Bottom-Up Model-Driven Development

Hamid Bagheri
University of Virginia,
151 Engineer's Way,
Charlottesville, VA 22903 USA
hb2j@virginia.edu

Kevin Sullivan
University of Virginia,
151 Engineer's Way,
Charlottesville, VA 22903 USA
sullivan@virginia.edu

***Abstract*—Prominent researchers and leading practitioners are questioning the long-term viability of model-driven development (MDD). Finkelstein recently ranked MDD as a *bottom-ten* research area, arguing that an approach based entirely on development and refinement of abstract representations is untenable. His view is that working with concrete artifacts is necessary for learning what to build and how to build it. What if this view is correct? Could MDD be rescued from such a critique? We suggest the answer is yes, but that it requires an inversion of traditional views of transformational MDD. Rather than develop *complete*, abstract system models, in ad-hoc modeling languages, followed by *top-down* synthesis of *hidden* concrete artifacts, we envision that engineers will continue to develop concrete artifacts, but over time will recognize patterns and concerns that can profitably be lifted, from the *bottom-up*, to the level of *partial* models, in *general-purpose* specification languages, from which *visible* concrete artifacts are generated, becoming part of the base of both concrete and abstract artifacts for subsequent rounds of development. This paper reports on recent work that suggests this approach is viable, and explores ramifications of such a rethinking of MDD. Early validation flows from experience applying these ideas to a healthcare-related experimental system in our lab.**

***Index Terms*—Model-driven development, Bottom-up, Partial synthesis.**

I. Introduction

The idealized transformational view of model-driven development (MDD) combines (1) technologies for top-down synthesis of hidden concrete implementations from complete, abstract systems models in domain-specific or full-spectrum languages, with (2) a *rationalist stance,* holding that people should develop abstract models only, with concrete artifacts hidden from view. Prominent researchers have joined notable practitioners in questioning the viability of this style of MDD. Among practitioners, Ambler says, "I'm concerned about the viability of the [Model Driven Architecture] MDA.... [A]lthough the MDA is a very wonderful idea I suspect that it will succeed in only a very small percentage of organizations [15]." He argues that current modeling languages do not support the real-world needs of most projects (e.g., the user interface and database components needed in many systems); developers lack adequate modeling skills; and tooling is inadequate. Fowler says, "Although I've been involved, to some extent, in ... [model driven development] for most of my career, I'm rather skeptical of its future. Most fans ... base their enthusiasm on the basis that models are ipso facto a higher level abstraction than programming languages. I don't agree with that argument - sometimes graphical notations can be a better abstraction, but not always... [13]." McNeily says that unless executability and translatability can be brought to the kinds of models that business, as opposed to real time embedded systems, developers use, that modeling will remain subject to criticism as "tool-centric busy work of dubious value, and that we should go back to using a whiteboard [14]."

Among researchers, Finkelstein is a notable critic. In a recent blog post on the *Bottom 10 Challenges in Software Engineering Research* [2], he said, "...the idea that changes could be made in a high-level specification and then somehow 'replayed' is appealing but ignores the ways in which ... learning arises, in the context of specific representations and through verification or testing tied to that representation." Finkelstein's concern is consistent with that of Ambler (people are generally poor at working only at an abstract level), Fowler (who questions whether available abstract modeling formalisms are useful in general practice), and McNeily (who makes much the same point).

In this paper, we address the question, what if Finkelstein is right? That is, we will assume his premise and derive a set of consequences. At the heart of this paper we offer the idea that Finkelstein's position does not require that we abandon the technologies of synthesis from abstract models, but that it does demand that we replace the rationalistic, top-down view of transformational MDD with an *empirical* approach rooted in the proposition that essential learning happens when people deal with the concrete artifacts of ordinary code-, design-, and verification-based software development.[1] We thus discard the rational stance and its associated complete, abstract system models, in ad-hoc modeling languages, followed by *top-down* synthesis of *hidden* concrete artifacts, in favor of an empirical stance and the idea that one learns "at the bottom" and, having learned, abstracts bottom up, to the level of *partial* abstract models, in general-purpose specification languages, from which one can then synthesize *visible* parts of the base of concrete artifacts making up a project.

The technical key to the viability of this position is found in our recent work on *partial synthesis* [5], which provides a key enabling technology for a *bottom-up approach to MDD*, by supporting modeling of *selected* aspects of systems, refinement from such models to support *partial synthesis*, and a clean

[1]We note that our use of the term "empirical" differs from that common in software engineering. We do not mean to focus on the validation of research results in industrial settings, but rather on the idea that one must learn from grounded experience, and not just deal in high-level abstractions.

