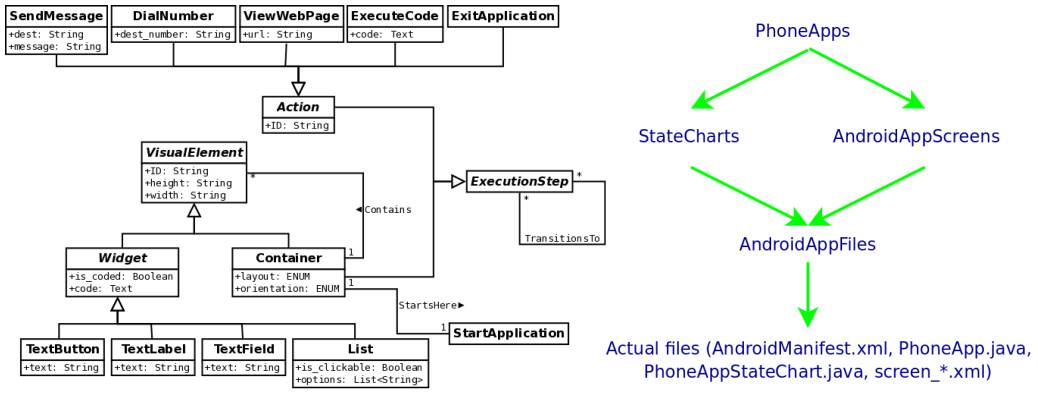


Figure 1: (a) The PhoneApps meta-model (as a Class Diagram); (b) A Formalism Transformation Graph [9] for PhoneApps.

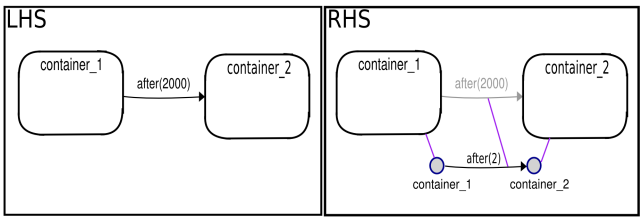


Figure 2: A PhoneApps timeout mapped to a Statechart timeout. The grayed out transition between `Containers` illustrates the marking of that transition as "*visited*".

formation 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 3a). When the full *PhoneApps-to-AndroidAppScreens* transformation has run its course, top-level `Containers` and `Actions` each have corresponding `Screens` and `Acts` respectively. These are appropriately connected to some number of constructs that represent snippets of generated Google Android-specific code (e.g., XML layout code, application requirement manifests, event listener code). Figure 3b shows an example translation rule from a subgraph of a PhoneApps model to its equivalent in the AndroidAppScreens formalism.

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

First, 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 [18]. 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 transformation rules via *generic edges*[5]. These have minimal impact on the readability of the 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 [6].

Second, the aforementioned generic edges can aid in the debugging of the graph transformations themselves. Advanced DSM tools such as AToM^3 [9] which support step-by-step execution of rule-based transformations provide a limited but free transformation debugging environment where one can very easily observe (and modify) the effect of every single rule in isolation. Complex tasks such as determining what was generated from which model entity become trivial and don't require any further instrumentation. Once again, this is considerably easier, more modular and more elegant that lacing a coded generator with output statements and breakpoints.

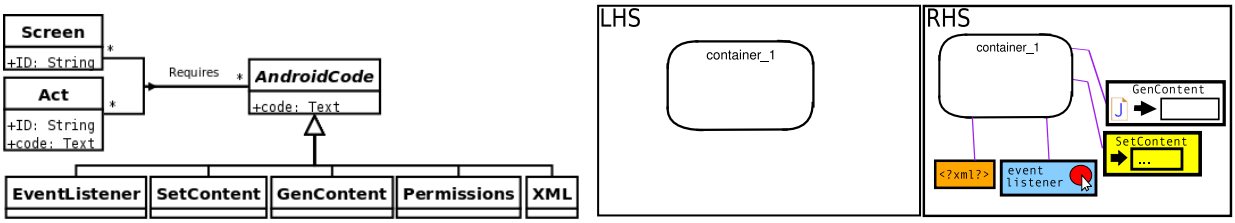
Third, although in a finished product the inner workings that convert DSms to artifacts should be hidden from the modeller, it may be useful for educational purposes to see how higher- and lower-level constructs are related (as demonstrated in Figure 5). Both our transformation rules and the cross-formalism links they produce explicit these relationships.

