SimuLizar NG: An extensible event-oriented simulation engine for self-adaptive software architectures

Sebastian Dieter Krach and Max Scheerer
{krach, scheerer}@fzi.de
FZI Research Center for Information Technology

Abstract

Software simulation constitutes an essential mechanism for design time architecture analysis. Domain-specific software, e.g. of cyber-physical systems, requires domain-specific extensions to the architecture models and their execution semantics. Existing simulators are cumbersome to extend, do not support self-adaptivity or do not scale well. In this paper we present concepts of SimuLizar NG, a scalable simulation engine for the SimuLizar approach. Its principal goal is to facilitate domain-specific extension and adaptations to the model interpretation semantic while at the same time ensuring reactive simulation execution in demanding scalability scenarios.

1 Introduction

Modern cyber-physical systems as e.g. self-driving vehicles rely on increasing amounts of software to perform critical tasks of the system. Hence, the requirements towards availability and resilience of the software components increase as well. The *Palladio* approach [4] allows for factoring predictions of runtime metrics into the early software design. One way of generating predictions is by simulating the system based on a model of the software, the hardware and the expected usage. SimuLizar [3] constitutes a simulation-based analysis within the Palladio approach. It supports analyzing the effect of self-adaptation mechanisms by allowing changes to the architecture model during a running simulation.

Employing discrete event simulation (DES), SimuLizar runs a monolithic model interpreter in a separate thread for each simulated user in the system. Limited through the single threaded nature of DES only one interpreter can run at a time. Consequently, the design is prone to scalability issues when simulating large numbers of concurrent users. In particular, when evaluating self-adaptive policies, the simulator needs to be able to cope with overload scenarios. As Merkle et al. [2] showed, the scalability improves when using event orientation. The cost of which is an increased complexity of the model interpreter.

The model interpreter of SimuLizar uses switch-case-like constructs to process the PCM model elements based on the element type. Adding support for new types or instance properties, e.g. *stereotypes*, requires intrusive extensions to the interpreter.

Therefore, in this paper we present the design of SimuLizar NG (*Next Generation*), a new simulation engine for the SimuLizar approach. SimuLizar NG aims to replace the current model interpreter and is designed to address the major drawbacks of the current implementation: 1) facilitate simulations of a large number of parallel activities while remaining responsive and 2) providing means of adapting the simulation logic in an non-intrusive manner.

2 Foundations and Related Work

Performance simulation in Palladio relies on queuing network semantics [4]. Based on a workload specification describing the manner of user arrival to the system, the simulator instantiates simulated users. These users interact with software component instances (*Assemblies*) through provided interfaces. Components specify behavior in relation to service calls of required components and internal resource demand. Thereby, simulated users issue demands to capacity-limited resources and continue once processing finishes.

Discrete event simulation represents a system as set of system state variables which only change at discrete points in simulated time (*events*) [1]. Two dominant world views in DES are *event orientation* and *process interaction*. While process interaction describes dynamic behavior as sequence of time-ordered activities and delays, event orientation models it through event routines which execute in zero simulated time [1].

The existing simulators for the PCM *SimuCom* [4] and *SimuLizar* [3] adhere to a process interaction view. User behavior is simulated as time-ordered sequence of resource requests and delays encapsulated into distinct threads. When issuing resource demand, the user thread is suspended until processing has finished. EventSim [2] adheres to a event oriented world view. Dynamic behavior is realized as zero simulated time consuming event routines. EventSim requires the developer to schedule behavior using closures and, in that, sacrifices the intuitive sequential specification of activities. SimuLizar NG conceptually adheres to a process interaction view while leveraging the technical advantages of event oriented processing.

3 Concept

SimuLizar NG aims at increasing simulation scalability by replacing the one-to-one mapping between simulated users and native operating system threads with an event oriented single-threaded realization. In order to support runtime changes to the architecture model SimuLizar NG continues to use a visitor-based approach navigating through the model.

