A Plug-in Based Approach for UML Model Simulation

Alek Radjenovic, Richard F. Paige, Louis M. Rose, Jim Woodcock, and Steve King

Department of Computer Science, The University of York, United Kingdom
{alek,paige,louis,jim,king}@cs.york.ac.uk

Abstract. Model simulation is a credible approach for model validation, complementary to others such as formal verification and testing. For UML 2.x, model simulations are available for state machines and communication diagrams; alternative finer-grained simulations, e.g., as are supported for Executable UML, are not available without significant effort (e.g., via profiles or model transformations). We present a flexible, plug-in based approach to enhance UML model simulation. We show how an existing simulation tool applicable to UML behavioural models can be extended to support external action language processors. The presented approach paves the way to enrich existing UML-based simulation tools with the ability to simulate external action languages.

1 Introduction

The UML 2.x standard supports modelling of behaviour through a number of mechanisms, including state machines and activity diagrams, and pre– and post-conditions expressed in OCL. In parallel, executable dialects of UML have been developed to support more fine-grained specification of behaviour using *action languages*. As a result, these languages support rich simulation and code generation from models, but they are not directly supported by UML 2.x compliant tools, nor are they based on the same metamodels as UML 2.x. If modellers want to use action languages (and supporting simulators) with UML 2.x models and tools, they either have to acquire a tool that already provides such capabilities (which may not support their exact simulation requirements), or make use of, e.g., model transformations to a different set of languages and supporting tools.

We present a flexible, plug-in based approach for UML model simulation. We show how existing modelling and simulation tools capable of processing UML behavioural models can be extended to support external action language processors, including for their simulation. This is achieved through precisely modelling the interfaces between the modelling tool and an external action language processor, and implemented using a plug-in mechanism. The approach supports enrichment of UML modelling and simulation tools with simulation capability for action languages. The approach even enables addition of *further* action languages to UML tools that already possess one, thus allowing engineers to increase and tailor the simulation support available in the existing tools.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 328–339, 2012.

This approach has been developed to meet industry requirements, in the context of the European FP7 project INESS, to (i) provide enhanced simulation support for existing railway interlocking models, particularly for safety analysis and validation; and (ii) without requiring new modelling tools to be purchased or developed.

As a result of our work, we have modified an existing UML 2.x modelling and simulation tool in several ways. We have: (i) extended its internal behavioural metamodel to support embedded expressions specified in external languages, (ii) enabled the tool to support a plug-in mechanism, and (iii) extended the tool's functionality with a new set of services compliant with the interfaces mentioned above. Importantly, the operation of the tool in the absence of external plug-ins is identical to the normal operation of the tool prior to the modifications. We have also implemented a plug-in based on an existing action language, and validated our approach using a number of real-world case studies from the railway signalling domain, as we describe later in the paper.

As opposed to existing approaches, we tried to generalise the interactions between a UML simulation tool and an *external* language processor, allowing the capability of existing tools to be augmented without further tool development.

The remainder of the paper is structured as follows. Section 2 presents the background and context. Section 3 presents the overall requirements for tool support (both modelling and simulation), and the mechanisms used to flexibly extend existing tools to support external action languages. Section 4 presents examples of industrial application and describes how the approach exploiting the plug-in based approach was assessed. Section 5 summarises related work, and we analyse the effectiveness of the approach in the conclusions in Section 6.

2 Context and Background

This work was undertaken within the INESS (Integrated European Signalling System), funded by the FP7 programme of the European Union. This industrial project focused on producing a common, integrated, railway signalling system within Europe. Signalling systems are perhaps the most significant part of the railway infrastructure. They are essential for the performance and safety of train operations. A significant number of large UML models produced by the signalling experts using a tool called CASSANDRA [7] (a plug-in for the UML tool Artisan [1]) had been developed. The input models comprise UML class and state machine diagrams and are used to model railway signalling systems and simulate their execution. In addition, the models were enriched with expressions described in CASSANDRA's bespoke action language, SIML. A strict requirement in the project was that modelling tools (Artisan) remain unchanged, for the purposes of validation (engineers were unwilling to change the tools or the way in which they used them, partly because model modification was too expensive).