Fourth, the multiple intermediate layers between model and artifact (and the links between them) provide a means to observe models from various "viewpoints". For instance, in the context of PhoneApps, to study only the behavioural aspects of a model, one could observe the generated Statechart in isolation from the DSm and the other generated artifacts.

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 rela-

[5]Notice the purple, undirected edges between constructs of different formalisms in the figures describing rules.

[6]Future work will explore extending these animation capabilities to more advanced debugging activities such as altering the execution flow.



(a) (b)

Figure 3: (a) The AndroidAppScreens meta-model; (b) Extracting information from a `Container` into new AndroidAppScreens constructs.

tively simple (graphical) model transformation 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- and Statecharts-to-AbstractFiles

Benefits of keeping links between models and generated artifacts were discussed in the previous subsections. Following the same mindset, we introduce the *AbstractFiles* formalism. This trivially simple formalism 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 4a and 4b show two example transformation rules from subgraphs of an AndroidAppScreens model to their equivalent in the AbstractFiles formalism. 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.

4. CASE STUDY: CONFERENCE REGISTRATION IN PHONEAPPS

We now present a hands-on example that demonstrates the successive 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 and code synthesis engines that result from our approach. Figure 5a shows the modelled conference registration application CR in the PhoneApps formalism. There are 3 main use cases: (1) registering, (2) viewing the program schedule and (3) canceling a registration. The first 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" on the *ActionChoice* screen and is taken to the *EnterName* screen;
3. The user enters his 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.

The output of the PhoneApps-to-Statecharts transformation is shown in Figure 5b[7]. Essentially, the application behaviour encoded in CR's transitions is isolated and used to produce an equivalent Statechart. Not visible are the *state entry actions* which effect function calls to generated methods that carry out tasks on the mobile device such as loading screens and sending text messages.

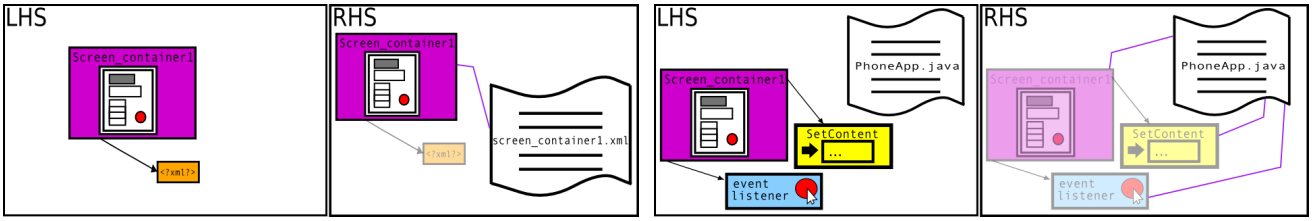
The output of the PhoneApps-to-AndroidAppsScreens is shown in Figure 5c. Essentially, the layout and mobile device specific aspects encoded in CR are translated to appropriate elements of the AndroidAppsScreens formalism.

Figure 5d shows the model after the AndroidAppsScreens-to- and Statecharts-to-AbstractFiles transformations have completed[8]. The two transformations output to a disjoint set of AndroidAppsFiles entities 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 final step is the trivial transformation of the `ModelledFile`s to actual files on disk. The end result of this series of transformations is two-fold. First and foremost, a fully functional Google Android application that perfectly reflects the original PhoneApps model is synthesized as shown