Leveraging the scalability advantages of event oriented processing requires the model interpreter of a single user not to block on encountering simulated time consuming activities. Instead, it needs to temporarily return and allow for processing of other users. Furthermore, to increase extensibility we split the monolithic interpreter into multiple interpreters and a selection mechanism. In accordance with the *single responsibility principle*, each interpreter targets onyl one type of model element for which it realizes only one of the following effects: 1) simulate time consuming side effect, e.g. CPU demand processing, 2) change the current execution context, e.g. *External Call Actions* might change the *Resource Container* on which execution continues, or 3) determine model elements which are processed next, e.g. in *ResourceDemandingBehaviours* or *Loops*. Consequently, adding support for new model elements types only requires providing the appropriate interpreters.

SimuLizar NG conceptually emulates a process oriented world view through stateless interpreters for which simulation state is captured by a *Concurrent Interpretation Context (CIC)*. The CIC constitutes the equivalent of a simulation process and allows to identify the activity source (e.g. a user). Similar to a call stack, the CIC itself consists of stack frames called *Interpretation Step Contexts (ISC)*. *Interpretation step* refers to the execution of a set of interpreters for a particular (set of) model element(s) in the context of a user. Hence, every ISC is characterized by I) the model elements which are actively processed during this step, II) an ordered set of interpreters suitable for processing I), and III) the context determined previously executing interpretation steps. Figure 1 depicts the relationship between CIC, ISCs and the PCM architecture model.

Since an interpreter only simulates a single effect w.r.t. a model element, multiple interpreters can be executed during an interpretation step. For *InternalAction* elements, an interpreter simulates resource demand, while another one simulates software failures. During every step the interpreters are executed sequentially ordered by their priority. Higher priority interpreters can prevent an execution of lower priority ones in order to override default behavior. The interpretation step is finished as soon as all interpreters finished or an interpreter decided to abort the interpretation step (e.g. in case of a simulated failure).

Once an interpreter identifies a model element to be interpreted next in the same execution context it requests the addition of an ISC to the current CIC. This behavior is similar to adding a new stack frame to a stack upon calling a function.

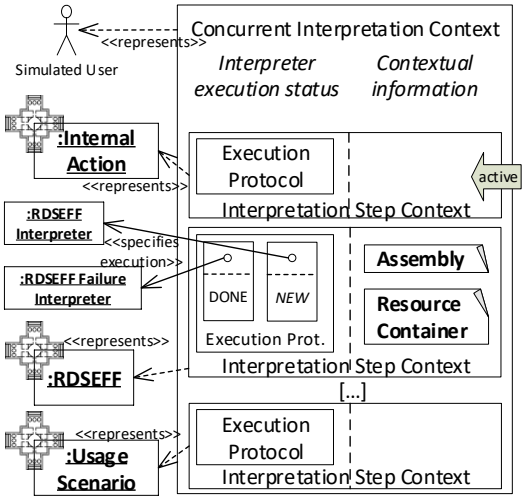

Figure 1: ISC stack of an exemplary CIC

3.1 Interpreter design

SimuLizar NG's simulation of software behavior is realized by modular *Interpreters*. Once a model element is selected for interpretation (e.g. an *InternalAction* in the context of a *ResourceDemandingBehavior*) an *ExecutionProtocol* is calculated and stored in the newly constructed ISC. The protocol specifies the execution order of the interpreters which were determined suitable for the selected model element(s). Since interpreters are stateless and cannot suspend execution during delays the execution protocol serves as storage for interpreter state. The execution protocol is stored within the ISC as the captured interpreter state is only valid in the context of a particular user.

Invocation of an interpreter results in one of the states *Done*, *In Progress* or *New*. *Done* signifies that the interpreter has finished. *In Progress* states that the interpretation of the current model element takes further simulated time. Hence, the CIC will be rescheduled once these activities have finished. *New* states that the interpreter requested interpretation of a new set of model elements. In this case, a new ISC is constructed within the current CIC. Interpretation of the current ISC will continue once the execution protocol of the newly constructed ISC has finished.

The execution protocol of an ISC is finished once all invoked interpreters result in *Done* or, as outlined before, when an interpreter request an early stop to the execution protocol, as long as all interpreters with a higher priority resulted in *Done*. As soon as an execution protocol finishes, the ISC is marked *stale* and interpretation of the previous ISC continues. Stale ISCs are temporarily kept to allow interpreters to process the result of nested interpretation. They are removed on creation of new ISCs.