Engineers also used CASSANDRA to perform simulations, in order to check functional properties and explore safety requirement violations. The simulations

were executed via a Prolog-based engine [8], and as such the simulator was considered to be slow and inefficient by engineers. Additionally, customisation of the simulator was not possible. Requirements for more fine-grained control of simulations were expressed by the industry engineers.

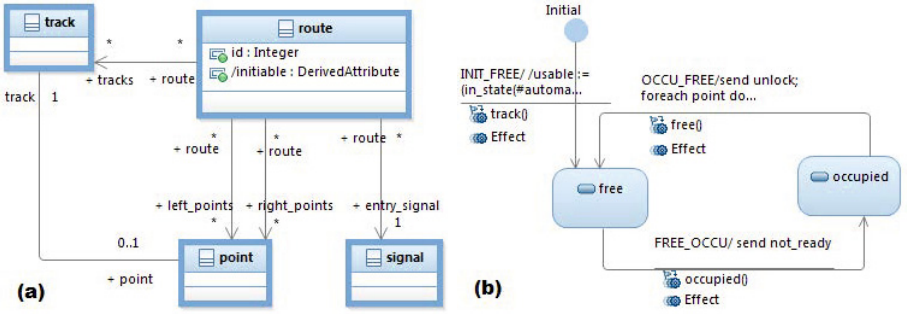

Fig. 1. (a) Class diagram; and (b) State machine for Track class

A simplified excerpt of a class diagram is shown in Fig 1 (a). The UML state machine in Fig. 1 (b) models the behaviour of Track objects. The expressions that follow the transition names, such as '*send not_ready*' are written using SIML [7]. SIML is composed of *four parts* that roughly deal with the following concerns of model simulation:

- *declaration* – allows users to define basic elements, typically corresponding to the UML model elements (classes, events, etc.) that can be read directly from the input model *only if CASSANDRA is used*; also, define elements not present in the original model (e.g. inputs from the environment)
- *expression* – assists users in building and evaluating complex expressions (classified according to the data type) formed from multiple elements
- *action* – provides mechanisms to define elementary pieces of behaviour (e.g. creation/deletion of instances and association links); invocation of behaviours defined in the source model (e.g. class operations or transition triggers)
- *control* – allows users to combine elementary actions into ordered sequences and iterations

Engineers working on INESS had created a substantial number of railway signalling models using the CASSANDRA extensions to Artisan. The simulation capabilities of CASSANDRA did not provide adequate performance or scalability to sufficiently validate the models, explore them, and provide assurance that safety properties were met. As a result, new requirements for simulation were specified. In particular, it was mandated that engineers would still be able to use CASSANDRA (and Artisan), that the existing action language would still be supported, but external simulators (that were more efficient, more scalable, or performed better) that would also simulate the action language could be exploited.

To satisfy the industrial requirements to provide richer and higher performance simulation capabilities while still retaining use of CASSANDRA/Artisan, we have developed a plug-in based approach, detailed in the next section. Our development so far has connected a specific external language processor and a simulation tool as proof-of-concept to the railway signalling engineers. The external processor is capable of parsing an action language and making requests into the simulation tool's API (e.g. to create objects, or fire events), but has awareness of the global clock, tasks, message queues, and other standard simulation tool's resources. The simulation tool is based on the SMILE platform [15,14]. The tool's architecture is shown in Fig. 2, and its operation is best described using the following scenario.