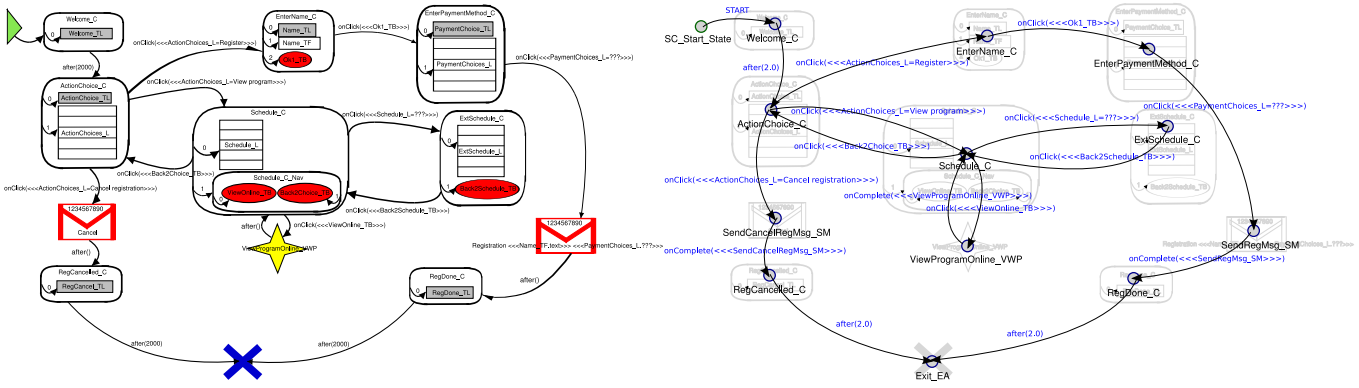
[7]For clarity, we refrain from reproducing the entire CR model and generic edges between it and generated constructs in Figures 5b, 5c and 5d. Instead, we overlay corresponding constructs.

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

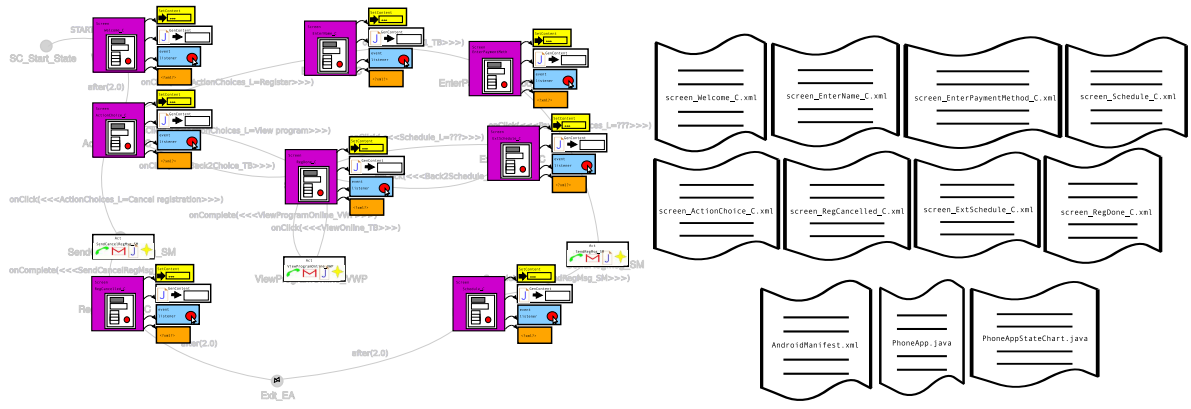


(a) (b)

Figure 4: (a) Creating one `ModelledFile` per `Screen` to hold its XML layout specification; (b) Appending event listener and content initialization code to a `ModelledFile` of the main Java artifact "PhoneApp.java".



(a) (b)



(c) (d)

Figure 5: (a) Conference registration in PhoneApps; (b) The CR model after applying the PhoneApps-to-Statecharts transformation; (c) The CR model after applying the PhoneApps-to-AndroidAppsScreens transformation; (d) The CR model after applying the AndroidAppsScreens-to-AbstractFiles and Statecharts-to-AbstractFiles transformations.

in Figure 6. Second, an intricate web of interconnections between model entities at different levels of abstraction is created. This web can be used for explanatory purposes (i.e., as we have used it to relate corresponding constructs in Figures 5b and 5c) or to ease debugging and simulation of DSms.

