SyLaGen: From Academic Tool Engineering Requirements to a new Model-based Development Approach

Moritz Balz[a], Michael Striewe[a], Michael Goedicke[a]

[a] *Paluno – The Ruhr Institute for Software Technology*
University of Duisburg-Essen, Campus Essen, Germany

Abstract

In this contribution we reflect on the development of SyLaGen, an academic load generation tool for performance tests. It is able to generate a defined amount of requests to a system under test and measure the response times. The development of this tool has been influenced by two facts over the last ten years: First, its variety in functionality and the high number of platforms and frameworks in use; and second, the desire to specify the main functionality for measurements as precise as possible with respect to appropriate models. However, these requirements often contradict, since model-driven development is not easy to apply to existing architectures. In the case of our tool, this lead to a different approach for model-based development embracing formalized design patterns. We will here introduce the nature of this academic tool and the side effects of its development to other software engineering domains.

1. Introduction

Tool building in academia can be driven by different factors: functional requirements, non-functional requirements, experimental development processes, case studies, or others. If industrial partners are involved, all of these factors can be influenced not only from inside academia, but also from the outside, including change requests and time constraints. In this contribution we reflect on the development of SyLaGen, which has experienced several development steps during the past ten years, some of them with project partners from the industry. The development has been influenced by functional and non-functional requirements as well as by development styles, which makes it an interesting case study for academic tool building. One of the most important findings was the creation of a new implementation style for model-based programming that will be explained in detail in this paper.

In general terms, SyLaGen (the name is an acronym for the German tool description "Synthetischer Last-Generator") is a load generator application for performance tests. It is able to generate a defined amount of requests to a system under test and measure the response times. Several important capabilities are explained in detail in sections 2.1 and 3.1. More general information can be found in existing publications, one referring to an initial version [3] and one to a re-designed version [11].

Email addresses: moritz.balz@uni-due.de (Moritz Balz), michael.striewe@s3.uni-due.de (Michael Striewe), michael.goedicke@s3.uni-due.de (Michael Goedicke)

This contribution is organized as follows: Section 2 reports on the initial development of SyLaGen to give a feeling for the initial requirements and the nature of the load generation tool. Section 3 elaborates on the re-development that took place some years later as a pure academic project. Section 4 provides the main contribution of this paper and focuses on the most important findings on development and implementation style during the re-development phase. Section 5 concludes the paper.

2. Initial development of SyLaGen

The initial development of SyLaGen was started as a cooperation between the University of Duisburg-Essen and the Siemens company in the late 1990s. Thus, the first version of SyLaGen had to meet both academic and industrial requirements. In addition, the development process had to obey limited project resources and some engineering and development standards used by Siemens.

