Shadow Models: Incremental Transformations for MPS

Markus Voelter
independent/itemis
Stuttgart, Germany
voelter@acm.org

Klaus Birken
itemis AG
Stuttgart, Germany
birken@itemis.de

Sascha Lisson
itemis AG
Stuttgart, Germany
lisson@itemis.de

Alexander Rimer
itemis AG
Stuttgart, Germany
rimer@itemis.de

Abstract

Shadow Models is an incremental transformation framework for MPS. The name is motivated by the realization that many analyses are easier to do on an model whose structure is different from what the user edits. To be able to run such analyses interactively in an IDE, these "shadows" of the user-facing model must be maintained in realtime, and incrementality can deliver the needed short response times. Shadow Models is an incremental model transformation engine for MPS. In the paper we motivate the system through example use cases, and outline the transformation framework.

CCS Concepts • **Software and its engineering → Application specific development environments**; *Domain specific languages.*

Keywords domain-specific languages, model transformations, incrementality, language workbenches, MPS

ACM Reference Format:
Markus Voelter, Klaus Birken, Sascha Lisson, and Alexander Rimer. 2019. Shadow Models: Incremental Transformations for MPS. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3357766.3359528

1 Introduction

A problem when representing information formally with models is that different tasks suggest different representations of the same information: one particular abstract syntax might be useful for the user when editing the model, a second representation might be more suitable for a particular analysis, and a third one might suit execution. It is a well-known approach in any number of tools, including compilers, to transform a source model into several intermediate representations for particular kinds of analyses, and ultimaltely, execution.

To be maximally useful, the results of analysis should be available to the user while she edits the model. This is useful to interactively guide the editing process (through realtime analysis feedback) or by executing the program directly (live programming [11]). This requires that the representation that suits the particular analysis is maintained as the user edits the program. For all but the computationally cheapest transformations and analyses, this requires incremental maintenance (and ideally, analysis) of the derived representations: the user makes an edit to the input model, the change is propagated to the transformation engine, the target model is updated incrementally, the analysis is performed, and then the (incrementally updated) analysis results are piped back up to the user. This can potentially be done in multiple steps (to form a pipeline), and one might also want to maintain several shadow models from a single source.

> Shadow models is an incremental model transformation language and engine, fully integrated into MPS.

The paper gives an overview of the framework, prototypical use cases and for future evolution.

2 Use Cases

2.1 Growing Domain-Specific Languages

An important approach for developing languages is to grow a specialized language from a more general one ([6, 13]. The semantics of extensions is defined through reduction to the base language.

Because of MPS' rich support for language modularity, this approach is idiomatic. For example, mbeddr extends C with domain-specific concepts for embedded software

SLE '19, October 20–22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6981-7/19/10...$15.00
https://doi.org/10.1145/3357766.3359528

```
transformation t1 overrides Desugar.desugar [ alt: AltExpr ]
-> [ o0: foldR alt.cases, NeverLit {  }, IfExpr {
         cond: copy it.cond
         thenPart: copy it.val
         elsePart: ElsePart {
           expr: acc
         }
       }                                              ]
```

Figure 1. Example transformation from a multi-case `switch`-style expression to nested `if` expressions.

development [20] and KernelF [17] has been used as a base language for DSLs in finance and healthcare [18].

In the case of mbeddr, the reduction to the base language happens on demand when the user invokes `Make` in the IDE. However, in addition to compilation to Java, KernelF is also executed with an in-IDE interpreter to shorten the feedback cycle and reduce the need for external build and execution infrastructure. So for every DSL language construct, the language engineer has to develop both a code generator to Java *and* an interpreter. This duplication is tedious and error-prone.

Why Realtime KernelF2 is a minimal functional language with an interpreter and a code generator. When extending KernelF2, the semantics of the extension is defined through a *single* Shadow Model-based transformation to KernelF2. We define the semantics *once* and get a generator as well as an interactive interpreter. Our vision also includes various verifiers as backends for the language. Fig. 1 shows an example transformation.

2.2 Code Weaving for Safety

A consequence of using separation of concerns to reduce code complexity and increase modularity is that for the final system, the previously separated concerns have to be reintegrated. In the context of SAFE4I [1] we use Shadow Models to incrementally weave safety concerns into C programs written in mbeddr [19].

