# Adding a HenshinEngine to GEMOC Studio

An experience report

Steffen Zschaler
Department of Informatics, King's College London
London, UK
szschaler@acm.org

## ABSTRACT

Executable DSMLs (xDSMLs) are becoming more popular as a means of efficiently building domain-specific high-level models that are testable through simulations. To develop an xDSML, one needs to provide the abstract and concrete syntax, but also the operational semantics of the language. GEMOC Studio provides a language workbench for developing xDSMLs and their associated tooling. To date, it supports the expression of operational semantics in a number of imperative formats. GEMOC Studio is intended to easily support also operational semantics in other formats, but this idea has so far not been tested. In this paper, I report on my experience implementing a new execution engine for GEMOC Studio that allows operational semantics to be expressed declaratively using graph-transformation rules written in Henshin. I hope that this experience report helps to (a) explore the current flexibility of GEMOC Studio and provide insights into areas that may benefit from refactoring, and (b) give some guidance to others wishing to develop their own execution engines for GEMOC Studio.

## 1 INTRODUCTION

Executable modelling languages have been discussed for some time [9], but recently have seen increased interest through projects such as GEMOC[1] and MOLIZ[2]. The key idea is that by specifying operational semantics in addition to abstract and concrete syntax, simulators and other dynamic analysis tools can be easily (and possibly automatically) derived for these executable domain-specific modelling languages (xDSMLs).

The key issue, then, is how to specify the operational semantics for an xDSML. Two different approaches have been taken: (1) Using imperative definitions—for example added to the abstract syntax through new operations [1] or by defining fUML-based actions—and (2) Using declarative definitions—typically using endogenic graph-transformation systems (GTSs) [2, 10].

Imperative definitions are close to how programmers think and can be expressed in languages programmers are already familiar with (*e.g.,* Kermeta-based languages use Xtend to express individual operations). However, their imperative nature also makes it more difficult to reason about the semantics of such languages and to safely compose different language semantics. The primary means of composition of languages with imperatively expressed operational semantics is by a variant of role-based composition [13], but this cannot easily provide guarantees about semantic preservation.

On the other hand, declarative definitions of operational semantics using graph transformations (*e.g.,* [2, 10]) may be less immediately intuitive for programmers, but offer strong reasoning opportunities, including giving guarantees for semantics preservation in the face of composition [4]. However, they are currently somewhat underrepresented in the xDSML literature.

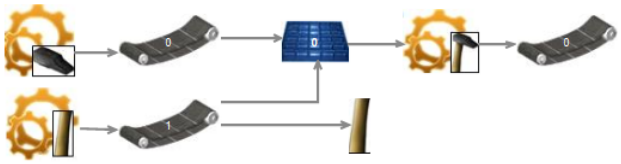

**Figure 1: A production-line system model**

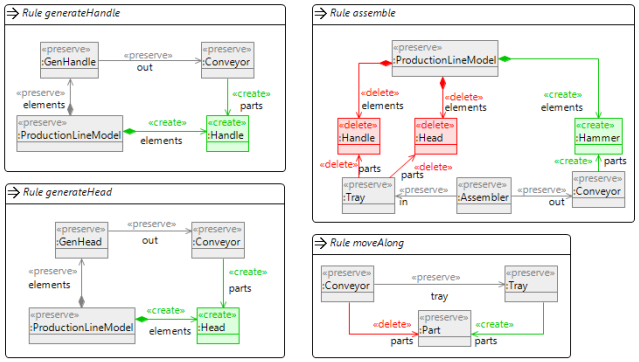


**Figure 2: Henshin rules for PLS-system semantics**

Figure 1 shows an example model in a (toy) language for describing production-line systems [5]. I show here only the concrete syntax in a Sirius-based diagram. The language allows modelling the flow of parts (heads, handles, and hammers) through a production-line system. In Fig. 1, for example, we can see that a handle has been produced and is currently waiting to be processed by the assembly machine. Figure 2 shows the rules defining the operational semantics of this language. The two `generate. . .` rules define the behaviour of generators, while the `assemble` rule defines how hammers are assembled. Finally, rule `moveAlong` defines how parts are transferred from conveyors onto trays.