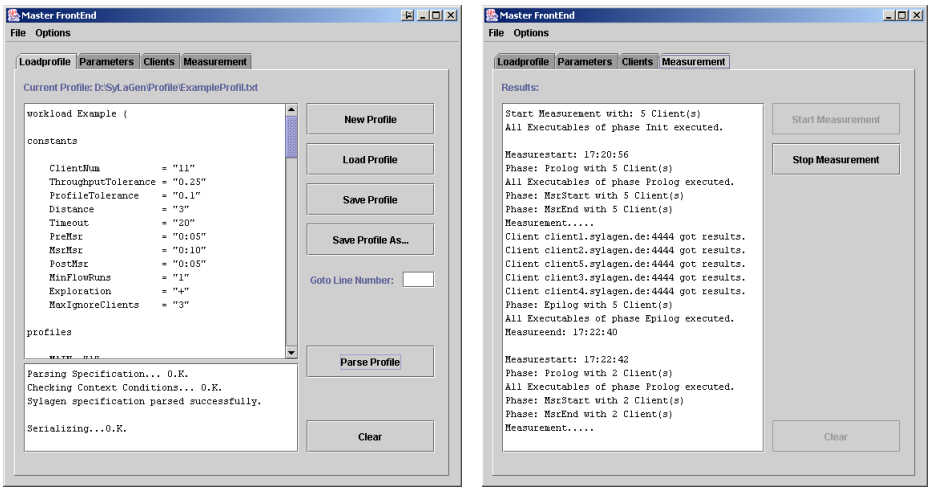
2.1. Requirements of SyLaGen

From its beginning, SyLaGen was intended to be a framework rather than a monolithic tool and thus focused on extendability with respect to four different aspects: First, a system under test may offer different interfaces for handling external requests, thus a load generator must be able to handle different protocols randomly and in parallel. This was the major key argument for developing a new load generator at all, because Siemens wanted to be able to test their proprietary protocols, which was not possible with available tools. Second, load generation for a client-server system may require complex client behaviour that cannot be formulated in a simple descriptive way, but instead with non-trivial algorithms that have to be implemented programmatically. Third, more than simple atomic measurements may be required in complex environments, so that strategies applying sequences of measurements to a system should be configured. In particular, these strategies may incorporate preparations for each single measurement, e.g. resetting databases, or adjust load generation parameters between two atomic measurements. Finally, complex load generation may result in complex use cases that cannot be modelled as linear scripts, but as probabilistic networks.

Some additional properties regarding the general system design were required in parallel to the flexibility requirements named above: To ease generation of high load volumes, the system architecture must allow distributed load generation from different clients. For this purpose a "master" component controlling the measurement should connect to client instances that generate the actual load and may be either physical computers or logical instances sharing a host system.

2.2. Design Implementation of the Initial Version

The implementation of the first version of SyLaGen was straightforward: The master component was a monolithic desktop application based on Java's Swing user interface. The workload was specified textually as shown in figure 1(a). A parser for the workload input format was generated automatically with JavaCC [5] from a grammar description. Besides several parameters like measurement time, timeouts and so on, a workload was composed of so-called "flows". Each flow can be understood as a simple state machine where each state contains load generating requests to the system under test. Transitions are labeled with a weight, allowing for probabilistic paths through these machines. This provides the required flexibility for complex load generation which is far more powerful than linear scripts.



(a) Textual workload definition in the first version of SyLaGen.

(b) Textual measurement output in the first version of SyLaGen.

Figure 1: The user interface of the first SyLaGen version.

Similar to the textual input, during measurement the output was also displayed as a log file on the screen as shown in figure 1(b). The results of the complete measurement were stored in plain text files since sophisticated reporting was outside the scope of the requirements.

The client components were written in C to allow for efficient execution even on limited devices and were connected to the master with a simple socket-based network protocol. In order to achieve the desired flexibility for protocols used in requests to systems under test, so-called "adapters" were used. Each adapter was implemented in a single DLL library and provided a set of methods to access a certain communication interface or protocol of the system under test. Since these adapters were written in C like the clients, they could implement non-trivial algorithms as desired. Each workload definition referenced the necessary adapters so that they were distributed to the clients by the master before starting the measurement.

While the resulting architecture was thus comparatively simple, one part of the application was given special care: The measurement strategies as the core of the measurement process were designed as state machines. Note that the state machines for load definition described above are of descriptive character and subject to be created and changed by the every-day users of SyLaGen, while the state machines for strategies we are now talking about are intended to be models for the implementation and thus created and edited by the developers. However, no modeling tools were used to derive the implementation from models systematically – at the time of the initial development, the related technologies were far less advanced than today. However, it was not desirable to implement the measurement process as large parts of sequential program code and loose all semantic information by this means. For this reason, a design pattern for state machines was employed that stored the basic information about states and transitions inside the program code. It followed some simple rules:

- A class `Priostate` was instantiated to represent single states. Each state instance was given a name to make logging messages comprehensible later on.

```
Priostate verify = new Priostate("Verify");

// ...

Transition foundMarkGoToVerify = new Transition("FoundMarkGoToVerify", verify) {
  boolean checkCondition() {
    return theState.loadTooHighOrNoMoreMeasurementsPossible();
  }

  void doAction() {
    phase="Verify";
    theState.saveResults();
    theState.decreaseToFirstUnexplored();
    theState.doMeasure();
  }
};
```

Listing 1: The informal program code pattern used in the first version of SyLaGen with a state and a transition.

- An interface with `get` methods provided access to the business logic of SyLaGen and allowed to extract variables related to the measurement, for example the number of restarts after errors.
- A class `Transition` was instantiated for each transition and connected to state instances. Each instance implemented two methods: `checkCondition` evaluated expressions related to method calls on the variables interface; `doAction` executed business logic by accessing other parts of the program code.