5. CASE STUDY: PERFORMANCE METRICS FROM PHONEAPPS

Our approach can also be beneficial in the context of modelling embedded systems and more specifically, in the addressing of their characteristic non-functional requirements. Although the mobile devices we target are indeed embedded systems, the Google Android API abstracts away traditional embedded system concerns. However, it is conceivable that information such as expected running time and battery usage may be required by the modeller. Indeed, the designer of a PhoneApps DSm might be faced with non-functional requirements pertaining to resource utilization and application response time. Hence, means to constrain or at least measure such performance related aspects should be provided. In this case study, we compare how such facilities could be implemented using both our approach to artifact synthesis and the traditional coded generator approach.

First, see Table 1 for a set of imaginary performance specifications for all Google Android devices. Second, let us assume that these specifications are stored and formatted such that they can be easily read from a modelling tool or a coded program. Finally, let us also assume that the modeller of a conference registration application is faced with the three non-functional requirements listed below:

- The full execution must require less than x% battery power;
- The full execution must require less than y seconds;
- The waiting time between two `Screen`s should never exceed z seconds.

Such requirements in conjunction with target platform specifications (ala. Table 1) could help a modeller discriminate between design decisions such as preloading a device with data versus downloading it at runtime, or communicating via email versus text messaging.

The task of generating performance models and/or statistics from DSms 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 (e.g., estimates of battery usage and running time[9]). One option would be to extend an existing generator (in this context, a generator that would synthesize complete Google Android applications from PhoneApps models) with performance measuring provisions. Another option would be to write a new generator from scratch and have it focus solely on extracting performance related information from models. Both approaches have merits and limitations. Although the latter will be more efficient, it may induce considerable code duplication since model traversal and information extraction will conceivably be carried out in a similar fashion than in existing generators. On the other hand, the former option might introduce undesired complexity and reduce the modularity of an already complex generator. In either case, providing more advanced features (e.g., "tagging" domain-specific constructs with battery usage or running time information at the DSm level, live performance data updates from DSm modifications) that exploit one- or two-way links between model and artifacts will require considerably polluting the generator's code.

[9]These estimates could be parameterized and plotted to illustrate pertinent bounds such as "the application respects requirement r provided SMS messages are restricted to x characters" or "local data as opposed to web data should be used if the said data is larger than y megabytes".

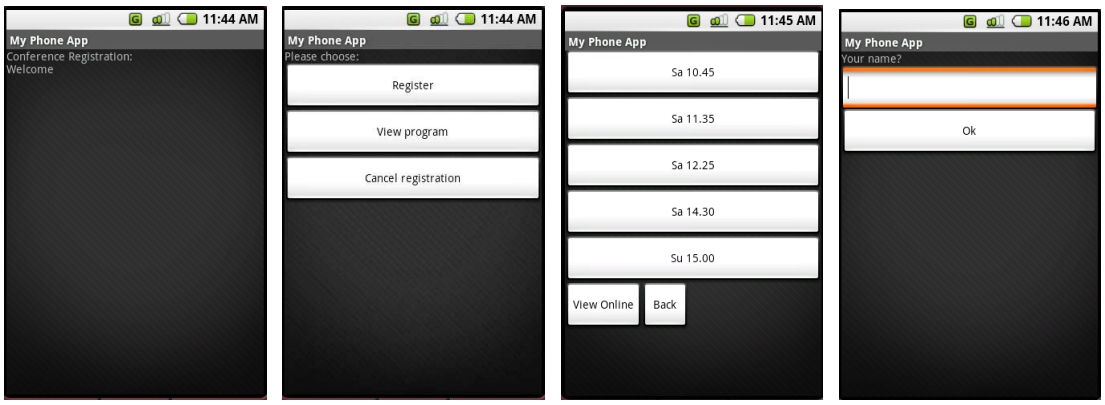
Analogous options for generating performance information using our model transformation-based our approach are fairly obvious: a new orthogonal model transformation could be created or existing model transformations could be refactored. For instance, *PhoneApps-to-* or *AndroidAppScreens-to-Metrics* transformations could be introduced with rules that count such things as the total number of `Widget`s on all `Screen`s along each possible execution path. Both options raise the same concerns as in the traditional approach; namely that PhoneApps-to- and AndroidAppScreens-to-Metrics will conceivably bare numerous similarities to the PhoneApps-to- and AndroidAppScreens-to-AbstractFiles transformations respectively whereas merging everything into a single transformation might result in a complex intermingling of concerns. Nevertheless, our model transformation-based approach will facilitate the creation of links between DSm and synthesized performance related artifacts thereby facilitating the implementation of the aforementioned advanced features.