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA
New Ideas and Emerging Results

separation and integration of model-driven and hand-crafted code artifacts. Our conjecture is that we are able to reap the benefits of MDD, including improved abstraction and productivity, in a manner consistent with iterative, bottom-up development practices.

The rest of this paper introduces the working scheme of our approach, reports and discusses our experience on applying the approach in an evolutionary development of a laboratory-scale operational model of nation-wide health information systems, surveys related work, and concludes.

II. Bottom-up Partial Formal Synthesis

The essence of our approach is based on a set of principles different from those typically held in traditional model-driven development (MDD) approaches. In this section, we present our approach based on those principles.

- *Bottom-up.* Our approach is bottom-up. In this view, models follow from code and other concrete artifacts, rather than the other way around. Specifically, we posit that software engineers, having worked diligently in the concrete, empirical world of code, can find it profitable to derive and validate abstract models of *selected aspects* of code, which then support analysis and synthesis. The benefits are not in hiding the code behind abstract models, but rather in leveraging the technologies of MDD for improved abstraction, productivity and reliability going forward.

- *Partial Models.* Software development, in traditional MDD, is centered around model specifications of the system, and everything is then derived from those model specifications. Our second principle, by contrast, states that it is often not practical to develop abstract models for an entire system. Rather, it is often better to extract models for certain stabilized aspects of the system. Indeed, we split the code-base for a system into two parts: (1) a part that is synthesized from partial models, and (2) a part that continues to be developed manually. The artifacts that software engineers develop thus include both code and models from which additional code is synthesized.

A criterion for making a decision to lift some idiomatic aspect of the code to the model level is that the aspect has become sufficiently well understood and stabilized. In some sense, for those concerns we expect that the learning is over. As a concrete example, architectural styles are among such aspects of the system, representing recurrent architectural situations [16]. They are expressive abstractions for software understanding. We thus learn architectural styles of systems and use them in documenting application architecture. Technically speaking, we consider architectural styles as metamodels to which abstracted architectural models conform.

- *Partial Synthesis.* Our abstracted models are partial with respect to the underlying empirical domain. They thus support only selective system analysis and synthesis. We use the term *partial synthesis* to refer to an approach in which part of the code base is synthesized from partial models [5]. In fact, partial synthesis technologies give us the ability to decide which aspects of the system are dealt with formally in terms of model representations—from which code will be synthesized—and which aspects of the system are dealt with in terms of code. Partial synthesis techniques required for this approach thus should provide a basis for separation of generated and non-generated code with support for merging that limits the impacts on hand-written code modifications required when the synthesized parts are regenerated.

Different from traditional MDD, where developers produce domain specific languages for use by non-programmer domain experts, in our approach developers both produce and consume concern-specific modeling notations within the scope of the application under development. Specifically, given that a new requirement can be modeled in terms of the already identified and captured model elements, development starts from the model-level. Otherwise, it starts at the code-level.

III. Evaluation

A. Implementation

In this section, we show that our ideas can be reduced to practice. There are many possible approaches to realizing it. The novel concept in this work encompasses any technology that takes partial models and generates partial code base, and that provides an appropriate level of support to limit the amount of developers' code modifications required when the synthesized parts are regenerated. Here we describe one approach, in which we use the *Pol* framework [5] for partial code synthesis.

Pol is an approach and tool-suite for specification-driven synthesis of object-oriented frameworks for platform-based applications. The choice of object-oriented frameworks as synthesis outputs facilitates evolutionary software development process. Specifically, it helps to limit the impacts of changes in input models on hand-crafted code. We developed Pol in part based on Alloy [11], a declarative specification language based on first-order logic with transitive closure that has been optimized for automated analysis. Synthesis is based on constraint solvers, and driven by formal, partial specifications of target platforms and application architectures, expressed in modeling languages embedded in Alloy, and by code fragments encoding framework usage patterns.

B. Case Study

In this paper, we evaluate our previous experience in developing a healthcare-related experimental system as reported on [5], as a case study in bottom-up MDD. We follow the case study method guidelines proposed by Kitchenham, Pickard and Pfleeger [12].

Hypotheses. The claim we make in this paper is that the proposed approach has meaningful potential to provide the key benefits of MDD technologies, including improved abstraction and productivity, while it is still consistent with iterative, bottom-up development practices.

Subject system. The project that we selected as a basis for the evaluation is called *CyberHealth*, a laboratory-scale, operational model of a national-scale health information exchange among diverse institutions that produce and use such information. The system models numerous institutions, including hospitals, health record banks, public health agencies, and the flow of information among these systems.