An exemplary part of the program code pattern can be seen in listing 1. In the upper part, a state instance is created. In the lower part, a transition instance is created. In order to supply it with guard and action program code, it is created as an anonymous class implementing both required methods.

The state machine was first populated by creating instances for states and transitions. Afterwards an execution method was called that repeated the following steps beginning with the start state: (1) Invoking and evaluating all `checkCondition` methods of the transitions emanating from the current state. (2) Selecting a transition whose guard returned `true` and invoking its `doAction` method. (3) Setting the target state of the chosen transition as the new current state.

By this means the state machine functionality was made explicitly and comprehensible for developers in the program code. However, no connection to any formal model was maintained, neither implicitly nor explicitly.

3. Review and Extensions

After the initial development, SyLaGen had been in use for several years both for industrial purposes and in academic courses. Besides minor maintenance changes, there was no critical review of the source code and no plans for further major development steps. However, even maintenance of the clients became difficult since the C-related tool chain was outdated and no experienced C developers were available to create a new one. To cope with this problem, the client was re-implemented in Java in 2006. At this time, the authors discovered that documentation for the initial development was partly incomplete or out-of-sync with the actual program code, which can be considered a quite common problem for long running projects. In addition,

the look and feel of the master's user interface had also become outdated and old fashioned, as well as the style of communication and data exchange between master and clients became laborious and frail in comparison to more up-to-date technologies. These were major issues since industry projects were still accomplished, although the project partners did not participate in the development, but only ordered measurements without knowing the underlying system.

Thus it was decided to start a complete re-engineering project for SyLaGen in 2007. In contrast to the initial development, the decision for re-engineering SyLaGen was a pure academic project without industrial partnership. Moreover, it did not happen inside a limited time frame or with dedicated resources. One the one hand, this allowed for more freedom and experiments during development, but on the other hand the available resources where generally low and could neither be increased by referring to upcoming project deadlines nor by additional funding. As a consequence, the first step towards a new version of SyLaGen was a student project that was concerned with design recovery and code review for the existing version of SyLaGen. This critical review resulted in two interesting facts: First, measurement strategies were implemented as explicit state machines as described in the previous section, but they were not as well decoupled from algorithmic details as it was required for the intended flexibility. Second, the behaviour of several other parts of the system could also easily be described as state machines, suggesting at least a far more modular architecture than the one that was implemented.

3.1. New Requirements and Re-engineering Goals

Not only the code review and design recovery activities but also experiences from years of using SyLaGen set up the goals for a second version of SyLaGen: First, the architecture of the master should be modularized, implementing aspects like measurement control, master-client-communication and user interface in independent software components. Particular focus was put on decoupling the state machine descriptions for measurement strategies from execution details realized by the measurement control component. This requirement was not only driven by the academic goal of a fully modularized architecture, but also by the need for implementing new measurement strategies. A more modular architecture was also required in order to add sophisticated reporting capabilities which were not included in the initial version. Second, the old user interface and communication protocols should be replaced by new versions. The new user interface should be based on the Rich Client Platform of the Eclipse development environment [13] since it provides many features for building editors and is at the same time integrated in program code development tools, which is of interest for adapters. The communication protocol should make use of XML to wrap complex data structures, so that it could be handled with standard protocols instead of proprietary protocols and still stay flexible for extensions in the future. Replacing the old proprietary protocols in turn includes rewriting the communication interface of the clients as well. Third, the new capabilities gained from using Eclipse RCP should be used for a new input editor for formulating workloads in a more convenient format. However, this last point was more a nice side effect than a crucial argument in favour of re-development.

3.2. Implementation of the New Version

The considerations for a redesign lead to a module structure that is shown in figure 2: The overall measurement is controlled by a user interface module and an attached configuration store. The main task of the user interface is the creation of the workloads for the performance tests, which are validated by a related module. When the measurement is started, the workload information is passed to the measurement module. It uses the client module which is connected to