In this paper, I report on my experience building a new execution engine for the GEMOC Studio [1], a state-of-the-art language workbench for xDSML development. The resulting implementation is available on Github[3]. One aim of the GEMOC Studio was to easily support the integration of different forms of operational semantics, but so far this has only been tested for imperatively defined semantics. Here, I explore what it takes to extend GEMOC Studio with support for xDSMLs with GTS-defined semantics. Specifically,

[1] http://gemoc.org/
[2] http://www.modelexecution.org/?page_id=2
[3] https://github.com/szschaler/henshin_xdsmls

I report on building a GEMOC `HenshinEngine`, for semantics expressed using the popular Henshin graph-transformation tool [11].

Beyond the execution engine itself, I hope that this paper can make two contributions to the community:

(1) Provide some insights into the current flexibility of GEMOC Studio and, potentially, some lessons about which areas may benefit from further refactoring and generalisation; and
(2) Provide some guidance to others integrating new execution engines for operational semantics expressed in other formats.

The remainder of this paper is structured as follows: In Sect. 2, I introduce the key elements of the GEMOC-Studio architecture that are relevant to the introduction of a new execution engine. This is followed, in Sect. 3, by a description of key aspects of the HenshinEngine implementation and some lessons learned in Sect. 4. Finally, Sect. 5 concludes the paper.

## 2 GEMOC STUDIO

GEMOC Studio is a language workbench for developing xDSMLs [1]. It provides an interpreter infrastructure that takes a language with abstract syntax defined in Ecore [6], concrete syntax defined in Sirius [12], and an operational semantics that can be interpreted by one of the existing execution engines. From this, the infrastructure provides integration with Eclipse's launch mechanism so that models can be executed like other programs. When models are executing, GEMOC can use Sirius to provide visual animation of execution progress as well as language-agnostic debug functionality.

GEMOC Studio currently offers execution engines that can interpret language semantics defined in Kermeta, fUML, and a concurrency coordination language called BCOoL [7]. An execution engine, essentially, performs the following steps:

(1) Initialise engine, using the model already pre-loaded by GEMOC Studio (see below);
(2) Initialise model, if needed—for example to create initial dynamic-state elements; and
(3) Execute individual semantics steps in a loop.

The final step in the execution is the central step. For existing engines, this invokes a so-called execution-entry point, an operation of one of the model elements, specially annotated to mark it as the entry point. This will usually run a loop in which individual steps, operations annotated as semantic steps are executed. For each step, GEMOC inserts some functionality before and after the step which allow it to trace model execution and provide additional functionality, including debugging and animation, on top of the execution. For GTS-based xDSMLs such an entry-point method does not exist. Instead, we will need to provide our own interpretation of steps and communicate these to GEMOC Studio. To this end, we can build on functionality encapsulated in `AbstractSequentialExecutionEngine`: before each step is executed, we invoke a `beforeStep` method. After each step is executed, we invoke an `afterStep` method. These two calls enable GEMOC Studio to trace model execution as well as offer users control and allow them to stop execution at any time.

## 3 CREATING THE HENSHINENGINE

I consider the semantics of a model to be given by a graph grammar with the model as the starting graph, and the metamodel and the semantics rules as the graph-transformation system.[4] Thus, any sequence of rule applications from the starting model is considered to be a valid trace in the semantics. Such semantics can be non-deterministic. Non-determinism is useful to represent the degree of concurrency in our problem: heads and handles are generated independently of each other, and hammers are produced independently of the generators, as long as sufficient numbers of heads and handles are available.

In my execution engine, I implement these semantics by executing *one randomly chosen trace*[5]. As a result, two individual executions of the same model may produce different results.

Deciding which rule to apply is a two-stage process: we first randomly pick a rule and then we pick a random match for this rule in the current model. An arbitrarily picked rule may well not have a match in the current model, so we need to know about rule matches before picking rules. Checking whether a rule matches is, however, the computationally hard part of graph transformation, so we want to do as little as possible of it. The easiest solution would be to hand all of this work to Henshin, by asking the matching engine to try to apply a randomly chosen rule: if the application succeeds a step has been taken and we move on. Otherwise, we randomly pick a different rule to try. If none of the rules can be applied, the execution stops. In the process, we make use of Henshin's capability for non-deterministic matching, which ensures we get a different match for rules with multiple matches at every execution.

