Model-level Execution

High-level Architecture and Task Breakdown

István Gansperger[1], Máté Karácsony[2], Róbert Kitlei[1], Gábor Ferenc Kovács[1], Boldizsár Németh[1], Tamás Kozsik[1], Sándor Sike[1], Ádám Tarcsi[1], Gergely Dévai[2]

[1]ELTE-Soft Nonprofit Ltd.

[2]Ericsson

December, 2014

1 Contents

1 Introduction

The purpose of this document is to propose architecture and task breakdown for the model executor of the future evolved xtUML modeling language. The two main use cases to consider are:

- interactive debugging sessions
- mass test case execution

1.1 Short summary of the proposal

- Translate the model to Java, where the generated code can connect to the Eclipse Debugging Framework and produces replayable logs.
- Translation must be incremental for the interactive use case.
- Actions are executed in a run-to-completion manner.
- Execution is the sequential emulation of concurrent behavior, with randomization of action scheduling and event dispatching.

1.2 Risk analysis

- Semantics of the future language regarding atomicity of actions has key impact on both model compiler and executor design. Current language discussion converges to sequentialized scheduling inside components for the first iteration (~2 years) of the project, which provides atomicity. If the tools are created with this assumption in mind, then major changes in design or nontrivial extra components will be required if intra-component parallelism with non-atomic actions would be introduced. On the other hand, creating a design for non-atomic actions from the beginning of the project would lead to more complex and expensive tools, slower prototyping and considerably less runtime performance. If intra-component parallelism with non-atomic actions were never introduced, these costs would be superfluous.
- End-to-end transformation of a model to code in an IncQuery-based toolchain is still a research topic, partly to be addressed by a demonstrator by the model compiler team this year. The feasibility of the incrementality aspect of our architecture proposal (opening up the possibility to use IncQuery as well as larger-step incrementality) is yet to be confirmed.

2 Requirement analysis

2.1 Annotated high-level Ericsson requirements

- Model interpretation: This requirement is to be generalized to "model-level execution". The technical way it is realized can potentially be code generation, just-in-time-compilation and pure interpretation as well.
- Interactivity
- Mass execution
- Scalability, performance: In the mass test execution use case, this means that the overall execution time of a test case set, including possible startup time should be minimal. In the interactive use case this requirement means that the startup time should be minimal, the execution should not cause inacceptable delays in the interactive environment and any incremental solution should avoid any user disruption during model edit.
- Visual feedback
- Test coverage statistics
- Deterministic & nondeterministic mode: Test execution should randomize in order to be able to test all execution traces and find as many errors on model level as possible. The deterministic mode requirement should be generalized: users are able to replay logs of earlier executions and have control over nondeterministic choices in new interactive runs.
- Timers
- Connection to native code
- Tracing: As a minimum, traces should contain all information necessary to replay an execution. Additional tracing content is to be clarified later. The user should have control over what is included in the traces.

2.2 Further requirements considered

- Can replay logs
- Supports randomization of schedule
- Does not prevent fine grained scheduling and analysis possibly added later
- If still-in-research technology is used, must have alternative
- Do not use unreliable/unmaintained libraries and technologies
- Easy installation, no complicated dependencies

3 High-level design decisions

3.1 Run-to-completion actions vs. instruction level scheduling

Whenever the execution of a run-to-completion action starts, it is executed until completion without interruption. On the other hand, instruction-level scheduling allows actions to be interrupted any time in order to execute instructions of other actions. The run-to-completion solution is simpler, faster, but it rules out execution paths that are possible in physically parallel execution environment. The decision about this question is of topmost importance, because it has direct impact on the runtime performance

of model execution that estimated to overwhelm the impact of other decisions, like interpretation vs. code generation, sequential vs. parallel execution etc.