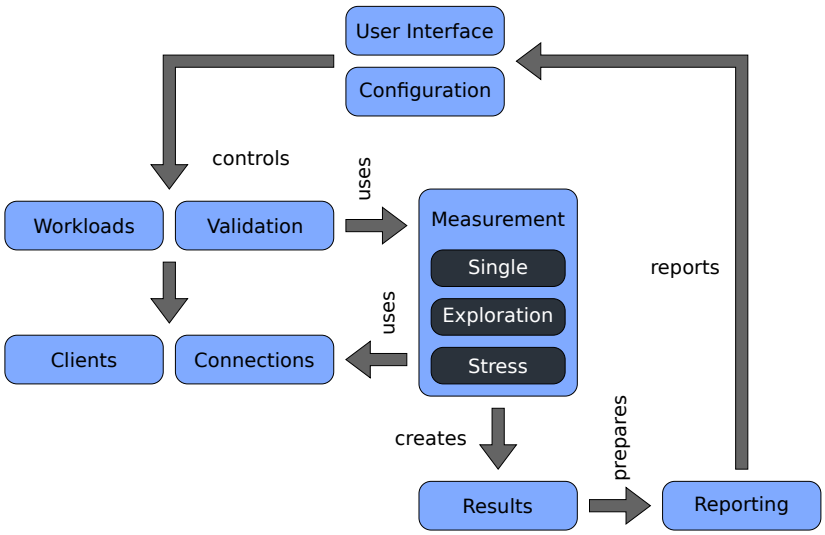


Figure 2: The module structure in SyLaGen with the basic kinds of relations between modules during the measurement process. The load generation strategies in the "measurement" module are embedded in and tightly connected to the different modules.

actual clients by means of a connection module that servers as an abstraction layer over different possible communication protocols. In the measurement module, the different load generation strategies are realized as plug-ins. They share the business logic provided by SyLaGen, but differ with respect to the number of measurements and the number of clients to be used. When a measurement is finished, the data collected by the clients is submitted to the result module. It is passed over to a reporting module that can employ different reporting technologies, e.g. writing the raw numbers to a file or alternatively passing them to a sophisticated reporting system. The prepared results are then provided to the user by the user interface again.

In order to facilitate a clean and formal approach to definition and implementation of the measurement strategies, a systematic re-engineering was applied to the existing source code. This re-engineering was supported by the fact that the state machines contained information about states, transitions, guards, and variables. The result of this was a set of state machines models defined in the timed automata model checker UPPAAL [8], which was chosen for its good visualization, simulation, and verification capabilities.

The other parts of the application were not developed based on models. The reason for this is the high number and tight connection of dependencies regarding platforms and frameworks:

- The user interface module completely depends on the structures determined by the Eclipse RCP platform. The resulting user interface can be seen in figure 3.
- The workload module depends on the XML schema of the desired workload format and frameworks to map them to in-memory object structures.
- The validation module depends on frameworks that can inspect compiled Java libraries in order to validate adapters.

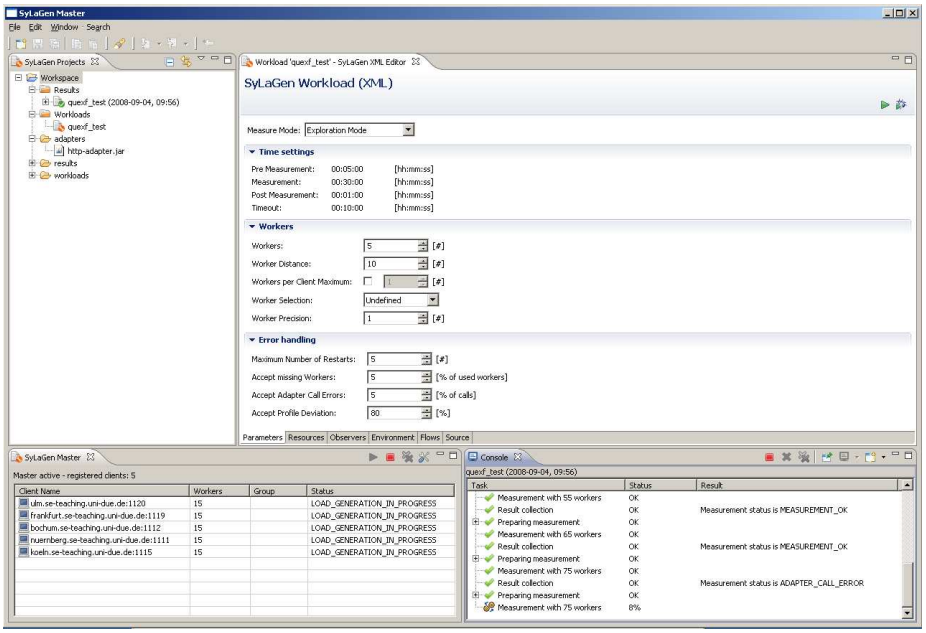


Figure 3: The user interface of the second version of SyLaGen based on Eclipse Rich Client Platform. We can see the project explorer on the left hand, a workload being edited in the middle, and the status display of clients and a running measurement at the bottom.