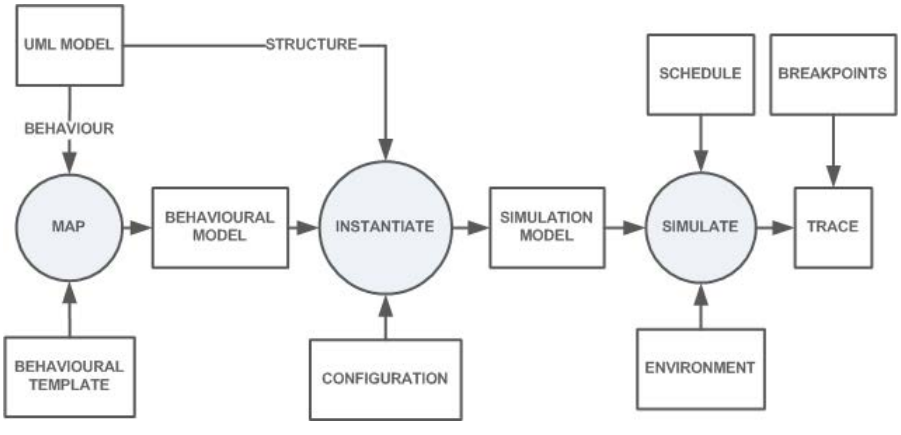

Fig. 2. Simulation tool architecture

The input UML model (created by any UML compatible tool) is first queried (as explained in [15]) in order to extract the behavioural information from state diagrams (states, transitions, triggers, etc.). This is then mapped to a state machine *behavioural template* defined in one of the SMILE family of languages to produce a set of types which describe only the behaviour of UML model components. The set of types is stored in a *behavioural model*. In parallel, the input model is also queried to generate structural information (e.g. from class diagrams). The next step involves using manual *configuration* (supported graphically by the tool) to create a *simulation model*. The simulation model is a set of instances of types from the behavioural model that also take into consideration the structural hierarchy obtained in the previous step.

The simulation tool supports concurrency in the form of tasks which provide execution separation. Consequently, in the *configuration* step, users may define multiple tasks and map each simulation object to one of them. After selecting a *scheduling mode*, the simulation can be run. Optionally, users can define the system's environment in the form of one or more stimuli. The tool produces a

simulation trace, providing a detailed report on events that occurred during the simulation (e.g. triggers, transitions, message queues, or unsatisfied conditions).

3 The Plug-in Based Approach

The existing tool could simulate basic UML behaviours (described by standard UML behaviour diagrams). More fine-grained behaviour descriptions could only be provided using external action language expressions. Thus, the key objective was to provide a flexible plug-in mechanism to allow connection and interoperation between the tool and the external language processor, and to allow us to detach, disable or replace the new simulation capability when not needed. The work was divided into the following packages:

1. Inclusion of *external* (SIML) behavioural data from the input model into the *simulation models* (Fig. 2.), ensuring additional data in the simulation model is ignored by default, avoiding disruption of the existing tool operation.
2. Enhancement of the simulation tool so that it provides a mechanism for plug-ins capable of processing expressions in an action language
3. The implementation of the plug-in for the SIML action language
4. Evaluation by performing simulations and verifying model correctness
5. Generalising and formalising the interface(s) between the tool and the language plug-ins, enabling usage with a variety of (Executable UML) notations

3.1 Extensions to the Behavioural Metamodel

Behavioural templates are used to create *behavioural types* based on information extracted from a source UML model. Fig. 3. shows an extended metamodel for the types created based on the behavioural template for state machines. We observe that the behavioural types are composed of one or more *properties* (e.g. states) and *transitions*, which in turn comprise a *trigger* and one or more *conditions* (guards) and *actions*. The existing behavioural type metamodel was extended with the `External` class to enable the inclusion of the embedded action language expressions from the source model into the simulation. This class defines three attributes: *language* – to signify which action language is used, *classifier* – to describe which particular part of the behaviour the expression applies to (e.g. *effect*, for transitions; *exit* or *entry*, for a state change), and a *body* – which contains (external notation) expressions unknown to the simulation tool.

3.2 The Simulation Tool Plug-in Mechanism

To allow for an arbitrary action language to be used on the core UML models, a plug-in mechanism has been chosen. This choice ensures that there will be no need to modify the simulation tool to accommodate different kinds of executable UML. The loading of plug-ins (one per project) is dynamic. (A project includes input models, queries, a behavioural template, configuration files, schedule, breakpoints, etc.). The mechanism's implementation details are specific to the implementation platform and are not relevant here.