However, GEMOC Studio requires a step operation name *before* executing a step[6]. With the above naïve implementation, we can only provide a generic name, such as '`invokeRule`' as we do not yet know *which* rule will be invoked. We solve this by splitting the processing of a single step into two sub-steps: (1) determine the rule and (non-deterministic) match to apply; (2) apply the identified match. We then call `beforeStep` between these two sub-steps and can thus provide more precise tracing information. Algorithm 1 shows a summary of the core execution-engine behaviour.

The current launch infrastructure of GEMOC Studio assumes Melange-defined languages [3]. Melange only supports a limited

**Algorithm 1** `performStep` algorithm

```
1:  function findNextMatch
2:      rules ← semanticRules
3:      while rules not empty do
4:          rule ← remove random element from rules
5:          match ← find match for rule
6:          if match not null then
7:              return match
8:      return null
9:  procedure performStep
10:     match ← findNextMatch
11:     if match not null then
12:         beforeStep(match.rule.name)
13:         apply match.rule at match
14:         afterStep
```

[4] A graph grammar defines a language of graphs from a starting graph and a graph-transformation system. Any graph reachable from the starting graph is considered part of the language.
[5] See the discussion in Sect. 5 for other options
[6] In the call to the `beforeStep` operation

number of ways in which language-semantics are defined, all of them imperative in nature. As a result, I had to create my own launcher, which takes a model and a set of Henshin rules as input.

## 4 LESSONS LEARNED

In this paper, I have reported on a *deep* embedding of GTS-based semantics into GEMOC Studio. An alternative would have been to create a *shallow* embedding, reusing the existing K3 engine, which expects the semantics to be expressed as operations on the meta-model. Our execution entry point could have been an operation that invokes the `performStep` operation in an infinite loop. `performStep` would look like the operation above except for the `beforeStep` and `afterStep` calls. Instead, the operation would have been annotated as a `@Step`. The obvious advantage of such a shallow embedding is that it can completely reuse the implementation of existing execution engines and launch infrastructure. However, a shallow embedding also means that the actual semantics become a second-class citizen, only indirectly loaded and referenced from a K3-based 'meta-semantics'. As a result, details of the semantics, such as rule names become more difficult to report to the wider GEMOC infrastructure, which only sees the K3 semantics. This creates problems when debugging models and recording traces. More importantly, it would create challenges when later trying to provide more dedicated support based on the specific semantics expression, such as advanced support for concurrency and analysis.

Implementing and integrating this new execution engine into GEMOC Studio has highlighted some areas of difficulty, where GEMOC Studio may benefit from some additional refactoring:

(1) *Assumption of operations-based semantics.* GEMOC has a built-in assumption that all semantics are defined through operations on the meta-model. For every step, GEMOC logs the name of a meta-class and meta-operation that was 'executed'. These information are, for example, exposed to the user in the generic debug interface. Rule-based semantics do not have meta-operations. Instead, these need to be mocked up when tracing a step. This may cause problems, in particular where some operations exist and their names (perhaps accidentally) overlap with rule names.
(2) *Inflexible launch infrastructure.* The current launcher infrastructure is tightly linked in with Melange-based language definitions. For languages that are not defined in Melange, it is necessary to implement a new launcher and launch configuration. Unfortunately, the Melange-dependent and independent launcher code is not well separated and reusable, which leads to code cloning in any new launchers.

Overall, integrating the Henshin engine has been surprisingly straightforward: the core engine consists of only 2 classes with a total of 268 LOC Xtend code. Infrastructure made up the bulk of the code: 567 LOC of Xtend and Java code over 6 classes[7] implement the launcher infrastructure; only 2 classes provide bespoke behaviour. There are some 47 LOC of configuration data. Including investigating parts of the GEMOC-Studio sources in depth to identify appropriate hooks for extension, the engine was developed in a week. While improvements are still clearly possible, I consider this a reasonable amount of effort to spend on a task that will not need to be repeated for every new language.