Example A *RDBehaviorInterpreter* processes *ResourceDemandingBehaviors*, one action after another, starting with the behavior's *StartAction*. For each action the interpreter requests interpretation in the current context. After the respective interpreters for an interpretation request resulted in *Done* the *RDBehaviorInterpreter* gets called and continues requesting interpretation of the next action. Since the interpreter is stateless, it determines the next action based on the stale ISC of the previous action for which it requested interpretation. The modular interpreter design lets us decouple the simulation of software failures. The *InternalActionFailureInterpreter* adds a specific marker to an ISC in case a failure occurs. On the *ResourceDemandingBehavior*-level a separate *RDBFailureInterpreter* checks the stale ISCs for these markers. When it encounters one, it aborts the execution protocol, otherwise the lower-prioritized *RDBehaviorInterpreter* and, hence, the default case is run.

3.2 Interpreter selection

When an interpreter initiates the construction of a new ISC suitable interpreters are determined. Their priority order is calculated based on the model elements to interpret, the active CIC and the simulation configuration. In compatibility to existing interpretation semantics, the order can be determined by the type of the model element to interpret. *Interpreters* are registered upon simulation configuration, assigning their respective priority to the supported types. Consequently, for each type used in the architecture model, the execution order can be calculated ahead of simulation time. Additional extensibility mechanisms are provided to identify suitable interpreters based on instance properties, i. e. applied stereotypes.

3.3 Interpreter scheduling

Every CIC represents a thread of sequential software activities. SimuLizar NG manages a central queue of CICs which are scheduled for the current point in simulated time to run. The queue is processed sequentially. For each CIC, the execution protocol of its most recent ISC is processed. If interpreter execution lead to either the construction of a new ISC or marking one as stale the CIC is appended to the queue again. As soon as the queue is processed the underlying DES framework advances to the next point in simulated time for which events have been registered.

3.4 Contextuality

The interpretation of model elements is dependent of the context of the interpretation, i.e. the history of previously visited elements. For example, an Internal Action issuing CPU resource demand requires information about the component instance (*Assembly Context*) which is currently executing and its allocation to a resource container. Both cannot be determined based on the model element itself, as Internal Action are type-level specifications. Instead the information has to be inferred from interpretation history.

CICs serve a double purpose. First, they represent sequential threads of actions. Additionally, the CIC constitutes a dictionary of interpretation context-specific information. Interpreters can request the information during execution. Each information aspect is identified by a *Context Information Type*. The types are declared by concrete interpreter implementations.

For example, upon interpretation of an *InternalAction* the *ResourceDemandInterpreter* requests the *Resource Container* to which the currently executing assembly is deployed. For each type the first ISC which specifies an explicit information fragment serves as information provider. The lookup is conducted from the most recent ISC, proceeding down in the ISC-stack of the current CIC. Therefore, ISCs can override existing information or, by specify an empty mapping, remove certain information from the current context.

The available contextual information is determined by the executed Interpreters, which rely on the context to store simulation state. The *LoopInterpreter*, e. g., stores its loop counter in the context. Based on its value, the interpreter either requests processing the nested behavior (again) or results in *Done*.

4 Conclusion and Future work

In this paper we presented the conceptual design of SimuLizar NG, a new model interpretation engine for the SimuLizar approach [3]. Decoupling model traversal and side-effect simulation for each kind of model element into distinct entities facilitates customization to domain-specific use cases. Our conceptually process-interaction world view using event oriented mechanisms allows us to keep interpreters simple while relieving us of the overhead of separate threads per user.

We are currently in the process of implementing the concepts described in this paper. We plan to evaluate the prototype against existing solutions, particularly w.r.t. simulation duration and memory consumption.

References

[1] J. S. Carson. "Modeling and Simulation Worldviews". In: *Proceedings of 1993 Winter Simulation Conference (WSC '93)*. Dec. 1993.

[2] P. Merkle and J. Henss. "EventSim – An Event-driven Palladio Software Architecture Simulator". In: *Palladio Days 2011 Proceedings*. KIT, Fakultät für Informatik, 2011, pp. 15–22.

[3] M. Becker, S. Becker, and J. Meyer. "SimuLizar: Design-Time Modelling and Performance Analysis of Self-Adaptive Systems". In: *Proceedings of SE 2013*. Lecture Notes in Informatics (LNI). Gesellschaft für Informatik e.V. (GI), 2013.

[4] R. H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, Oct. 2016. 408 pp.