Separating the safety concern is feasible because most safety measures rely on a limited number of established patterns such as checksums or redundant computation with subsequent voting [8]. This way, the core logic and the safety patterns can evolve independently and can be rewoven on demand. In addition, the same pattern can potentially be applied to many different target locations. It also fits well with a development process that distinguishes between safety engineers and (regular) embedded developers: each can maintain their own artifact.

Safety engineers use a DSL to specify *safety patterns* modularly. The pattern describes the constraints regarding a potential *weaving site* (in terms of structure, type system and data flow), plus the modifications to the core code. The embedded software engineers mark the locations in their code where a particular safety pattern will be woven in. Finally, a *weaver*, implemented as a Shadow Model transformation merges the two concerns.

Why Realtime A drawback of SoC is that it requires reassembling the overall system from the separated artifacts. To minimize this drawback, it is useful to show the weaving result to the user. The shorter the feedback, and the lower the requirements on the build infrastructure the better. This is especially true because some of the weavings are non-trivial; it is useful to show the result and give the safety engineer the opportunity to fix potential problems.

2.3 Incremental Staging of Feature Models

Feature modeling is well-established for modeling variability in product lines [9]: a *feature model* specifies the set of possible products by defining identifiable features and the constraints between them; *configurations* specify individual products by selecting features while respecting the constraints. The formalism comes with a set of predefined constraints (such as `mandatory`, `optional`, `n-of-m` and `1-of-m`) but also allows custom constraints using Boolean expressions.

Sometimes the product is configured in steps, each step making additional selections; only the final step defines a concrete product. Such *staged configurations* [2] are typically used along a supply chain or to distinguish between build-time and runtime configuration decisions.

We implemented feature models in MPS as a building block for customer-specific modeling environments. In addition to staged configuration, the tool also supports attributes, modularity via instantiation, and cardinalities [3]. The tool uses the Z3 SMT solver [4] to check consistency of feature models, and interactively guide the user towards valid configurations.

Creating a partial configuration `C` of feature model `M` implicitly defines [2] a specialized feature model `M'` by removing all features, attributes and constraints that became redundant due to the user's decisions in `C`; it also leads to specialization of constraints, for example, $F_1 \Rightarrow F_2 \vee F_3$ will be specialized to $F_1 \Rightarrow F_3$, if F_2 has been deselected in `C`. In the next stage, a more specific configuration `C'` is derived from the specialized feature model `M'`. The creation of the derived feature models is implemented via Shadow Models.

Why Realtime The specialized feature model becomes available right after each user decision. This has several benefits for the user: (i) for each user decision the impact on the resulting feature model is immediately visible; (ii) the user understands at all times which downstream decisions are still open; and (iii) the solver checks on the derived feature model provide additional insights, e.g., if the specialization leads to redundant constraints.

[1] https://www.edacentrum.de/safe4i/, BMBF FKZ 01|S17032

3 Framework

3.1 The Core Transformation Framework

The framework consists of five components: the transformation DSL, an engine for incremental computations, the transformation engine itself, an integration with MPS' model repository and various IDE integrations.

Transformation DSL The language is functional: each function takes one or more source nodes as input and produces one or more output nodes. Functions are polymorphic in all arguments and support multimethod-style dispatch [12]. The DSL exploits MPS' strength regarding language extension and composition: queries and low-level expressions reuse MPS' Java implementation and model access APIs. They need not be declarative, because dependency analysis happens dynamically at runtime. Reference resolution is based on (cached) re-invocation of transformation rules or explicitly defined labels; we cover this in more detail in Section 3.2. Finally, there is syntax to help with lifting analysis results from the target model back to the source(s). These are functions implemented as part of a transformation rule that attach error messages to the input of a rule when particular errors are present on the output.

Incremental Computation Engine The core engine is similar to Adapton [7]: the engine caches the result of function calls and records dependencies on other functions and mutable data for invalidation after a change. Computations are lazy: a transformation is only executed if the particular (part of the) result is accessed. This makes it suitable for IDE services where only the currently edited part of the input model is relevant to the user. Essentially, Shadow Models map the domain of graph transformations to the general notion of incremental computations as implemented by Adapton.

Transformation Engine The core engine expects computations to be expressed as pure functions whose results can be cached. Thus, each transformation rule expressed with the DSL is generated into a function that returns a fragment of the final output graph. Each fragment is connected to other fragments by a specification of the transformation rule and the parameter values.

