Supporting Maintenance Tasks on Transformational Code Generation Environments

Victor Guana
Department of Computing Science
University of Alberta
Edmonton AB, T6G 2E8, Canada
http://webdocs.cs.ualberta.ca/~guana
guana@cs.ualberta.ca

***Abstract*—At the core of model-driven software development, model-transformation compositions enable automatic generation of executable artifacts from models. Although the advantages of transformational software development have been explored by numerous academics and industry practitioners, adoption of the paradigm continues to be slow, and limited to specific domains. The main challenge to adoption is the fact that maintenance tasks, such as analysis and management of model-transformation compositions and reflecting code changes to model transformations, are still largely unsupported by tools. My dissertation aims at enhancing the field's understanding around the maintenance issues in transformational software development, and at supporting the tasks involved in the synchronization of evolving system features with their generation environments. This paper discusses the three main aspects of the envisioned thesis: (a) complexity analysis of model-transformation compositions, (b) system feature localization and tracking in model-transformation compositions, and (c) refactoring of transformation compositions to improve their qualities.**

***Index Terms*—software maintenance, transformation composition, transformation complexity, transformation refactoring.**

I. Introduction

Increased software flexibility, portability and reduction of coding errors have driven the slow, yet increasing adoption of transformational software development [1]. A substantial body of research has already been accumulated on theoretical and practical aspects of (a) domain-specific modeling languages (DSMLs), and (b) transformation engines [2][3]. DSMLs promote the use of models to capture software problem and solution spaces; transformation engines aim to automatically translate problem models into solution artifacts, such as code and deployment descriptors.

The transformational software-development process starts with a set of domain models, representing the problem, and concludes with the solution, in the form of executable code. The process itself consists of a composition of model-transformation steps; each such step parses the information contained in its input models and translates it into a set of output models, which, in turn, are translated further by new steps, until final executable artifacts are obtained [2]. The main advantage of this type of transformational software generation lies in the intermediate models produced, which explicitly represent different system functional, quality, technical, and implementation concerns [4]. Intermediate models can be used for the specification of the software architecture, the data-distribution strategy, or platform-specific implementation decisions. Using the same problem specification, different intermediate models imply the automatic generation of software systems with different architectural styles, data organizations, security requirements, and/or execution platforms.

Maintenance issues, such as (a) analysis of model-transformation chains and (b) synchronization of evolving software features and model-transformation compositions, challenge the popularity of this software-construction paradigm. The objective of this thesis is to analyze and develop support for the maintenance of model-transformation compositions, in the face of evolving requirements for the to-be-generated software.

II. Maintaining Model-Transformation Compositions

The maintenance of software-transformation environments presents two main challenges of our interest: (a) understanding the nature of the transformation complexity and (b) supporting the localization of software features in transformation compositions.

A. Model-Transformation Composition Complexity

As models and model transformations become first-class entities of the software-construction process, we need to consider how software qualities, such as modularity and cohesion, can also be applied to model transformations to indicate how "good" the transformational environment is overall [5]. Nowadays, only empirical evaluations are performed to assess the maintainability of transformational strategies. There is not yet any systematic support for how model-transformation compositions should be designed for maintenance purposes, and the design of modeling languages and transformations is still an art.

There are two major categories of transformational approaches: model-to-model, and model-to-text. A model-to-model transformation contains mapping rules that specify how an input set of models are mapped into a set of output models (i.e. source and target models). In rule-based transformation languages such as ATL[6], RubyTL[7], or ETL[8], each mapping is specified with a transformation rule. On the other

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
Doctoral Symposium

hand, model-to-text transformations are usually implemented as modules of metacode templates and expansion rules in languages such as EGL[8] and Acceleo [9]. Expansion rules select and print string patterns that map source-model elements to target code, or simply text syntaxes such as XML.

Multiple mappings may be implemented in a single transformation rule set or module. Languages like ATL and ETL provide different execution semantics for rule inheritance and context-dependent execution behaviors. As in object oriented programming, rich language semantics make the design and maintenance of transformational generations a complex process, involving a substantial cognitive challenge for developers.

Transformation modules can be composed in order to accomplish complex tasks, with an internal or an external composition [10]. External composition allows the integration of transformation scripts developed using (potentially) multiple transformation languages. They are usually specified in a pipeline architecture, where the output of a transformation serves as the input for the next one, resulting in a model-transformation chain of model-to-model and model-to-text transformations[2]. Internal composition is characterized by the "compilation" of multiple transformation rules into a single transformation unit. It is often implemented using a single transformation language that can be executed as a whole [5][10]. From the developer's perspective, using an external composition implies that information about models and rule-execution semantics is only accessible inside each shackle of the transformation chain. With internal composition, transformation semantics allow scope access to the entire transformational process from any point of its execution.