The semantics of the evolved xtUML modeling language is not specified yet. At the current state of the discussion, it is foreseen that shared data parallelism will be ruled out at least in the first iteration of the supporting toolchain. (This means that components, communicating via asynchronous messages will be compiled to parallel code, while execution of intra-component model elements with shared data will be scheduled sequentially.) With this assumption, executing run-to-completion actions do not rule out any possible target behavior on the intra-component level. On the inter-component level, however, certain message sequences are ruled out. For example, if component A sends messages a1 and a2 to component C, while component B sends messages b1 and b2 to C then parallel execution can lead to message arrival sequence a1 b1 a2 b2 at C, but run-to-completion execution of the actions of A and B actions in any order cannot produce this sequence. Randomization of event dispatching can solve this issue. For example, even if the messages arrive in the order a1 a2 b1 b2, randomized event dispatching can process them in a1 b1 a2 b2 order. It is important to note that randomized selection has to be constrained: It has to respect the language semantics about ordering of messages.

In later phases of the project, shared data parallelism might be allowed. If atomicity of actions will still be kept, then run-to-completion actions will still be a correct solution for model-level execution. (In this case, the model compiler will need intensive static analysis to find out which actions can be run in parallel.) On the other hand, if actions will not (necessarily) be atomic then run-to-completion actions are too restricted to correctly emulate target execution. For example, two actions, running in parallel, reading and writing the same shared piece of data multiple times, can lead to a value of the shared data that can never be achieved by running the two actions one after the other, sequentially, in any order. Possible solutions:

- Use instruction level scheduling to be able to test all possible execution traces. However, if even individual instructions are not atomic, then possible data hazards are not correctly emulated, even with this solution.
- Log the sets of actions that were ready to run at a given point of execution and use (offline) static analysis to find out if any two of these are in read-write or write-write conflict regarding shared data resources. Issue warnings in case of conflicts found. Note that this can lead to false positives. Also note that sending messages and starting the behavior of new object instances can change the set of ready-to-run actions. This can be taken into account by breaking the actions at each such instruction and use run-to-completion only for the pieces.

In summary, shared data parallelism will be a major issue both for the model execution and model compiler teams.

Proposal: For the first iteration of the project, we propose run-to-completion actions scheduled in a randomized way and randomized event dispatching.

3.2 Parallel vs. sequential execution

Parallel model execution uses multiple threads to boost speed. Sequential execution emulates the concurrency of the model by using a scheduler to alternate between active objects having events to dispatch and actions to execute.

Parallel model execution can be realized in the following ways:

- Allocate a thread for each active object instance. This solution does not scale, because of the limits on the number of threads on different platforms. See section 1.4 of the case study document.
- Thread pool with configured number of "worker threads" that grab work packages (one or more actions to be executed by an object instance) and complete them. This solution scales, however, only insignificant speedup was measured on real multicore hardware compared to the sequentially scheduled execution. The reason for the relatively low speedup is small amount of action code and synchronization requirements. However, even if significant speedup could be achieved by making model execution parallel, the considerations discussed in the next two paragraphs suggest a sequential solution instead.

Regarding the complexity of implementation, parallel component execution is relatively simple, because the inter-component communication is by asynchronous messages. On the other hand, intra-component parallelization needs heavy locking and/or nontrivial static analysis in case of atomic actions.

Even if a single model is executed sequentially, mass test case execution can be made faster by executing test cases in parallel. Since these are independent, more speedup can be achieved than introducing parallelism inside model execution. This strategy is only applicable for mass test case execution, and not for interactive model execution and debug. On the other hand, the delays introduced by the user interacting with a graphical user interface will dominate the execution time of actions in typical models. For this reason we consider sequential execution a good-enough solution for the interactive case.

Proposal: Execution will be sequential with a scheduler that emulates the concurrent behavior of models. Running multiple test cases in parallel will speed up mass test case execution.

3.3 Interpreter vs. Code generator vs. Just-in time compilation

A model interpreter maintains an internal state of the executed model (object instances, links, non-dispatched messages, current states of the object instances etc.), decides on the next action (eg. dispatching a message, executing an instruction, doing a state transition etc.) to take and queries the model to be able to find out the effect of the action to change the internal state accordingly.

Advantages of model interpretation:

- self-contained (no tools needed to compile and load generated code)
- can start immediately, better first response time to the user
- can react to changes of the model
- easy to integrate with the eclipse debugging framework

Model execution via code generation uses model compilation techniques to translate the model into an implementation language, then compiles and executes the generated code.

Advantages of code generation:

- faster execution
- more synergies with model compiler work

Just-in-time compilation starts as an interpreter, but translates processing-intensive or frequently executed tasks on-the-fly to an implementation language, compiles and loads and uses the generated code to execute these tasks.