[7] Some copied from existing code in GEMOC Studio because of export settings

## 5 CONCLUSIONS

I have integrated a new execution engine for Henshin-based language semantics into GEMOC Studio. Overall, the claim that GEMOC Studio is easy to extend has been upheld. Developers of new extensions can choose between deep embeddings (by developing a new execution engine, as shown here) or shallow embeddings (by encoding the interpretation of semantics in a K3-based semantics). Some areas of the GEMOC-Studio framework are currently still tightly coupled to languages whose operational semantics are defined using operations on the meta-model, and where the language definition can be expressed in Melange. Refactoring these areas of GEMOC Studio would substantially increase the ease with which new execution engines can be integrated.

Language semantics expressed as a GTS are inherently non-deterministic, typically, capturing concurrent execution. Latombe et al. [8] describe support for concurrent semantics in GEMOC. In future work, I intend to build on this for a new version of the Henshin Engine. Support for concurrent execution is less well abstracted in GEMOC Studion and, therefore, more difficult to reuse and extend. Latombe [8] introduces a separate specification of the concurrency model of a language. Such a concurrency model is very useful for further analysis and efficient execution. With a GTS-based semantics it should be possible to infer the concurrency model instead. I hope to explore this in future work. Additionally, I would like to extend the current Henshin Engine to support timed rules in a similar fashion to *e-Motions* [10], which would open up opportunities for building bespoke advanced analysers for high-level, domain-specific models. Finally, I will explore supporting more complex operational semantics that use rule scheduling (called units in Henshin). This creates challenges around identifying execution steps and larger-scale rollback.

## REFERENCES


[1] E Bousse, T Degueule, et al. 2016. Execution framework of the GEMOC studio (tool demo). In *SLE'16*.
[2] A Corradini, R Heckel, and U Montanari. 2000. Graphical Operational Semantics. In *Workshop on Graph Transformation and Visual Modelling Techniques*.
[3] T Degueule, B Combemale, et al. 2015. Melange: A Meta-language for Modular and Reusable Development of DSLs. In *SLE'15*. https://doi.org/10.1145/2814251.2814252
[4] F Durán, A Moreno-Delgado, et al. 2017. Amalgamation of Domain Specific Languages with Behaviour. *JLAMP* 86 (Jan. 2017). Issue 1. https://doi.org/10.1016/j.jlamp.2015.09.005
[5] F Durán, S Zschaler, and J Troya. 2013. On the Reusable Specification of Non-functional Properties in DSLs. In *SLE'12 (LNCS)*, Vol. 7745.
[6] IBM. 2006. Ecore API Documentation. http://download.eclipse.org/modeling/emf/emf/javadoc/2.4.0/org/eclipse/emf/ecore/package-summary.html. (2006).
[7] M. E. Vara Larsen, J. DeAntoni, et al. 2015. A Behavioral Coordination Operator Language (BCOoL). In *MODELS'15*. https://doi.org/10.1109/MODELS.2015.7338249
[8] F Latombe, X Crégut, et al. 2015. Weaving concurrency in executable domain-specific modeling languages. In *SLE'15*. 125–136. https://doi.org/10.1145/2814251.2814261
[9] S. Mellor and M. Balcer. 2002. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley.
[10] J Rivera, F Durán, and A Vallecillo. 2009. A graphical approach for modeling time-dependent behavior of DSLs. In *VL/HCC'09*. https://doi.org/10.1109/VLHCC.2009.5295300
[11] D Strüber, K Born, et al. 2017. Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In *ICGT'17*.
[12] V. Viyović, M. Maksimović, and B. Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *INES'14*. https://doi.org/10.1109/INES.2014.6909375
[13] C Wende, N Thieme, and S Zschaler. 2010. A Role-Based Approach towards Modular Language Engineering. In *SLE'09 (LNCS)*, Vol. 5969. https://doi.org/10.1007/978-3-642-12107-4_19