Automating the complexity analysis of model-transformation compositions will ease the identification of transformation chains bad smells. Particularly, we pursue the detection of transformation composition anti-patterns that might cause the backward integration of code refinements and feature modifications expensive in maintenance terms.

B. Feature Localization and Tracking

Once code is generated, processes such as user and acceptance testing may necessitate that the original problem code solution be modified. Modifications may include code refactoring for design improvement, performance tuning for mission-critical systems, energy-consumption optimization for mobile applications, and bug fixes, among others. In order to synchronize the transformational environment (including models and transformation steps) with the changed requirements and code, refinements have to be backwardly integrated throughout the code-generation environment.

In a backward integration scenario, modifications such as the creation or deletion of elements in the DSMLs to capture uncovered or residual information, or the logic adaptation in transformation modules to map modified or new model elements, have to be performed in order to comply with new code refinements. This process is iterative and part of the natural evolution of the transformational software-construction paradigm [11]. Given the different architectures in which transformations can be composed, and the numerous models that pivot the code-generation process, tracking the origins of a system's feature through model-to-model and model-to-text transformations can potentially be a very challenging task. Usually, the semantic and syntactic origins of a generated feature are scattered among different pivot models and transformation modules. Therefore, its manual identification is either infeasible, or at the very least, error prone for large and complex generation processes.

Automatic feature localization and tracking in transformation-based generations will reduce maintenance efforts for tasks that involve code refinements, and feature evolution. It will support developers on the identification of model, and model transformation compositions hotspots that have to be modified in order to integrate new requirements to an existing generation environment.

III. Background and Objectives

The overall objective of this research agenda is to develop tools for mitigating the challenges around the maintenance of transformational code-generation environments, focusing on model and model-transformation analysis and maintenance. To that end, we focus our research around three objectives: feature localization and traceability through models and transformations, cognitively valid complexity assessment of models and model transformations, and refactoring of rule-based model transformations.

A. Feature Localization and Traceability

Code changes that have to be backwards synchronized with the model-transformation composition that led to its generation are typically motivated by features that are required to be changed (e.g. bug fixes, performance improvements, functionality extensions), or to be deleted (e.g. in case of unused functionalities), or to be created (e.g. new hardware adaptations, or user requirements) [12]. There are two important challenges that we want to address in this process. First, we plan to support the identification of the features implemented by the manually modified code segments. Second, we intend to trace these features throughout the models, and model-transformations, involved in the transformation composition that generated the original code in the first place, in order to highlight hotspots where transformation rules, models, or even DSMLs have to evolved to support the manual modifications.

In [13], a survey of model-driven engineering traceability analyzes how current traceability techniques represent the unit of information to be traced, and how the traceability processes are executed (e.g. using auxiliary models, extending models with trace information, or using annotations). Existing techniques trace individual model concepts looking for one-to-one mappings in transformation rules. Usually they are not, or only partially, supported by automatic tools. Furthermore, they do not deal with traceability software features in transformation chains that involve both model-to-model, and to model-to-text transformations. In our approach, we intend to

make information about the interdependencies of features and transformation paths between models visible to maintainers. This information will enable developers to isolate the to-be-changed transformation elements and ease their decision making on how to modify the impacted assets. We are currently exploring two mechanisms to tackle feature localization and traceability challenges in transformation compositions: formal concept analysis classifications based on lattice theory [14], and static analysis combined with symbolic execution [15] of generated code sources.

B. Cognitively Valid Complexity Analysis of Model-Transformation Compositions

Since maintenance requests originate from the need for code refinements and changes to the software architectures, we believe that the complexity analysis of transformation compositions should focus on (a) how difficult it is to change the derivation of a system feature, (b) how scattered the transformation rules, relating to a specific feature, are across the transformational-generation environment, and (c) how rules can be reused and composed for the derivation of similar feature structures.

Fenton and Pfleeger [16] understand four types of software complexity: problem, algorithmic, structural and cognitive complexity. To the best of our knowledge, all existing transformation-complexity measurement proposals focus on understanding the structural complexity of single transformation modules. van Amstel et al. presented perhaps the most important contribution on this matter [17]. Authors quantify the structural complexity of individual transformation units with numerous metrics that involve the syntactical characteristics of a transformation module.