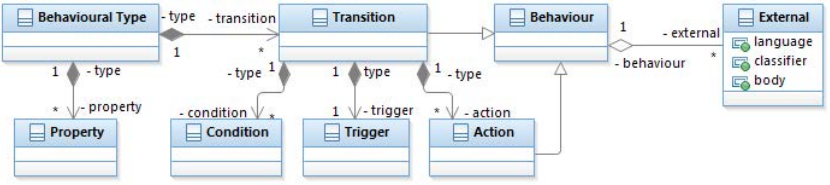

Fig. 3. Extended behavioural type metamodel

3.3 The SIML Plug-in

The SIML plug-in is the SIML-specific implementation of the external language processor interface, `IPlugin` (Fig. 4. (a)), and was developed incrementally, adding the functionality found in SIML expressions in a gradual manner.

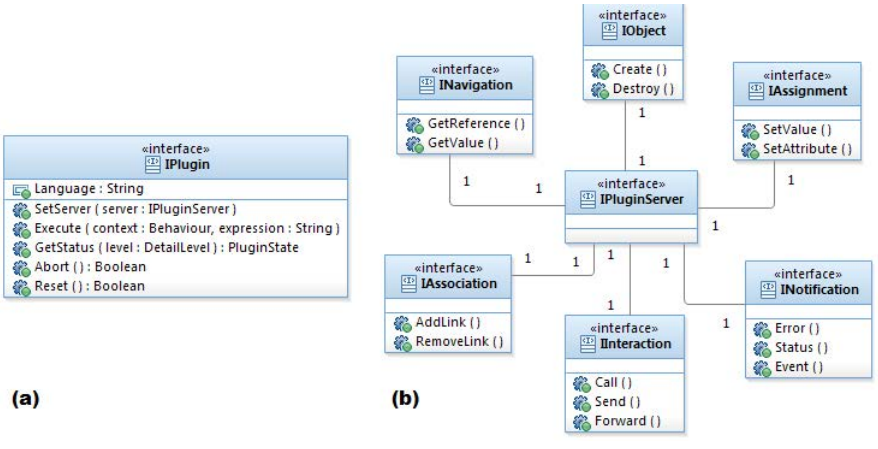

Fig. 4. (a)IPlugin and (b) IPluginServer interfaces

The `SetServer` method is called when the plug-in is loaded. The simulation tool then uses the `Language` attribute in order to determine which action language it is capable of processing. The `SetServer` method passes a reference to the simulation tool's component that is the implementation of the `IPluginServer` interface (described in the next section). When the `SetServer` call is made, the plug-in resets its internal state to the default initial state. The tool can also force this operation by using the Reset method. The `Execute` method is called when the plug-in is required to process an action language expression. This method takes two parameters: `context` – informing the plug-in which (behavioural) object is requesting the processing, and `expression` – the action language expression that needs to be processed. For debugging, management or reporting purposes, the tool can also request a snapshot of the plug-in's internal

state through the `GetStatus` method, as well as issue an `Abort` request as a safety mechanism to abandon further proecssing.

3.4 The Simulation Tool Plug-in API

`IPluginServer` essentially describes an API (a set of simulation tool's services that `IPlugin`–compatible plug-ins can use). The API is described by the `IPluginServer` interface (Fig. 4. (b)), and consists of the following (sub-) interfaces grouped by the functionality:

- **navigation** – providing access to model elements
- **object** – required to create and destroy objects
- **assignment** – required to set values to object references and to allocate values to objects' attributes
- **association** – required to create and remove relationships between objects
- **interaction** – providing mechanisms to pass data between objects in synchronous and asynchronous manner
- **notification** – asynchronous mechanism for providing management and operational notifications from the plug-in to the tool

Basic methods associated with each section are also shown in Fig. 4(b). With the exception of those in the `INotification` interface, these methods represent a minimum set of services required to 'drive' a state-machine based simulation scenario from an executable UML plug-in.