Advantages of JIT:

- speeds up interpretation (but not as fast as full code generation)
- keeps initial response-time low (but can be longer than simple interpretation)

The chart in section 1.8 of the case study documentation shows that interpretation is 30 times slower than generated code. Therefore interpretation is not acceptable in the mass testing use case. JIT is better, but still 6-13 times slower (depending on the machine used for the measurements). However, let us consider the advantages of interpretation and JIT to see whether those are really valid arguments against code generation:

- Self-containedness: If code generation uses Java, then the necessary tooling is present in an Eclipse environment. If C++ were used, it would increase the dependencies.
- Startup time: Section 1.9 of the case study document reports code generation times for large models that are not acceptable as startup times for the model execution process. Therefore code generation can only be used in an interactive setup if translation is incremental, i.e. it happens while the user edits the model. Incremental transformation makes the solution more complex, possibly increases the tool dependencies (see section 1.9), but also keeps startup time minimal.
- Reaction to model changes: If Java code is generated then JRE's hot code swapping support can be used to propagate the changes. Another possibility is to use the log replay functionality: The user executes the original model in an interactive session and interrupts it, saves the log of the session, then changes the model. The log is replayed with the modified model. It is possible, that the change invalidated the log, in which case it is only partially executed and the user gets the control at the state right before the first invalidated log entry. Otherwise the log is fully

executed and the user can continue the execution of the altered model from the same state as the previous session was interrupted in.
- Integration with the Eclipse Debugging Framework: Section 1.13 of the case study document reports that no technical difficulties were found in this respect.

Proposal: Use code generation.

3.4 C++ vs. Java

Using C++ would have the following advantages:

- C++ is faster than Java, but not on all machines used for measurements (1-2 speedup was measured, see section 1.8 of the case study document)
- More alignment with model compiler work made possible (However, see section 5 for limitations.)

Java would have the following advantages:

- Easier integration with the Eclipse Debugging Framework
- Less tooling dependencies
- Possibility to reuse the Java Debugging Framework for suspension on breakpoints (see section 4.2)
- Hot code swapping possibility
- The implementation would be a good basis for future Java model compiler (see section 5 for limitations)

Proposal: Code generation in Java.

3.5 Support for incrementality

Section 3.3 pointed out that incremental code generation is necessary in the interactive use case.

A possibility is to use EMF-IncQuery that provides fine-grained incrementality and a declarative model query language. While preparing the IncQuery-related case studies (see section 1.9 in the case study document) and during discussions with the IncQuery team we have identified the following issues to solve:

- There are issues when using EMF-IncQuery with EMF-UML2. Profile handling is one issue. IncQuery needs UML2-specific development to solve the problem.

- Certain recursive model structures and specific queries on these can lead to unmanageable slowdown/memory consumption in case of large models. The probability of running into this problem is said to be low, but existing.
- Papyrus needs to be extended to provide sufficient input about model changes to IncQuery.
- Specifics of the UML metamodel cause unexpected results for IncQuery queries. For example, asking for all the classes in a model returns many elements from the UML meta-model, as well as state machines of the model. (The reason for the former is the technicalities how UML2 references meta-model elements, while the state machine issue is present because a state machine is a class according to the UML2 meta-model.) When querying the model using the EMF-UML2 API, these issues are not present, because the model is discovered throught its hierarchy: components of the top-level packages, classes inside these components, state machine of a class etc. This kind of querying is possible but said to be discuraged in IncQuery.
- The application of IncQuery in an end-to-end chain of M2M and M2T transformations is still a research topic. This is a risk for product-grade implementation.
- It is unclear if the fine-grained incremental support provided by IncQuery will be beneficial considering that any small change will trigger at least one Java compilation unit to be recompiled.

An alternative solution would be to use larger step incrementality on class or component level. This solution recompiles those classes/components that were changed. Recompilation of the "dirty" units can be triggered by change events during model editing, save actions or by the start of an interactive model execution session.

In the mass test case execution scenario, incrementality is out of scope. A clean recompilation of the full model is proposed in that case.