Opportunities remain open for compositional-complexity measures that estimate how difficult it is to maintain a generated feature across a model-transformation execution. Properties, such as how many rules are involved in the generation of a system feature, or how these rules are scattered across multiple modules of the transformation composition, may indicate "bad smells" that impact how developers understand and navigate through generation hotspots. Such smells will point to the creation, deletion, or modification of transformational structures that collaborate for the derivation of a single software feature. We envision the development of a suite of *compositional-complexity metrics* to evaluate how transformation compositions manipulate the information along a model transformation process. Although they include structural elements of transformation modules to understand their complexity, they will be mainly driven by the cognitive challenges that maintainers face while synchronizing evolving features with their generation environments.

C. Refactoring of Model-Transformation Compositions

Based on the analysis of transformation-composition complexity and the feature localization and tracking capabilities, I will pursue the creation of a tool to automate the identification of opportunities for, and the implementation of, refactorings on transformation compositions. These refactorings are intended to ease the maintenance tasks that involve backward synchronization of software features with their generation environments.

Gniesser in [18] presents a refactoring conceptual framework for ATL-based transformation rules. The framework involves the definition of syntactical bad smells such as duplicated transformation code, oversized rules, unused language features, among others. Wimmer et al. [19] propose a broader catalog of refactorings for model-to-model transformations that also includes refactorings for the optimization of OCL expressions [20]. Both proposals use ATL refinements in order to test semi-automatic bad smell detection and application of refactorings.

Current refactoring alternatives pursue the simplification of the structural complexity in individual transformation modules. We want to support the automatic detection of bad smells, and refactoring opportunities of model transformation compositions. Based on compositional-complexity measures, and feature traceability detection simplification, model-to-model and model-to-text composed transformations can be refactored. Refactoring suggestions could include modularizing transformation rules according the type of features they help to generate or, depending on how their execution context relates with model elements and feature traces, change their design structure from an external to an internal composition.

IV. Progress Thus Far

We are currently developing PhyDSL, a physics-based domain-specific language and a transformational generative environment for physics-based games. Currently, we are using PhyDSL and its code-generation environment to construct customizable, and cost-effective tablet-games for the rehabilitation of visuomotor conditions suffered after brain injuries. Using PhyDSL we can generate a variety of games, including, but not limited to, *avoid object collisions*, *solve the maze*, and *capture the flag*. The PhyDSL generation process is divided in three independent model-transformation chains that start with the same game specific language: *Physics Rules* (i.e. how the game world is affected by forces like gravity, or friction), *Game Navigation* (i.e. how different screens represent game information), and *Game Logic* (i.e. how scoring rules, and the interaction between game actors are defined).

As a concrete example of the challenges involved in the maintenance of model-generated code, let us consider our recent experience with the implementation of two new features: first, to support the submission of game scores to a remote server, and second, to optimize the rendering of visual objects to reduce the game-energy consumption. These features required the implementation of a whole new set of classes that collaborate to support their execution. Additionally, we had to implement a set of refinements to extend the existing elements and weave the new feature behaviors. For example, game screens should now provide the score submission buttons, and login fields; the scoring manager should keep track of an authentication certificate, and the game canvas should use

additional rotation a position solvers for the rendering of game objects. As expected, these refinements cut across existing features generated by the three generation levels.

In order to reuse these feature in future game generations, we backwardly integrated the new code and respective refinements to the model-based generation process (i.e. model compositions conformed by model-to-model and model-to-text transformations). We, first, listed all the features that were impacted by the refinements. Next, we traced the models and transformation rules that collaborate for their generation. Third, we identified the transformation modules that required modifications. Next, we identified candidate models that could capture the newly required information (e.g. add or remove model concepts and attributes). Finally, we integrated new transformation logic to include new model structures and embedded design decisions. We found this process quite difficult, in spite of having been the original developers of PhyDSL and its environment. The transformations and models required for the generation of a single software feature were scattered among different transformation modules and generation chains. Moreover, since the existing features were not isolated, transformation rules and models became redundant and less cohesive. For example, we extended transformations in both *Game Navigation* and *Game Logics* generation chains to include the remote scoring generation artifacts. In both levels, we duplicated model elements and transformation logic. This duplication was caused by the fact that the composition design did not provide access to required model concepts and transformation rules. Among many others, these maintenance issues revealed that if we modularize the generation process to leverage the trace and extension of software features, our feature addition and modification would be cognitively easier to understand, and therefore, faster and sustainable for future feature extensions.