3.5 Normal Operation, Control, and Exceptions

The simulation tool was also required to implement a mechanism to control the overall simulation execution in the presence of an external plug-in (e.g. in order to avoid deadlock scenarios, either because of a faulty or partial plug-in implementation, or an incomplete/erroneous behavioural specification in the input model). The tool and the plug-in operate collaboratively during the simulation runs, using a master-slave mode of operation. The simulation tool is a passive master of the workflow, at times relinquishing control to the plug-in in full, but continually monitoring the execution and intervening in case of a problem by: disabling the plug-in, aborting the simulation, and reporting back to the user. In a nutshell, a simulation run goes through the following stages and sub-stages:

- **Power-up** – initialisation of tool and plug-in internal states and variables
- **System execution** – composed of *simulation steps*; in each step, the tool sequentially executes all system tasks; this stage is repeated until either (a) the simulation runs its natural course where all simulation objects have become idle, (b) the users halts the execution, either manually or through a breakpoint, (c) an error is reported by a plug-in, or (d) the plug-in has become unresponsive
  - **Task execution** – task's simulation objects are executed sequentially; if an object's message queue is empty or blocked, the object is skipped

    - **Object execution** – executing object's current behaviour (e.g. a transition in a state machine scenario, comprising multiple actions)
- **Power-down** – in this stage, the simulation trace may be logged, or perhaps a report generated if particular analysis is required

The above execution flow is controlled by the tool. The control is partially relinquished to the plug-in at the *object execution* level, if and when an external action language expression is read from the object's message queue. Before doing so, the tool sets a timer to guard against situations when the plug-in blocks further operation of the tool. If the control is returned to the tool before the time-out, the timer is cancelled; if not, the plug-in is forcefully disabled, the simulation is aborted, and feedback is given to the user.

4 Industrial Application and Assessment

The work presented is in response to strong industrial demands to provide enhanced simulation support for railway interlocking models, particularly for safety analysis and validation, without requiring new modelling tools to be purchased or developed. The enhanced control of simulations is achieved through external action languages that allow a more fine-grained specification of behaviours in UML models. We have demonstrated how to extend a simulation tool's capabilities to exploit external action languages, achieved through a dynamic integration of the tool and the external module using the plug-in mechanism. We have also formalised the interaction between these two by precisely modelling their interfaces.

Following the implementation of the extension to the tool and the SIML plug-in, we have used the following criteria in order to validate our approach, by verifying the tool operation:

- *without a plug-in* – in the absence of a plug-in, the tool's operation must be equivalent to that prior to the modifications made
- *with a plug-in* – in the presence of a relevant language plug-in, simulation is either fully automatic or hybrid (largely driven by the executable UML expressions embedded in the model, but manual user input is also supported)
- *with a malfunctioning plug-in* – the tool must prevent the plug-in from blocking the simulation indefinitely by disabling the plug-in under such circumstances, aborting the execution, and providing the feedback to the user

All scenarios above were tested and the criteria satisfied. In particular, the 'malfunctioning plug-in' was repeatedly tested during the incremental development; during this period, when the plug-in's functionality was only partially implemented, blocking situations were in abundance. The verification of the normal operation of the tool (without a plug-in) was compared against the unmodified version of the tool on the same set of case studies. Finally, the hybrid operation was tested by disabling selected functionality in the plug-in (e.g. sending a message) and replacing it by issuing a prompt to provide a manual user input.

The fully-automatic operation was tested on a number of case studies from the railway signalling domain, the smallest containing 7 classes and 25 SIML statements, and the largest 89 classes and several hundred SIML statements. Every class had its own state machine defined. Simulation runs were performed on all case studies, validating the approach and the tool/plug-in operation. Due to project's time constraints, it was not possible to fully verify the correctness ([2]) of the larger models. The modified tool outperformed Artisan-CASSANDRA combination, if only just – possibly attributed to the 'lightweightedness' of the tool, but because of its prototype status, further improvements are expected. The real power of the approach, however, lies in its extensibility and generalisation. The SIML plug-in was implemented in 3 weeks, but after the initial learning curve, we anticipate the future plug-in development to shrink to several days.

The following are typical scenarios in which the integrated solution (the simulation tool and the plug-in) was (can be) used:

- *normal execution* – performing a simulation run according to the specification derived from the source models. Two most typical outcomes are: (a) execution runs its natural course (i.e. no further activity detected), a snapshot of the current state of the system is taken, then analysed together with the simulation trace; (b) simulation runs indefinitely; user forcefully halts the execution and analyses trace.
- *property checking* – checking if specified property is satisfied. We use conditions (Boolean expressions composed of model elements, attributes, and Boolean operators) specified in breakpoints to signify properties. A breakpoint causes the simulation to pause (or come to a halt) when its condition evaluates to true.
- *error injection* – analysing the robustness of the modelled system, i.e. its behaviour in the presence of errors. Errors are purposely introduced into the system through user input.