Planning and Execution. The basic procedure that we used for this study was to iteratively evolve this system through several stages following the approach as we sought to satisfy increasingly demanding requirement. We developed the initial version of our architectural model for the system by reverse engineering of the hand-crafted version, focusing on abstraction of the major entities and interconnections.

To test the improved abstraction hypothesis we measured, throughout the project development lifecycle, the size of specifications captured to realize partial model-driven code synthesis. To test the improved productivity hypothesis we conducted several evolutionary experiments, and through each experiment we measured the portion of code modifications automatically generated.

Results and Interpretation. In this section we report and interpret data we gathered through executing our case study.

1) Abstractive Support. The models we captured include: application architecture, platforms models, and mapping specifications. We also captured recurrent code patterns as design fragments [9].

We recovered architecture specifications of the original version of the CyberHealth system—implemented in over 12 thousands lines of code—and formally modeled it in the Alloy language such that it can be used in our synthesis technique. The architecture specification contains more than 300 lines of Alloy code and involved more than 50 Alloy signatures, several of which extend other signatures. We modeled the application architecture based on published architectural style specifications. The architectural style specifications accounted for about 400 lines of Alloy.

We then developed platform models for some of the platforms being used in this system that we want to synthesize code for them. A platform model is a partial formal model oriented towards the ways in which the platform is to be used in a given application [5]. These captured specifications are partial and do not attempt to capture the complete structure or semantics of a platform or application, but are contrived to enable a desired level of partial code synthesis. During the development of the CyberHealth project, we formally specified four widely-used industrial platforms, namely Restlet, OAuth, CometD and HornetQ platforms, in about 90, 80, 50, and 80 lines of Alloy code, respectively.

We captured architecture-to-platform mappings to produce platform-based implementation model with respect to the architectural models. Mapping predicates explain how architectural style elements map to underlying platform constructs. Details of these studies including the complete versions of Alloy models are available for download [1].

The other significant aspect of the code that we want to generate from more concise model specifications are recurrent code patterns. Platform usage patterns are instances of recurrent code. Using Pol, we document them as *design fragments* [9]. In particular, a design fragment is a specification of how other applications can use platform resources to achieve a specific goal [9]. It defines a pattern of platform use in the form of a parameterized code template. Overall, we documented 51 design fragments for 4 platforms involved in this project.

2) Generative Support. To assess the extent to which this approach can improve the development productivity, we conducted several evolutionary experiments. Here, we briefly discuss the data we obtained through two such experiments.

Experiment 1. In this experiment we wanted to evolve the CyberHealth system developed initially using the pure style of REST [10], such that to use server push technology for eager updating of client views. There were several situations in which we wanted the client views to get updated dynamically. This means that we needed implementation of this technology in different components of the system.

The first step was to decide from where (model-level or code) we would better to start developing the new requirement. As we did not have the elements necessary for modeling the concepts involved in the development of the requirement under consideration, we had better to start from the code-level.

After developing this requirement for only one component based on the CometD platform as a widely used implementation of the server-push technology, we studied the code base and abstracted models to facilitate use of the synthesis technique for implementing this requirement in other components.

Now we can leverage the partial synthesis technique to generate substantial part of the required code for this scenario. The new synthesized code framework providing support for the server-push mechanism encompasses 450 new lines of generated code. To merge the code-base with the new framework, we then modified about 100 lines of hand written code to push specific messages to CometD service channels.

Experiment 2. This experiment is about an evolutionary requirement change where part of the system's data flow protocol should be modified. The system needed to be changed such that data channels are dynamically created only by user request through the principal control system (PCS) component. In that way, components never publish/receive to/from a given channel except when a user establishes the appropriate connections through the PCS.

Implementing this requirement could be started from the model-level, as the elements necessary for modeling the concepts involved in its development were available. In fact, a high-level architectural change was required. After we modified the system architectural model, resynthesis of the architectural code framework provides support for the protected data flow protocol, which accounts for about 700 lines of code scattered in several framework classes. The developer then modified around 570 lines of code to develop details of the application logic.

It is also worth noting that using analyzable specification languages, we can immediately check the correctness of even partial specifications. Precisely, during the modeling process, we iteratively checked the conformance of the partial system architectural model to its architectural style [4]. It also helps us to explore whether there exist implementation models that satisfy constraints implied by target platforms and simultaneously conforms to the structures specified in the architectural model, albeit within the limits of the finite scope.

Overall, our experiments have provided evidence in support of the hypotheses that a pragmatic bottom-up approach based on a partial model-driven synthesis technique improves intellectual control through abstraction and productivity through automation under certain circumstances. We envisage that in an ultimate implementation, one uses several specification languages and corresponding synthesis technologies handling different aspects of the system. This calls for a modularization mechanism so that different models can be changed independently within certain constraints without breaking the whole system, which is an interesting research avenue for future work.