- The connection module currently uses a lean socket-based protocol to allow for connection to clients running on different platforms. For this purpose, all data is adjusted to text and number formats defined for this protocol.
- The measurement module provides calculations regarding the data used during measurement, for example mean throughput and request times.
- The result module embraces statistical calculations to merge results collected from single clients into a consistent overall result.
- The reporting module interacts with existing libraries for creating simple files, spreadsheet files, or input for reporting systems.

While for some of these tasks certainly model-driven approaches [4] can be applied, the number and variety of requirements precludes a consistent model-driven approach. A derivation of the implementation from models would either have been limited to few parts of the application or would have introduced a notable effort for customization of existing model-driven development tools. A generation of program code for the strategies would have been possible, but was not desirable since it would have been an isolated solution [7, 15].

However, an ad-hoc implementation was also not feasible. Since SyLaGen is an academic tool used in different projects and research contexts, requirements change often and quickly. Thus it was soon clear that it was not desirable to manually derive the implementation from these models again, since this would lead to the facts that the models (1) would just be a documentation and (2) would soon be inconsistent if no notable constant effort would be put into synchronizing

models and implementation. It was thus clear that possible approaches currently available did not fulfill the objectives to develop the software in a consistent manner while at the same time automating the synchronization with the models.

Following this conclusion and reflecting on the pattern-based solution from the initial version, we developed a new approach for implementing functionality based on models that can be integrated with arbitrary program code, as we will explain in more detail in section 4. At this time of the development it turned out as very beneficial to run the re-design project without industry partners and tight time frames, because this situation allowed us to delay active development of SyLaGen for experiments on the approach. As a result of these experiments, the second version of SyLaGen is as modularized as desired, uses all frameworks and platforms as necessary, and contains measurement strategies that are based on and formally connected to formal models.

4. Formalized Design Patterns

The need for a model-based development technology that integrates in a heterogeneous application like SyLaGen lead to the development of alternatives. It is likely that our research in this area would have been of different character if it was not motivated by the need to development SyLaGen with its different – and partly contradicting – requirements. Moreover, we are convinced that we would have needed more time to achieve practical results without this use-case-driven way of research.

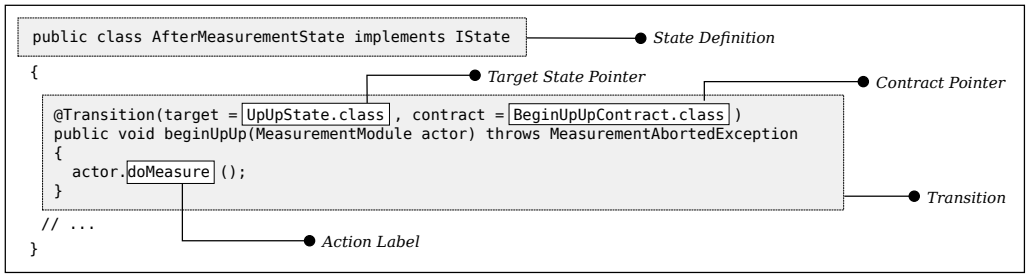
4.1. Approach

The informal design pattern used in the first version of SyLaGen as shown and explained in section 2.2 was considered to be optimally integrated in the overall application. However, it lacked the possibility for a synchronization with formal models. The main reason for this is that it would not be possible to interpret the program code with respect to abstract models in an automated way: The fact that instances are created for single model elements makes it necessary to interpret possibly arbitrary algorithms to determine the model specification from object instantiations.