Property checking and error injection are illustrated using the example in Fig. 5. Two routes R1 and R2 (comprising tracks T, signals S, and points P) are defined as:

```
R1 = {S1; T1,P1(left),T3} and R2 = {S1; T1,P1(right),T2}
```

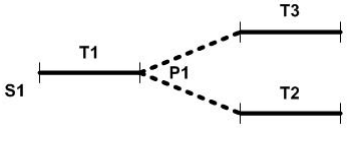

Fig. 5. An example railway interlocking model used for assessing the approach

We observe that that the routes share elements such as the entry signal S1 or track T1. When specifying properties, we use *negative application conditions* (NACs) (e.g. to verify that a system is safe, we define conditions that make

the system unsafe). For instance, we may want to verify that signal S1 is not in the `proceed` state if, at the same time, track T1 is in the `occupied` state. Thus the property that we want to check is defined using the Boolean expression:

```
(S1.state == 'proceed') AND (T1.state == 'occupied')
```

specified as a breakpoint condition. If, during a simulation run, the expression evaluates to *true*, the execution of the model is stopped. The user is then able to see which breakpoint triggered this, and can analyse the trace to find the design fault.

We can also deliberately inject errors into the system's behaviour. Using the same scenario, we can define a different breakpoint, as follows:

```
(R1.state == 'active') AND (R2.state == 'active') AND
(T1.state == 'occupied')
```

Routes essentially have two main states: *idle* and *active*. Since R1 and R2 share a common track, they cannot be both *active* at the same time when T1 is *occupied*. During the simulation, for instance, when R1 is active and T1 occupied, we can (through user input) change R2's state to 'active' and observe how the system behaves afterwards. This method works well; however, the simulation clock has to be slowed down substantially in order to inject errors at the right time. Work is in progress to provide a mechanism to use rule-based scripts for error injection.

5 Related Work

Shlaer and Mellor [18,19,17] developed a method in which objects are given precise behaviour specifications using state machine models annotated with a high-level action language. In the late 1990s, OMG started working on Action Semantics for the UML with an objective to extend the UML with a compatible mechanism for specifying action semantics in a platform-independent manner.

There are a number of research and academic platforms and tools that attempt to define behaviours in UML models more precisely, and to execute such enriched models [10,6,16]. There are also several commercial tools that define their own semantics for model execution [7,5,4,9,3]. They often include a proprietary (action) language. Consequently, models developed with different tools cannot be easily interchanged and cannot interoperate.

The Foundational UML (fUML) [12] specification (adopted in 2008) provided the first precise operational and base semantics for a subset of UML encompassing most object-oriented and activity modelling. fUML, however, does not provide a new concrete *surface* syntax; rather, it ties the precise semantics solely to the existing abstract syntax model of UML. Subsequently, OMG issued an RFP for Concrete Syntax for a UML Action Language. Currently, what is known as the Action Language for fUML (or Alf) [11] is in beta 2 phase (since 2010).

Alf is a textual surface representation for UML behaviours. Expressions in Alf can be attached to a UML model in any place where a UML behaviour

can be. Any Alf text that can be mapped to fUML can be reduced to a set of statements in first-order logic. Unfortunately, it does not allow us to use model-checking features or a theorem prover for validation or verification. In that respect, significant additional work remains in order to provide a complete formal verification for a system [13]. This is exactly why simulation in combination with Alf (or indeed any other executable UML language) is beneficial. Simulations do not provide formal proofs; nevertheless, they can significantly increase confidence in the tested models.

There is one other significant gap. Although we now have the action semantics (in fUML) and are close to seeing a fully adopted concrete syntax (in Alf), the question of *how* the models will be executed still remains. In other words, there are currently no attempts to standardise or even specify a simulation environment for UML that would define how such platform should connect to Alf or another (proprietary) action language. In the meantime, we should do as best as we can with what is available and try to adapt the existing UML compatible simulation tools for use with diverse executable UML notations. This paper contributes to this objective.