Proposal: We envision an architecture where business logic of the transformation from model to Java is accessible via well-defined interfaces and can be called both from an IncQuery-based toolchain, from large step inceremental solutions and can be used for compilation of a full model with no incrementality. The feasibility of such architecture is still to be confirmed by prototype implementation scheduled for early 2015.

3.6 Single solution vs. separate solutions for the two use cases

The 'interactive debugging' and the 'mass test case execution' use cases have different requirements regarding execution performance and interaction with the executor's environment. This suggests two separate solutions.

On the other hand, these have to share common way of logging (tracing) and common model execution semantics. Creating a single, integrated solution is also cheaper than two separate ones.

The reasoning in the previous sections show that a solution aligned with the above stated decisions can address the requirements of both main use case, therefore we suggest one integrated solution for model-level execution.

Proposal: Create an integrated model executor serving both interactive model execution and mass test case execution.

4 Low level design possibilities

4.1 Reusing the Moka framework

When using the Eclipse Debugging Framework, three main components are interworking:

- The user interface, where breakpoints are managed, values of variables in scope are shown and, in the graphical modeling case, active model elements are highlighted.
- Execution engine
- Interconnection layer, that is responsible to transport messages between the user interface and the execution engine asynchronously.

Moka has given a graphical modeling specific implementation of the interconnection layer and parts of the user interface. If no technical difficulties will be found, then our work will provide a new Moka execution engine.

4.2 Reusing the Java Platform Debugger Architecture

Integrating a new language with the Eclipse Debugging Framework requires a mechanism in the execution engine to be able to stop execution on breakpoints. The Java Platform Debugger Architecture provides this mechanism for Java execution. If the execution engine is Java-based, it is also possible to define a mapping of the surface language elements and their Java implementation. The framework provides support for stopping the execution on breakpoints without explicit support for it in the Java implementation.

4.3 Using JDT's compiler for fine-grained incrementality

The Java compilation toolchain working in Eclipse is not relying on classical, file-based Java compilation. Instead, the Java compiler in JDT can input a source model for compilation. This enables the fine-grained incremental Java translation that is working in Eclipse. Our code generator could also feed the compiler in JDT with the source model besides generating text files with the Java source. This is an opportunity to make fine-grained incrementality relevant in a process ending with code generation.

5 Model-level execution with code generation vs. model compilation

The toolchain of a code generation based model-level execution is similar in many technical aspects to a model compiler. On the other hand, their different purposes make them different:

- Model compilers cannot be used directly for model-level execution purposes.
- Code-generation based model executors cannot be used as model compilers, but they can serve as a basis for a new model compiler.

The differences in detail:

- Model-level execution has to follow as many legal execution paths during intense testing as possible to reveal possible errors. Model compilers has to generate code that takes any legal execution path and is as performant as possible.
- Model-level execution is platform-independent. Model compilation targets a given platform and takes into account its specifics.
- Model-level execution has to connect to the debug framework. Model compilation has no such obligations.
- Model-level execution has to be prepared to execute logs. Model compilers have to support tracing, without replaying traces.
- Model-level execution can be single threaded emulation of a concurrent model. Model compilers have to emit parallel code, if that is required by the target platform.
- Model-level execution has to provide quick feedback about the correctness of the business logic captured in the model. Result of the model compiler is deployed on the target platform: The deployment can be time consuming, and the results can include platform-specific errors.

6 Task breakdown

6.1 2015 Q1

- Proof-of-concept
    - simple state machines + minimalistic Alf support

    - with basic visual debugging possibilities
    - basic support for log replaying
- Goals:
    - end-to-end toolchain validation with limited language
    - validate high-level architecture
    - validate/review incrementality plans
    - decide on low-level architecture choices

6.2 2015 Q2

- Intra component structure prototype
    - Support for associations, operations, attributes.
    - More feature-rich action language support

6.3 2015 Q3

- Refactoring and stabilization, documentation
- Publication for first application modelling pilot in September 2015
- Basic support for model - native code interconnection

6.4 2015 Q4

- Support for inter-component features
    - Dynamic component creation
    - Ports, interfaces
- Basic support for timers

6.5 2016 Q1

- Full support for model - native code interconnection
- Full support for timers

6.6 2016 Q2

- Refactoring and stabilization, documentation
- Publication for use in projects