Model-level execution
Case studies for architecture design

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

In order to make well-founded design decisions for the implementation of a new model-level execution solution, a number of case studies were implemented and evaluated. This document summarizes these case studies and the findings they led to.

2 Case Studies

1.1 Interpretation over EMF UML2

- **Directory name:** interpreter-noDebug-javaUML2

- **Short description:** Searches for signal events and a state machine region inside the input EMF UML2 model. Generates given number of random events to drive the state machine region. Logs the processed events and visited states in memory and prints report afterwards. Measures execution time containing only the event processing and in-memory logging part.

- **Technologies used:** Implemented in Java. Reads the model by the EMF UML2 API.
- **Results:** No technical problems found. Measurement results on Ericsson managed workspace, HP EliteBook8540w, Intel Core i5 CPU @ 2.53 GHz, 4 GB RAM, 64 bit Windows 7:

Execution time in ms.	10 events	100 events	1000000 events
2 states + 2 transitions	46	44	1717
10 states + 18 transitions	52	53	1934

See the results at the “Interpretation with transformation to own representation” for comparison.

1.2 Interpretation with transformation to own representation

- **Directory name:** interpreter-noDebug-javaUML2vsOwnRepresentation

- **Short description:** Searches for signal events and a state machine region inside the input EMF UML2 model. Transforms the state machine into a state/event matrix optimized for interpretation. Generates given number of random events to drive the state machine region. Logs the processed events and visited states in memory and prints report afterwards. Measures execution time separately for the transformation and the event processing plus in-memory logging parts.

- **Technologies used:** Implemented in Java. Reads the model by the EMF UML2 API. Uses a 2-dimensional int array to store the state/event matrix.

- **Results:** No technical problems found. Measurement results on Ericsson managed workspace, HP EliteBook8540w, Intel Core i5 CPU @ 2.53 GHz, 4 GB RAM, 64 bit Windows 7:

Execution time in ms.	Transformation	10 events	100 events	1000000 events
2 states + 2 transitions	41	0	1	167
10 states + 18 transitions	74	0	1	167

Comparing these with the results in section “Interpretation over EMF UML2” shows that around 10 times speedup can be achieved when using a model representation optimized for interpretation. For hundreds of events there is no significant difference between the costs of interpretation over EMF UML2 and transformation + optimized interpretation. When increasing the number of events, the optimized solution gets considerably faster.

1.3 Interpretation over a custom representation fetched from EMF UML2, with basic ALF action code support and different levels of Just in Time compilation

- **Directory names:**
  - interpreter-noDebug-javaUML2-alf-ast
  - interpreter-noDebug-javaUML2-alf-ast-jit
  - interpreter-noDebug-javaUML2-alf-ast-jit-bytecode

- **Reuse:** These case studies are reusing code from interpreter-noDebug-javaUML2vsOwnRepresentation

- **Short description:** The interpretation of state machine is semantically equivalent with the case study in “Interpretation with transformation to own representation”. It executes actions on state entry, exit and transition effects. Only the following basic ALF language elements are supported:
  - Primitive types: Boolean, Integer, Unlimited Natural, Real and String
  - Local variable declarations, referencing attributes of the context object (“this”)
  - Arithmetic operators: unary minus, +, -, *, /, %
  - Boolean operators: unary negation, &&, ||, ^
  - Relational operators: ==, !=, <, <=, >, >=
  - Blocks, assignments and if-elseif-else statements

  The first version applies pure interpretation. The rest two works as the following. When an action needs to be executed at the first time, a new compilation work is added to an executor backed by a thread-pool. Until the compilation completes, an interpreter will execute the action code. After the compilation is done, the next invocation of the given action will be executed as compiled code. The second solution compiles ALF by its AST, and then it generates Java code from actions which are compiled with javac, and loaded back into the same process. The last case study works the same way, except it uses the Javassist library to compile Java code in-memory, directly to JVM bytecode.

- **Technologies used:** Implemented in Java. Reads the model by the EMF UML2 API. Generates Java code using Java Emitter Templates. Reads and interprets ALF by its Xtext-generated AST. Compilation is done with javac and the Javassist library.

- **Results:**
  The reused Xtext grammar is left-factored, and very complicated in some cases. It was very hard and inefficient to implement even a basic interpreter on this representation. Direct compilation to JVM bytecode is more efficient than generating Java source and compiling it with javac, however this is a lot harder and more error-prone task as there is no real Java compiler in the process. For measurement results, see section 1.8.

1.4 Interpreting UML diagrams with object instances running in parallel, sequentially and in a thread pool

- **Directory name:** interpreter-threading

- **Short description:** Interprets a basic diagram that simulates a client-server connection. N clients send M messages to the server and the server responds to the messages. Relevant experiments are: running each object instance in its own thread, running each transition sequentially and running them in a thread pool. The obvious drawback of the parallel execution is that the overhead of the synchronization between the state machines will be much higher than the benefit. We expect the thread pool to perform the best when we choose the pool size to be close to the number of physical processors.

