A Domain Specific Aspect Language for Run-time Inspection

Wouter De Borger Bert Lagaisse Wouter Joosen
IBBT-DistriNet, Department of Computer Science, KU Leuven, Belgium
[wouter.deborger,bert.lagaisse,wouter.joosen]@cs.kuleuven.be

Abstract

When inspecting a running system, be it for debugging or monitoring purposes, developers are often faced with an abstraction gap: the run-time structure is not represented in terms of high-level programming abstractions, but in terms of low-level run-time constructs.

To present developers with an understandable view on the system, a transformation can be performed to restore the programming abstractions. In the current state of the art, two types of transformations exist: state-based transformations (model transformation) and event-based transformation (complex event processing). These two types of transformations can bridge the same abstraction gap, but deliver a different quality of service.

There are domain specific languages for these two kinds of transformations, but they can not be composed into a single overarching transformation automatically. While such unified transformations can deliver a superior quality of service, there is no language to express them. Therefor we are currently working on a unified model, that considers both types of transformation as aspects contributing to a unified transformation. This papers sketches our approach to unifying these two types of declarative languages.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

1. Introduction

Advances in composition technology allow more efficient development of software systems, by enabling more powerful abstraction. At run-time, however, when all components have been woven together, the full complexity of the system is exposed.

To restore abstractions, a transformation is required, that searches for patterns of synthetic code corresponding to high level abstractions [3, 4]. Currently two approaches exists to specify such transformations: state-based and event-based.

Currently, these two aspects of the transformation can not be unified. Both are specified in their own domain specific language and they can not be composed. This severely limits the ability of transformation systems to deliver the required quality of service. In our previous work [2], we focused on making state-based transformation applicable to run-time inspection. In this paper, we outline our approach to integrating complex event transformation as an aspect into these transformations.

We are currently working on an approach that allows both types of transformation to be composed as aspects. This allows unified transformation to be expressed in an elegant way, allowing reuse and simplifying development. Additionally, mixed mode transformation, combining both state and events, will have the advantage that they can dynamically switch between both types of transformation, allowing a better quality of service.

2. Context

State-based transformations define how concrete run-time state relates to the abstract state we wish to represent. The advantage of state-based transformation is that they can accurately show the current state. The disadvantage is that they can not perceive change. State-based transformations have already been described in the context of model-to-model transformations [1, 8].

However, the quality of service delivered by these transformation engines is not sufficient for run-time inspection. The existing approaches assume the entire model has to be transformed at once. In other words, it is not possible to selectively inspect a small part of the run-time state. The entire state must be transformed at once. As the run-time state of most programs is quite large, this would result in systems which are too slow. We addressed this problem by providing an alternate execution mechanism for the existing QVT language. Our solution supports lazy and on-demand execution of QVT transformations [2].

Event-based transformations have the opposite properties. Such transformations relate sequences of low-level events to high level events. This approach supports to track change, but makes it impossible to perceive parts of the sys-

tem that are not currently changing. Complex event processing has already been exhaustively examined in the field of run-time inspection [6, 7, 9].

For example, consider a stack trace of an aspect oriented system using the meta-aspect protocol (MAP) [5]. (See Figure 1). The stack trace of such systems is quite complex. Each joinpoint calls into the meta-aspect framework, where appropriate advices are selected and executed. As such, each call stack may contain a lot of MAP-framework code. Programmers using only base language facilities, would like to have these stack frames represented in a more abstract way .

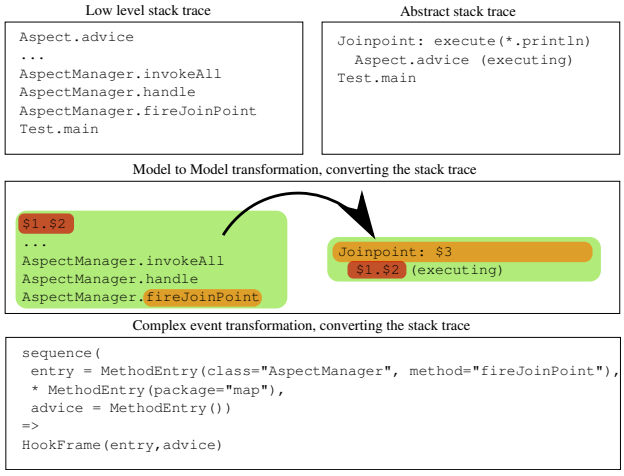

Figure 1. Overview of model transformation approaches

To do so, we may define a transformation that analyzes the stack trace. It walks up the stack, frame by frame, until it encounters the first framework call. It recognizes this frame and knows it may now skip a part of the stack trace, up to the first non-MAP frame.

Alternately, we may do the exact same thing using events instead of state. In terms of events, the stack is the list of all methods that have been entered, but not returned. By keeping track of all method entries, we can start dropping events when the *'map entry event'* is received and stop dropping events if the *'enter non-map event'* is received. This produces the same result.

As such, it is clear that events and state can provide the same information, but with a different QOS. The event-based approach can not show the part of the stack that has been built up before it started receiving events, while the state-based approach can not signal the exact moment the MAP is entered.

3. Vision

What we envisage in the future is to combine both type of transformations, as if they are aspects of the same transformation component. From a high level view, our approach is to consider the state transformation as the domain specific base language and the event transformation as a set of aspects.

Both state and event transformations define a set of relations between a low level an a high level model. An event signals the change of a relation in the corresponding state model. Each such change can be propagated through the state transformation to produce a number a high level events.