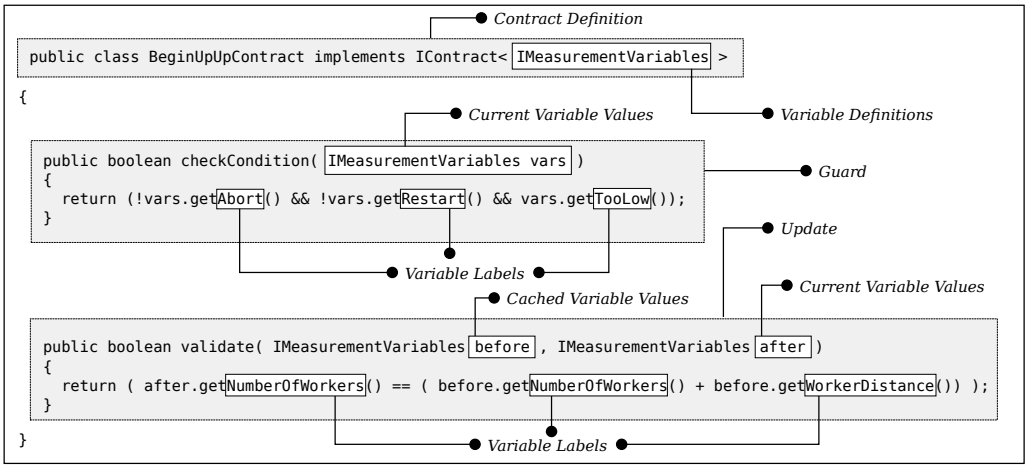
We therefore decided to modify the existing approach mainly with respect to the storage of information. For this purpose we use *attribute-enabled programming* [9] that uses meta data facilities introduced in programming languages like Java [12] to store meta data inside the program code and interpret the static structures by this means. This results in program code patterns that are formalized so that the static structures can represent the abstract syntax of models. Thus transformations can extract complete models from the code accordingly. The approach has so far been published as *embedded models* [2].

In SyLaGen, an embedded model for state machines based on UPPAAL has been created. It stores the complete state machine syntax in static elements of the program code and re-uses for this purposes some principles of the informal design pattern of the the first version, but also adds some additions:

- An interface provides `get` methods that abstract from the overall business logic of SyLaGen and provide a limited set of well-defined variables to be used in the state machine.
- Another interface (called *actor*) provides entry points to the business logic that can be called from inside the state machine. The names of the methods are by this means the action labels for the state machine.



State and Transition Definition in Source Code



Contract Definition in Source Code

Figure 4: A state definition with an outgoing transition and its contract. The first method of the contract checks a pre-condition with the current variable values, while the second method checks a post-condition by comparing the current values to previous values.

- States are represented as class definitions. The class name is interpreted as the name of the related state.
- Transitions are represented as methods inside state classes. They contain a set of method calls to the actor interface, so that each transition has a set of associated action labels.
- Meta data at transition methods refers to the target state class definition and to methods containing expressions for guards and updates. These expressions evaluate to boolean values and access the variables in the related interface.

An example of the formal program code pattern can be seen in figure 4. It shows a part of the SyLaGen state `AfterMeasurementState` in the "exploration" strategy that decides how to proceed after a single measurement was performed. As we can see, the single elements of the program code can be interpreted unambiguously with respect to the state machine model. Since the classes are not instantiated to be equipped with semantics, the program code can be statically analysed at development time and run time. Some more details have already been described in our previous publications [2, 1].

The execution of this pattern follows a similar principle as in the first version, but also with some important changes. These changes are based on the fact that our design pattern introduces a new abstraction layer and thus an additional layer of indirection and modularity. Consequently, execution is not based on object instances, but on the static structures instead, which are accessed by means of structural reflection [6]. Thus the state machine is not populated by creating instances of state and transitions inside the application source code itself, but by reading the program code by an execution framework in the measurement module. The execution framework for the embedded model interprets the program code as follows beginning with the start state's class definition and performing the following steps: (1) Instantiating the class, invoking and evaluating all guard methods of the transition methods in the current state. (2) Selecting a transition whose guard returned `true` and invoking its transition method. Besides the class instantiation at beginning of step 1, these steps are similar to what the old version did. (3) Invoking the update expression to determine if the business logic has performed changes that are compatible with the state machine specification. This step of permanent self-verification could now be introduced since the necessary information is now available from the embedded formal model. (4) Setting the target state of the chosen transition as the new current state until a final state is reached. This step again is similar to the last step in the old version. However, it has to be noted that all steps that are similar to the old version are now performed by the execution framework and not by the source code defining the state machine. Thus the goal of decoupling strategy definition and strategy execution can be considered fulfilled in the new version.