V. Evaluation Plan

My evaluation strategy is divided in three phases, that involve controlled empirical studies with developers using the tools I plan to develop. First, I will study the correlation between (existing and newly designed) metrics of model-transformation compositions and the cognitive difficulty faced by developers, in a variety of maintenance tasks of model-generated code. Second, I will evaluate the precision and recall of my semantic and syntactic feature-tracking methods on a given model-transformation composition, including both model-to-model and model-to-text transformation modules. This model-transformation environment will most likely be PhyDSL. At the same time, I plan to evaluate how exactly my feature recovery-location tools reduces the maintenance effort required by the developers. Finally, I will conduct empirical studies with developers to compare the maintenance costs of complex transformation compositions and their refactored counterparts, when synchronizing feature and code refinements, in order to evaluate the effectiveness of my model-transformation refactoring toolkit.

VI. Conclusions

In my dissertation I plan to analyze the complexity of model-transformation compositions and to develop methods and tools for measuring this complexity, supporting the location of code features throughout the model-transformation composition that produced it, and refactoring model-transformation compositions to improve their qualities. My work will be driven by systematic empirical studies to support and analyse the benefits of the proposed tools and methodologies, towards increasing the field's capacity for maintaining code generated through model transformations.

VII. Acknowledgements

I would like to thank my PhD supervisor, Prof. Eleni Stroulia for her helpful comments and suggestions.

References

[1] S. Mellor, T. Clark, and T. Futagami, "Model-driven development: guest editors' introduction." *IEEE software*, vol. 20, no. 5, pp. 14–18, 2003.

[2] K. Czarnecki, "Generative programming: Methods, techniques, and applications tutorial abstract," *Software Reuse: Methods, Techniques, and Tools*, pp. 477–503, 2002.

[3] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.

[4] B. Vanhooff, D. Ayed, and Y. Berbers, "A framework for transformation chain development processes," in *Proceedings of the ECMDA Composition of Model Transformations Workshop*, 2006, pp. 3–8.

[5] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault, "Towards a general composition semantics for rule-based model transformation," *Model Driven Engineering Languages and Systems*, pp. 623–637, 2011.

[6] F. Jouault and I. Kurtev, "Transforming models with atl," in *Satellite Events at the MoDELS 2005 Conference*. Springer, pp. 128–138.

[7] J. Cuadrado, J. Molina, and M. Tortosa, "Rubytl: A practical, extensible transformation language," in *Model Driven Architecture–Foundations and Applications*. Springer, 2006, pp. 158–172.

[8] D. Kolovos, R. Paige, and F. Polack, "The epsilon transformation language," *Theory and Practice of Model Transformations*, 2008.

[9] J. Musset, É. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire, "Acceleo user guide," 2006.

[10] A. Kleppe, "First european workshop on composition of model transformations - cmt 2006," *Technical Report TR-CTIT-06-34*, 2006.

[11] D. Hearnden, M. Lawley, and K. Raymond, "Incremental model transformation for the evolution of model-driven systems," *Model Driven Engineering Languages and Systems*, pp. 321–335, 2006.

[12] K. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*. ACM, 2000, pp. 73–87.

[13] I. Galvao and A. Goknil, "Survey of traceability approaches in model-driven engineering," in *11th Enterprise Distributed Object Computing Conference, 2007. EDOC 2007*. IEEE, 2007, pp. 313–313.

[14] B. Ganter, R. Wille, and R. Wille, *Formal concept analysis*. Springer Berlin, 1999.

[15] J. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[16] N. Fenton and S. Pfleeger, *Software metrics: a rigorous and practical approach*. PWS Publishing Co., 1998.

[17] M. van Amstel and M. van den Brand, "Quality assessment of atl model transformations using metrics," in *Proceedings of the 2nd International Workshop on Model Transformation with ATL), Malaga, Spain*, 2010.

[18] P. Gniesser, *Refactoring Support for ATL-based Model Transformations*. MSc Thesis, Faculty of Informatics - Vienna University of Technology, 2012.

[19] M. Wimmer, S. Martínez, F. Jouault, J. Cabot *et al.*, "A catalogue of refactorings for model-to-model transformations," *The Journal of Object Technology*, vol. 11, no. 2, pp. 21–40, 2012.

[20] J. Warmer and A. Kleppe, *The object constraint language: getting your models ready for MDA*. Addison-Wesley Professional, 2003.