IV. Related Work

This section puts our work in context with related efforts.

Inferring Partial Specifications. A large body of research focuses on inferring partial specifications from code, albeit more for property checking than for synthesis. Among others, *Daikon* [8] discovers likely program invariants by detecting patterns and relationships among values taken by variables during program executions. *SLAM* [6] automatically and incrementally abstracts a given program based on a set of user-provided predicates. The abstraction is captured in terms of a boolean program, which exhibits an identical control-flow structure to the program, but contains only boolean variables, each of which represents a given predicate. The important concept these research efforts have in common with ours is the emphasis on *selective specification recovery* rather than extracting complete specifications. In our work, developers progressively select aspects of the system to be captured by formal specifications to enable, among other things, model-driven synthesis and formal analysis.

Model-driven Modernization. Model-driven modernization [3] is about migrating from heterogenous implementation technologies to the homogenous world of models, from which everything is generated. The initial step though would be to obtain representative models of the legacy systems. While our approach is built upon reverse engineering techniques used in this area, it is different in several ways. First, they rely on after-the-fact model extraction from an already developed application. In contrast, our work is geared towards the application of an iterative model abstraction during the software development lifecycle, as opposed to a one-time reverse engineering for software *modernization* of legacy systems. Second, model is the only first class citizen in theirs, while in our approach code base is the main place for learning and modeling is a means to capture obtained knowledge from code.

Partial Code Synthesis. Recent research [17], [7], [5] recognize the potential benefits of the emerging class of partial code synthesis in different domains, where partial models generate partial code frameworks which are then combined with hand-written code to constitute the application. Among others, Zheng and Taylor [17] proposed *1.x-way mapping* approach for partial synthesis from architectural models, which supports synthesis from both structural and specific type of behavioral specifications. It also provides a deep separation model that puts synthesized code in classes separate from hand-written code. This paper, however, shows that partial synthesis techniques can provide a way of addressing the empirical challenge that developers face when attempting to realize the MDD [2].

V. Conclusion

This paper contributes a philosophical, technological, and methodological attempt to address the *empirical* critique of model-driven development. Philosophically we offer a *scientific-empirical* stance: as with scientific theories, useful models emerge bottom-up as *partial* abstractions from knowledge of the empirical domain. Technologically we offer *partial synthesis* as a key to using partial models for synthesis. Methodologically we propose an iterative approach in which a combination of hand-crafted and synthesized artifacts evolves under two-way refactoring. Early validation through experience of applying these ideas to a healthcare-related experimental system in our lab supports the claim that it promises many benefits of MDD, in intellectual control, reliability, and productivity, while escaping the rationalist trap.

Acknowledgments

We thank Anthony Finkelstein for discussing his position with us. All positions we take in this paper, including any errors, are ours. This work was supported in part by the National Science Foundation under grant #1052874.

References

[1] Pol tool suite. http://www.cs.virginia.edu/~hb2j/Downloads/Pol.zip.
[2] Anthony Finkelstein. Bottom 10 software engineering challenges. http://blog.prof.so/2012/06/bottom-10-software-engineering.html, 2012.
[3] Architecture-Driven Modernization Task Force. Architecture-driven modernization (adm). http://adm.omg.org/, 2006.
[4] H. Bagheri and K. Sullivan. Monarch: Model-based development of software architectures. In *Proc. of MODELS'10*, pages 376–390, 2010.
[5] H. Bagheri and K. Sullivan. Pol: Specification-driven synthesis of architectural code frameworks for platform-based applications. In *Proc. of GPCE'12*, pages 93–102, 2012.
[6] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proc. of PLDI*, 2001.
[7] Damien Cassou, Emilie Balland, Charles Consel, and Julia Lawall. Leveraging software architectures to guide and verify the development of Sense/Compute/Control applications. In *Proc. of the ICSE*, 2011.
[8] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27:213–224, 2001.
[9] G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *Proc. of OOPSLA'06*. 2006.
[10] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *Proc. of ICSE*, 2000.
[11] D. Jackson. Alloy: a lightweight object modelling notation. *TOSEM*, 11(2):256–290, 2002.
[12] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Softw.*, 12(4):52–62, 1995.
[13] Martin Fowler. Model driven software development. http://martinfowler.com/bliki/ModelDrivenSoftwareDevelopment.html, July 2008.
[14] A. McNeile. MDA: the vision with the hole?, 2003.
[15] Scott Ambler. Examining the MDA. http://www.agilemodeling.com/essays/mda.htm.
[16] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.
[17] Y. Zheng and R. N. Taylor. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In *Proc. of ICSE*, pages 628–638, 2012.