4.2. Usage for Development

This complete information about the model allows not only to specify the state machine semantics more precisely than before and execute them in an automated way. Moreover, we can extract the model from within the program code and view or manipulate it in appropriate modeling tools, in this case UPPAAL as shown in figure 5 [10]. Since the program code pattern is a valid notation for the complete model specifications, we can also transform a changed model back to program code and thus apply changes.

At run time it is possible to monitor the execution of the program code pattern with respect to the state machines since the static structures of the program code are interpreted by the execution framework. A tool exists that shows the related model elements and their behavior, including a visual view of the state machine of the measurement strategy currently in use. By this means it is possible to track errors in the context of the abstract specifications without relying on meta data or tracing information, but only on the embedded model that is contained in the program code without any additional effort.

Thus the approach fulfills the needs of the academic tool development in this case:

- The number and variety of frameworks and platforms in use precludes solutions that require a complete formal definition at higher levels of abstraction or determine the software architecture by themselves. Instead, program code based on models is required to seamlessly integrate in the existing structures. The formalized design patterns allow for this since they share the notation of the program code that is also used by the frameworks and platforms.

- The high frequency of change requests makes a separate maintenance and synchronization of program code and complex and time-consuming. The formalized program code patterns works with different abstraction levels, but uses only one notation to store them, so that changes always affect all abstraction levels.

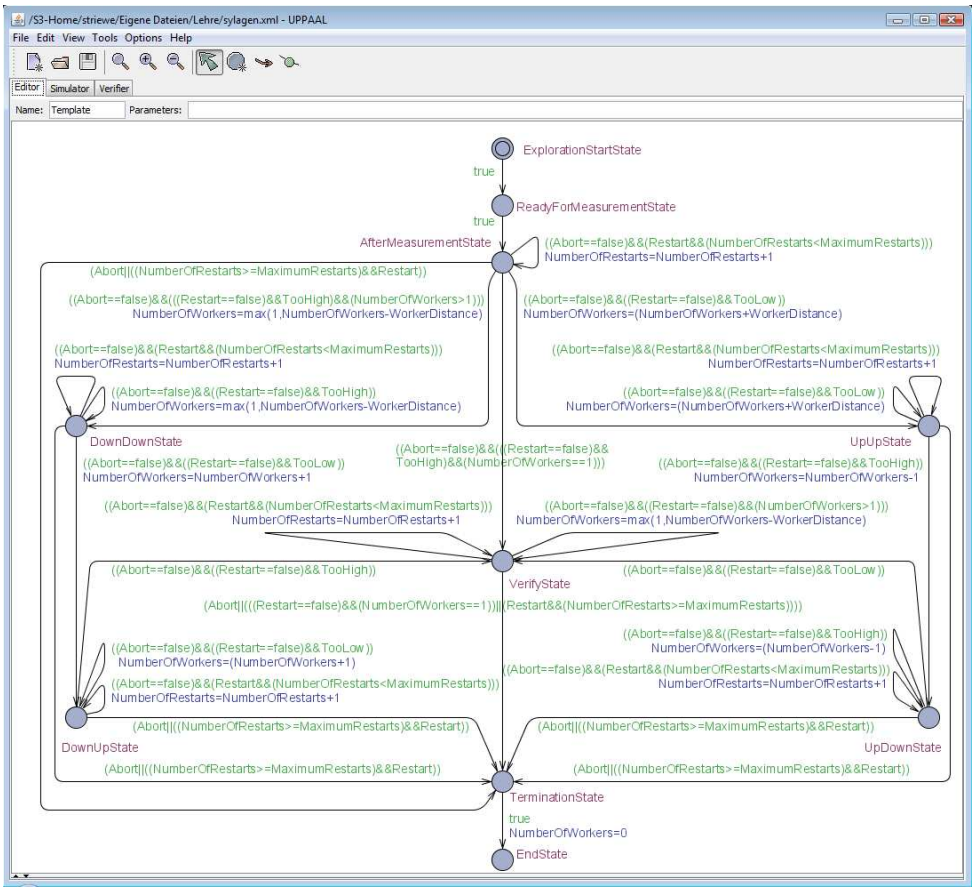


Figure 5: The UPPAAL model of the state machine controlling the “exploration” load generation strategy.