An important benefit of declarative transformations (both event and state-based) is that if facts can be derived in multiple ways, it does not degrade performance to specify both. When the transformation is compiled, this redundancy can be detected. The compiler can then choose to either discard the redundant derivation and produce faster code or maintain it and produce more robust code, that can detect model inconsistencies.

As such mixed mode transformations can be used to automatically synthesize a best of both worlds implementation, that switches between state and events at the appropriate time. Alternately, in order to validate the correctness of the model, an implementation can be generated that executes both models at the same time, allowing runtime verification of model consistency. As such, mixed mode transformation combine the strength of both individual techniques.

However, there is one other important advantage: **it is possible to specify parts of the transformation in terms of only events or state. When it is convenient, one can switch notations.** In the worst case, this will force the engine to generate suboptimal code. In the best case, the engine will be capable of converting one notation to the other.

For example, in our stack trace example, the event-based notation is much more elegant. Should we have specified the state-based pattern as code instead as graphical, it would have taken up the better part of the page. However, collecting events to build a stack trace is very inefficient. In a unified model, where the transformation engine knows that each method entry event adds an element to the stack, it is possible to derive the state-based transformation from the event-based transformation. This allows both an elegant specification and efficient execution.

4. Challenges

Both complex event processing and model to model transformation systems are fairly complex paradigms in their own respect. This presents us with two challenges.

4.1 Fundamental limitations

In the current state if the art, there is not much formal work on unifying cross cutting transformations. This makes it hard to estimate what is theoretically possible. When only considering the core of both types of transformation, unification seems to be quite feasible. By restricting both to a subset of invertible transformations, the equivalence between them can be exploited easily.

However, when considering the existing languages, it is less obvious what are the limitations and abilities of mixed mode transformations. As in all languages, there is an important trade-off between expressiveness and usability. Allowing more expressive languages will make many automatic analysis steps infeasible, placing more responsibility with the programmer. At the other hands, less expressive languages will allow more automatic verification. Based on the current theoretical understanding of the matter, it is hard to determine what the sweet spot is between expressiveness and usability.

4.2 Consistency

For efficiency reasons it is important to assume that both parts of the transformation are correct and consistent. This allows us to execute only one (or a part of one) transformation instead of both and still be confident about the quality of the result. To practically enforce this assumptions, a minimal level of consistency checking is required, both at compile time and at run-time. In this respect, it is comparable with type checking. However, it may prove to be quite hard to prove that two transformations are consistent. Not much formal work exists in this field.

One important consistency guarantee we will enforce is that each high-level event that can be derived by propagation of a low level event through the state transformation must be declared. It is easy for this property to derive an upper and a lower bound: the upper bound is that all conditions in the state transformation propagate the change, the lower bound is that none of them propagates the change. The part below the lower bound can be checked statically, for the part between the bounds, dynamic checks can be generated.

5. Approach

We are currently working on the basic semantics of mixed mode transformations. We are constructing a suited subset of the whole problem, that is not overly complex, but suited for practical validation. Currently, we envisage a three step plan.

1. In the initial stage, we consider relations in both languages as black boxes. Both transformations specify the whole system. The only mode of operation is to start with a state transform. Once an initial model is constructed, events keep it up-to-date. We require very strong consistency between models, where events are drop-in replacements for state transformations. The goal of this stage is to make an initial measurement of the capabilities and limitations of mixed mode transformations.
2. In the second stage we allow automatic analysis of the state transformation. Events can now also replace subparts of relations. Events may now refer to objects that have not been created by the state transformation. The goal of this state is to establish the importance of revertible relations in practical scenarios. This is important to assess how far automation can be pushed and at what cost.
3. In the final stage, model analysis encompasses both models. The limitations present in this system strongly depend on the lessons learned in the previous stage.

6. Conclusion

We briefly outlined our ongoing work on mixed-mode transformations, that encompass both event and state-based components. To achieve this we compose existing state and event-based transformations as if they are aspects of the same system.

From a perspective of domain specific aspect languages, this is an interesting case: two paradigms are in the most literal sense cross-cutting each other. They have the same domain of discourse, but represent a different dimension of it. Both base language and aspect language are executable on their own, but may contribute to each other in non-trivial ways.

While the approach is still immature, it may provide significant benefits. Mixed mode transformations offer programmers a more expressive syntax, combining both type of transformations and enable a better quality of service, by dynamically switching between both styles of transformation.

References

[1] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[2] W. De Borger, B. Lagaisse, and W. Joosen. A generic solution for agile run-time inspection middleware. In *Middleware '11*.

[3] W. De Borger, B. Lagaisse, and W. Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *In Proc of AOSD 8*, pages 173–184. ACM, 2009.

[4] W. De Borger, B. Lagaisse, and W. Joosen. Traceability between run-time and development time abstractions. In O. G. Jane Cleland-Huang and A. Zisman, editors, *Software and Systems Traceability*. Springer, 2011.

[5] T. Dinkelaker, M. Mezini, and C. Bockisch. The art of the meta-aspect protocol. In *AOSD '09*, pages 51–62. ACM, 2009. ISBN 978-1-60558-442-3.

[6] D. C. Luckham and B. Frasca. Complex event processing in distributed systems. Technical report, Stanford University, 1998.

[7] B. P. Miller and A. V. Mirgorodskiy. Diagnosing Distributed Systems with Self-propelled Instrumentation. *IFIP LNCS*, 5346(5346):82–103, 2011.

[8] OMG. Meta Object Facility (MOF) 2.0 Query View Transformation. http://www.omg.org/spec/QVT/1.0/.

[9] G. Wilkin, K. Jayaram, P. Eugster, and A. Khetrapal. FAIDECS: Fair Decentralized Event Correlation. In *Middleware 2011*, volume 7049, pages 228–248. Springer, 2011.