The engine works on an internal data structure that is independent of MPS and uses a dynamically-maintained dependency graph to detect changes; a change to a dependency triggers a retransformation.

MPS Adapter The model data structure in MPS requires transactions for read and write access. The projectional editor of MPS directly writes user input to the model and updates the UI by rendering the updated model. Long running transactions, such as transformations, will block the editor's write transaction, resulting in an unresponsive UI.

To decouple the transformations from the repository (and hence the editor), the first step in the transformation chain mirrors the MPS model into a persistent copy-on-write (COW) data structure [5] that allows reads without blocking writes. Because the MPS projectional editor broadcasts change events anyway, maintaining this copy is computationally cheap; no expensive diffs are required.

The result of the transformation can either be analyzed directly on the `INode` structure or after materializion to an MPS AST (through another COW). The latter is slower, but has the advantage that existing MPS analyses (such as type checks) can be used unchanged; it is also the basis for visualization in the editor.

IDE Integration Shadow Models is fully integrated into MPS. The DSL comes with editor support and type checking and is available as a language aspect (similar to the native MPS generators or type system specifications). The target models can be opened in MPS editors; editing is not possible, because this would require some form of bidirectionality, which Shadow Models do not support.

A new entry in the MPS project view, called the Shadow Repository, shows all the incrementally maintained models. Results of analyses on the target nodes can, after lifting, be annotated to the source nodes (red squigglies, markers in the gutter). Finally, there is a debugger that shows which transformation operated on which input nodes, created which outputs and ran in which forks (explained next).

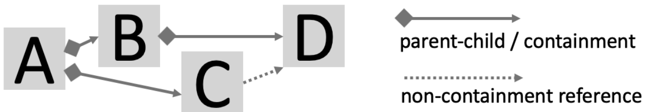
3.2 References, Forks and Eagerness

MPS models are trees with cross-references (or: graphs with a single containment hierarchy). Those cross-references are particularly challenging: a reference of some type `P` between input nodes `A` and `B` must be mapped to a reference of some type `Q` between the corresponding output nodes `A'` and `B'`. To obtain `B'` from `B` in the transformation that transforms `A`, one can invoke the transformation `T` that maps `B` to `B'` again; because of caching, `B'` is not created a second time.

Labels However, for reasons of modularity, you might not want to know `T`. To achieve this, the language supports transformation labels, named mappings between nodes. The transformation `T:B->B'` would populate a label `L`, and other transformations can find `B'` knowing `B` and `L`. This way, labels are a kind of interface.

Laziness While this approach enables transformation modularity, it conflicts with laziness: to be able to retrieve `B'` from `L`, the label must already be filled; lazy computation will not work because `L` cannot know the transformation that fills it – ignorance of this dependency was the reason for labels in the first place. A static analysis might reveal the transformation, but not the (runtime) input parameters.

More generally, the research roadmap by Kolovos et al. [10] identifies laziness as a core challenge in the context of graph transformations. The problem is that in a lazy system, less information is available at runtime because some parts of a transformation have not yet been executed; the issue with labels is an example. Another example is that the parent of a node in the output model might not yet be available when a node is accessed via a reference. Consider this graph:



If we follow the path `A.C.D.parent`, the node `B` will not yet be available because it is lazily computed when following the `A.B`; the transformation describes the parent-child relationship only from the parent to the child.

Forks Our solution is to compute results eagerly, but only in demarcated regions called forks. The transformations inside a fork are executed eagerly; labels can be used to look up targets, the `parent` can be retrieved. From the outside, the whole fork is lazy and when referencing nodes inside a fork from the outside, the lookup has to specify the fork. Effectively, the fork becomes part of the identify of the nodes created inside the fork.

Another consequence of the approach is that it is now possible to run a transformation multiple times, creating outputs with different identities, without adding an additional parameter to all involved transformation rules. This requirement was driven by the code weaving use case Section 2.2, where the same pattern has to be woven into target locations, and references must be resolved "locally" at each weaving site.

Finally, a fork can be marked as `fixpoint`, which means that transformations are eagerly executed until no more rules apply; this requirement was driven by the KernelF2 use case (Section 2.1), which requires that extensions are reduced stepwise, until only base language concepts remain, similar to a term rewriting system [1].

Summing up, we do *not* solve the general problem of references and laziness: we revert to eager transformations. However, using forks, we limit the eagerness to well-defined scopes, and retaining the lazy nature of the overall transformation. Initial exerpience suggests that this compromise works in terms of performance and scalability, but further evalution is necessary.