- Resources for industry-quality documentation are often not available in an academic context, so that there is a constant risk that the model documentation may not fully represent the reality. In the shared notation of the program code, documentation is always up-to-date since the model specifications can be extracted as long as the code complies to the formal specifications for the program code pattern. This can easily be validated or enforced with static code analysis.

In summary, the development and maintenance of SyLaGen was very interesting from our point of view: The reflection on the technologies in use and the requirements that occur for academic tools lead to a different model-based development approach, which has proved to work in this context since 2008.

5. Conclusion and Future Work

In this contribution we presented the development of the load generator platform SyLaGen. Considering the development of academic tools, it is from our point of view interesting with

respect to two aspects. First, we considered special (technical as well as organizational) requirements that lead to the need of non-standard solutions. In this case, certain quality criteria were desired, but not easy to meet in the dynamic context of academic projects. Second, the academic context lead to a solution that is from our point of view generally applicable to problems of the same class. This kind of research that happens beside actual projects, in similar contexts referred to as *serendipity* [14], is in our opinion a valuable contribution of the development of "real" applications in academia.

Future work regarding SyLaGen will include more reflections on model-based development. On the one hand, it is desirable to cover more parts of the application with formal models. This affects foremost the overall process of the measurement and also the modules and their interactions. Since it will not be feasible to generate the complete code, we want to create appropriate embedded models for these purposes. On the other hand, this leads to the questions how different domain-specific models can interoperate: By using the source code as the only notation, we will have a situation where different domain-specific models are tightly connected. A formal specification of their interoperability is desirable. SyLaGen as a tool is large enough to serve as a realistic and demanding case study for both kinds of research in model-based development.

References

[1] Moritz Balz, Michael Striewe, and Michael Goedicke. Embedding State Machine Models in Object-Oriented Source Code. In *Proceedings of the 3rd Workshop on Models@run.time at MODELS 2008*, pages 6–15, 2008.

[2] Moritz Balz, Michael Striewe, and Michael Goedicke. Continuous Maintenance of Multiple Abstraction Levels in Program Code. In *Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development - FTMDD 2010, Funchal, Portugal*, 2010.

[3] Reinhard Bordewisch, Bärbel Schwärmer, Michael Goedicke, and Peter Tröpfner. Lastsimulation für anwendungsumgebungen in vernetzten it-architekturen. *Mitteilungen der GI-Fachgruppe MMB*, (43), 2003.

[4] Alan W. Brown, Sridhar Iyengar, and Simon Johnston. A Rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, 2006.

[5] Tom Copeland. *Generating Parsers with JavaCC*. Centennial Books, 2nd edition, 2007.

[6] François-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995.

[7] Brent Hailpern and Peri Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[8] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct 1997.

[9] Don Schwarz. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. *ONJava.com*, June 2004. `http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html`.

[10] Michael Striewe, Moritz Balz, and Michael Goedicke. Enabling Graph Transformations on Program Code. In *Proceedings of the 4th International Workshop on Graph Based Tools, Enschede, The Netherlands, 2010*, 2010. accepted for publication.

[11] Michael Striewe, Moritz Balz, and Michael Goedicke. SyLaGen - An Extendable Tool Environment for Generating Load. In Bruno Müller-Clostermann, Klaus Echtle, and Erwin Rathgeb, editors, *Proceedings of "Measurement, Modelling and Evaluation of Computing Systems" and "Dependability and Fault Tolerance" 2010, March 15 - 17, Essen, Germany*, volume 5987 of *LNCS*, pages 307–310. Springer, 2010.

[12] Sun Microsystems, Inc. JSR 175: A Metadata Facility for the Java™ Programming Language, 2004. `http://jcp.org/en/jsr/detail?id=175`.

[13] The Eclipse Foundation. Eclipse website. `http://www.eclipse.org/`.

[14] Pek van Andel. Serendipity: Expect also the Unexpected. *Creativity and Innovation Management*, 1(1):20–32, 1992.

[15] John Vlissides. Generation Gap. *C++ Report*, 8(10):12, 14–18, 1996.