Thus far, we have focused on "static" performance metrics in the sense that we assume that the desired information can be extracted from static DSms. However, it may be required to execute (or simulate) models to obtain more detailed and precise results (e.g., average and expected measurements as opposed to best and worst case theoretical bounds). Constructing such "dynamic" performance metrics would require that the generator or transformation rules instrument the synthesized executable artifacts (e.g., with code to increment a counter for every displayed widget). Instrumentation to output global information such as total measured running time could just as easily be produced by the traditional approach than by our own. However, to output more localized measurements (ideally on the DSm itself and possibly even during runtime) like the battery usage of a `SendMessage` entity or the time of entry of each `Screen`, links between DSm and artifact become a necessity. As we have repeatedly argued, such links are easier to specify and represent with our approach than with coded generators.

In sum, the numerous benefits of our technique to artifact synthesis not only apply to generating coded applications but also to the production and display of performance data useful in the modelling of embedded system applications.

6. CONCLUSION AND FUTURE WORK

We proposed a structured approach to artifact generation where layered model transformations are used to modularly isolate, compile and re-combine various aspects of DSms, while leaving behind a web of interconnections between cor-



(a) (b) (c) (d)

Figure 6: Screengrabs of the synthesized application running on a Google Android device emulator: (a) The *Welcome* screen; (b) The *ActionChoice* screen; (c) The *ProgramSchedule* screen; (d) The *EnterName* screen.

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 1: Imaginary performance specifications for Google Android devices.

responding constructs from formalisms at different levels of abstraction. We argued that our approach improves upon the traditional ad-hoc coded generator approach to synthesizing applications from DSms. Discussed benefits include improving the domain-specificity and easing the development and debugging of code synthesis engines, providing clear pictures of the real and conceptual links between constructs at different levels of abstraction and simplifying the construction of inter-formalism mappings that enable advanced functionalities such as DSm animation, simulation, debugging and on-the-fly tagging.

The DSLs, transformations and case studies we presented provided empirical evidence to back our claims that (graphical) model transformations are a better means of generating artifacts (be they programs or performance metrics) from DSms than coded generators. However, our approach still requires some formalization. Since we essentially *ripped* and *woven* 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 language design. For instance, combining some form of explicit DSL concept generalization relationship (e.g., `PhoneApps.ExecutionStep` *is a* `Statechart.State`) with higher-order transformations[10] 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 of our approach, although (semi-)automatically generating transformation rules seems straight-forward, generating parts of a coded generator would not only require considerable effort but likely produce a complex and incomplete program that would be difficult to understand let alone complete and maintain. Thus, our technique is more amenable to (semi-)automation.

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

7. REFERENCES

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

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

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

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

[5] 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.

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

[7] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. 832 pages.

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

[11] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling : Enabling Full Code Generation*. Wiley-Interscience, 2008. 427 pages.

[12] 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.

[13] Raphael Mannadiar and Hans Vangheluwe. Modular synthesis of mobile device applications from domain-specific models. Technical report, McGill University, 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] Yu Sun, Jules Whit, and Jeff Gray. Model transformation by demonstration. In *MODELS*, volume LNCS 5795, pages 712–726, 2009.

[17] 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)*, 2008.

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