6 Conclusion

Model simulation is increasingly seen as a reliable complementary approach to formal verification or testing that attempts to establish the validity of model behaviours. Standard UML's limitation to describe these behaviours in greater detail is addressed by a number of action languages. Although the majority of these languages were designed following the same or similar philosophy, there is no common metamodel from which they can be derived. This presents a significant challenge for UML tool makers if they were to support a number of such languages in their tools.

We have tried to address this problem and have proposed a modular approach to UML model execution through simulation. We have demonstrated a scenario in which a UML-based simulation tool is extended in a non-disruptive, modular, manner to accommodate multiple action language 'engines' to collaborate with the tool during simulation runs. This was achieved by augmenting the tool's capability using a simple plug-in paradigm. We have formalised the solution by defining the interfaces required by the plug-in and the tool so that the two can be integrated. We have also illustrated the normal operation, the workflow control, as well as the exception handling mechanisms. We have evaluated our approach on examples from the railway signalling domain. The tool we used in our study was an in-house tool (based on the SMILE platform), while the language was CASSANDRA's action language SIML.

Our next steps will include investigation into more action languages, other simulation platforms, as well as further generalisation and formalisation of the approach. We anticipate an increased activity in the model simulation arena and are also mindful of the fact that new domain-specific action languages may emerge. As part of our desire for a broader research impact, our intention is to standardise the interaction between the simulation tools and action languages.

References

1. Atego. Artisan Studio (2011), http://www.atego.com/products/artisan-studio/
2. dos Santos, O.M., Woodcock, J., Paige, R.F., King, S.: The Use of Model Transformation in the INESS Project. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 147–165. Springer, Heidelberg (2010)
3. Dotan, D., Kirshin, A.: Debugging and Testing Behavioral UML Models. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA 2007), pp. 838–839 (2007)
4. IBM, Rational Rhapsody (2012), www.ibm.com/software/awdtools/rhapsody/
5. IBM, Rational Software Architect RealTime Edition (RSA–RTE) (2012), http://www.ibm.com/software/rational/products/swarchitect/
6. Jiang, K., Zhang, L., Miyake, S.: An Executable UML with OCL-based Action Semantics Language. In: 14th Asia-Pacific Software Engineering Conference (APSEC 2007), pp. 302–309 (December 2007)
7. Know Gravity. CASSANDRA (2011), http://www.knowgravity.com/eng/value/cassandra.htm
8. Mellish, C.S., Clocksin, W.F.: Programming in Prolog: Using the ISO Standard. Springer (2003)
9. Mentor Graphics. BridgePoint (2012)
10. Mooney, J., Sarjoughia, H.: A Framework for Executable UML Models. In: 2009 Spring Simulation Multiconference. Society for Computer Simulation International (2009)
11. OMG. Action Language for Foundational UML (Alf). Technical Report October 2010, OMG (2011)
12. OMG. Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0. Technical Report, OMG (February 2011)
13. Perseil, I.: ALF formal. Innovations in Systems and Software Engineering 7(4), 325–326 (2011)
14. Radjenovic, A., Paige, R.F.: Behavioural Interoperability to Support Model-Driven Systems Integration. In: 1st Workshop on Model Driven Interoperability (MDI 2010), at MODELS 2010, Oslo, Norway. ACM Press (2010)
15. Radjenovic, A., Paige, R.F.: An Approach for Model Querying-by-Example Applied to Multi-Paradigm Models. In: 5th International Workshop on Multi-Paradigm Modelling (MPM 2011), at MODELS 2011. ECEASST, vol. 42, pp. 1–12 (2011)
16. Risco-Martín, J.L., de La Cruz, J.M., Mittal, S., Zeigler, B.P.: eUDEVS: Executable UML with DEVS Theory of Modeling and Simulation. Simulation 85(11-12), 750–777 (2009)
17. Shlaer, S., Mellor, S.J.: Object-Oriented Systems Analysis: Modeling the World in Data. Prentice Hall (1988)
18. Shlaer, S., Mellor, S.J.: Recursive Design. Computer Language 7(3) (1990)
19. Shlaer, S., Mellor, S.J.: Object Lifecycles: Modeling the World in States. Prentice Hall (1992)