4 Related Work

For space reasons, we compare only superficially to a few related approaches. The **MPS Build Pipelines**, although using model-to-model-transformations, is not incremental. Unsuccessful experiments with running it interactively prompted the development of Shadow Models. **Incremental** transformations are not a new idea; for example, **VIATRA2** [16] supports incrementality based on the **IncQuery** [15] incremental graph pattern matching engine. **Dclare for MPS** is another incremental transformation engine that relies on contraints instead of functional transformations. Shadow Models is not **bidirectional** [14]; it supports unidirectional transformations that maintain a trace back, as well as specific APIs to propagate analysis results back to the source. Our use cases do not require true bidirectionality, and we decided to go with the simpler specifications that come with unidirectional transformatioms.

5 Future Directions

Based on experience from the projects described in Section 2, we have identified several areas of improvement.

Scalability Incremental transformations are useful especially for large models; for small ones, rerunning transformations from scratch is feasible. Although our initial experience is promising, we will have to characterize the scalability in terms of shadow update time and memory use more thoroughly, and then identify strategies for optimization of the engine. A comparison with Dclare and IncQuery is part of this.

Scope of Change Tracking Right now, all models in the MPS workspace that use languages with Shadow Model transformations are tracked and transformed, even though the user might only be interested in a subset. This can lead to unnecessary memory consumption. We will add a way to define a scope within which change tracking and transformation should be active.

Language Abstractions The current language exposes several engine internals (such as forks) that are hard to understand for users. We will abstract them into concepts that are less technically motivated and easier to explain.

Improved Lifting Currently, lifting of results to the input model is expressed using generic callback functions; a more concise, more declarative syntax will be provided.

Extract the Tracking Engine The incremental computation capabilities of the core engine can be used for other purposes. In particular, we plan to implement an incremental interpreter based on the same framework. This will allow clients such as KernelF2 to not just incrementally maintain the desugared shadow model, but then also run this model incrementally (as long as it is functional), achieving a fully interactive, Excel-style reactive programming environment.

6 Conclusions

At itemis we have had this situation for a few years now: whenever we start to talk about some new end-user relevant feature, it takes only a few minutes until we end up with Shadow Models as an important part of the solution; we have several additional concrete use cases in mind beyond those described in Section 2. And as we have outlined in Section 5 there is still work to do. However, our initial experience is promising, and we see many of the benefits of Shadow Models that we had hoped for.

References

[1] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge university press, 1999.

[2] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In R. L. Nord, editor, *Software Product Lines*, pages 266–283, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[3] K. Czarnecki and C. H. P. Kim. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories at OOPSLA'05*, San Diego, California, USA, 2005. ACM, ACM.

[4] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[5] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[6] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, 2018.

[7] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *ACM SIGPLAN Notices*, volume 49, pages 156–166. ACM, 2014.

[8] R. S. Hanmer. *Patterns for fault tolerant software.* John Wiley & Sons, 2013.

[9] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[10] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, et al. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 2. ACM, 2013.

[11] S. McDirmid. Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 53–62. ACM, 2013.

[12] T. Millstein and C. Chambers. Modular statically typed multimethods. In *European Conference on Object-Oriented Programming*, pages 279–303. Springer, 1999.

[13] G. L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[14] P. Stevens. A landscape of bidirectional model transformations. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 408–424. Springer, 2007.

[15] Z. Ujhelyi, G. Bergmann, Á. Hegedüs, Á. Horváth, B. Izsó, I. Ráth, Z. Szatmári, and D. Varró. Emf-incquery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99, 2015.

[16] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the viatra framework. *Software & Systems Modeling*, 15(3):609–629, 2016.

[17] M. Voelter. The design, evolution, and use of kernelf. In *International Conference on Theory and Practice of Model Transformations*, pages 3–55. Springer, 2018.

[18] M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Alff, L. Wiart, A. Wortmann, and A. Nordmann. Using language workbenches and domain-specific languages for safety-critical software development. *Software & Systems Modeling*, pages 1–24, 2018.

[19] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):1–52, 2013.

[20] M. Voelter, A. van Deursen, B. Kolb, and S. Eberle. Using c language extensions for developing embedded software: A case study. In *OOPSLA 2015*, 2015.