- **Technologies used:** The interpreter is written in Java. For the parallel execution we just use simple threads, the sequential execution is straightforward, and for the thread pool solution we chose the *ThreadPool* class introduced in java 7.

- **Results:** The tests were run on a 24 core (12 physical) Intel CPU @ 2666.760 MHz and 2 GB of RAM on a Debian system. As we expected the thread pool performed the best out of the three but the performance gains are insignificant over the sequential experiment. When we have many instances (100+) the one-thread-per-object approach performs really slowly.

Technical details (technologies used, task performed)	Nr. of instances	Nr. of transitions per instance	Pool size	Execution time (ms)
sequential execution of state machines	700	1000	-	3237
sequential execution of state machines	10000	1000	-	51481
multi threaded execution (one thread per object)	700	1000	-	17680
multi threaded execution (one thread per object)	10000	1000	-	310606
multi threaded execution with ForkJoinPool	700	1000	10	3273

multi threaded execution with ForkJoinPool	700	1000	24	4113
multi threaded execution with ForkJoinPool	10000	1000	10	45799
multi threaded execution with ForkJoinPool	10000	1000	24	51497

1.5 Compilation of UML2 model to Java without action code and debugging support

- **Directory name:** compiler-noDebug-javaUML2
- **Reuse:** This case study is reusing code from interpreter-noDebug-javaUML2
- **Short description:** The behavior of execution is almost the same as in case of "Interpretation over EMF UML2", except that it operates with triggers directly instead of signal events. Initially, the first state machine region found in the model is compiled into a single Java class, and then loaded into the same executor process. States are compiled into Java enumeration values, and the transition table is implemented with polymorphism and switch instructions. No action code processed or executed for state entries, exits or transition effects at all.
- **Technologies used:** Implemented in Java. Reads the model by the EMF UML2 API. Generates Java code using Java Emitter Templates. Compiles the code using javac.
- **Results:**
  No technical problems found. See measurement results at the end of the next section.

1.6 Compilation of UML2 model to C without action code and debugging support

- **Directory names:**
  - compiler-noDebug-javaUML2-jni
  - compiler-noDebug-javaUML2-stream
  - compiler-noDebug-javaUML2-socket
  - compiler-noDebug-javaUML2-file
- **Reuse:** These case studies are reusing code from compiler-noDebug-javaUML2

- **Short description:** Every case study compiles the first state machine region found in the given model into C code. They have the same execution semantics as described in the previous case study. In the first three cases the triggers to be processed by the state machine are generated by an executor process, which is implemented in Java. These are different only in the communication scheme. The first uses Java Native Interface to implement transition logic in the state machine, so the C code is compiled into a dynamic library, which is loaded by the executor process into the JVM. The second one uses the standard input and output streams to communicate with the Java executor from a separate process, and the third behaves the same, just uses TCP sockets instead of the standard streams. Finally, the communication is unidirectional in the fourth solution, as it only sends a trace in a regular file to the host process, but it generates the triggers for itself.

- **Technologies used:** Implemented in Java. Reads the model by the EMF UML2 API. Generates C code using Java Emitter Templates. Compiles the code using the GNU Compiler Collection. The different versions are using regular files, TCP sockets and JNI.

- **Results:** The last case study implements only a unidirectional channel, because bidirectional inter-process communication through a regular file would require additional synchronization, e.g. locking. However, it could be developed into a shared memory solution using memory mapped files. As using a TCP connection is very reliable, easy to use and relatively fast, it is recommended to connect an external process to a debugger. It is the most common solution used in many existing products. The case studies were measured on a simple, two-state machine processing one million random triggers. Measurement results on Ericsson managed workspace, HP EliteBook8540w, Intel Core i5 CPU @ 2.53 GHz, 4 GB RAM, 64 bit Windows 7:

Case study	Avg. compilation	Avg. execution	Total
compiler-noDebug-javaUML2	1295 ms	136 ms	1431 ms
compiler-noDebug-javaUML2-jni	2605 ms	135 ms	2740 ms
compiler-noDebug-javaUML2-stream	895 ms	3031 ms	3926 ms
compiler-noDebug-javaUML2-socket	1789 ms	45817 ms	47606 ms
compiler-noDebug-javaUML2-file	1717 ms	3027 ms	4744 ms

1.7 Compilation of UML2 model to Java with Eclipse Debugging Framework support, without action code

- **Directory name:** compiler-edf-javaUML2

- **Reuse:** This case study is reusing code from compiler-noDebug-javaUML2

- **Short description:** The base execution semantics is the same as in the previous group of case studies. It is able to step the state machine using the Eclipse Debugging Framework with the standard debugging tools on user interface. When execution is suspended, the current state is shown in the Variables view of Eclipse. It also provides a custom launch configuration to select which state machine should be executed.

- **Technologies used:** Implemented in Java. Reads the model by the EMF UML2 API. Generates Java code using Java Emitter Templates. Compiles the code using javac, and loads it into the same process. Eclipse Debugging Framework is used to implement stepping through the execution. The launch configuration window contains a little amount of UI code copied almost unchanged from Moka.

- **Results:** Integration with Eclipse required a lot of additional code compared to the initial code base size, but most of this code is general and reused from a tutorial. Writing the domain-specific parts of the debugger code was simple after the EDF cooperating layer was established. The measurement process was the same as in the previous section. Results on Ericsson managed workspace, HP EliteBook8540w, Intel Core i5 CPU @ 2.53 GHz, 4 GB RAM, 64 bit Windows 7:

Case study	Avg. compilation	Avg. execution	Total
compiler-edf-javaUML2	169 ms	449 ms	618 ms
compiler-noDebug-javaUML2	1295 ms	136 ms	1431 ms

Comparing to the non-EDF version, the execution is slightly slower, because it checks whether to stop at possible breakpoints after every transition. The difference in the compilation time is insignificant, as the two solutions are practically using the same method to generate the same code.

1.8 Comparison of state machine and basic ALF action code generation to C++ and Java, without debugging support

- **Directory name:** compiler-noDebug-javaUML2-C-Java-alf-ast

- **Reuse:** These case study is reusing code from compiler-noDebug-javaUML2 and interpreter-noDebug-javaUML2-alf-ast-jit

- **Short description:** The single test model is compiled to C++ and Java languages. Both executable can create a predefined number of state machine instances, and send another number of random events to these machines. The behavior of compiled state machines are semantically equivalent with the case study described in 1.3, so basic action code with assignments, conditionals and calculations is supported. Event dispatcher is scheduled with round-robin. The state machine transition table is implemented with nested switch-case constructions in each language.

- **Technologies used:** Implemented in Java. Reads the model by the EMF UML2 API. Generates C++ and Java code using Java Emitter Templates. Reads and compiles ALF by its Xtext-generated AST. Compilation is done with javac and gcc.

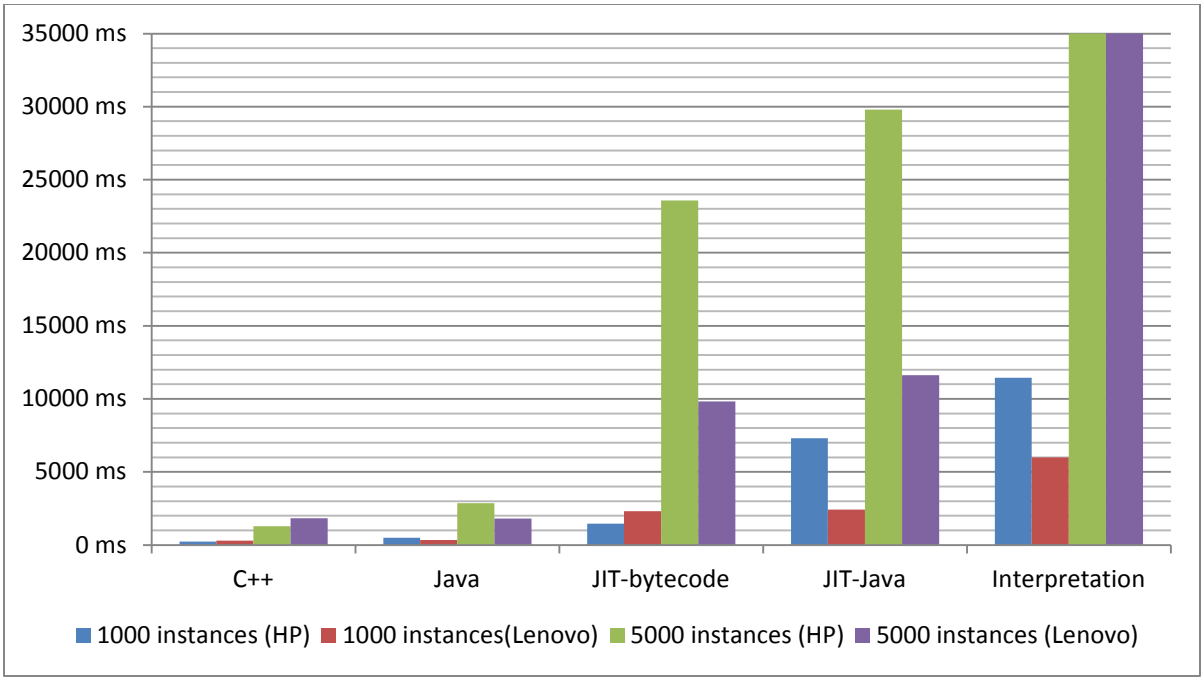
- **Results:** The measurements below are average values of 50 independent executions. They not include compilation and other conversion times, except just-in-time compilations in case of two interpreters, which were done on separate threads. Each machine received 10000 events. Tests were run with different number of state machine instances, on different hardware and software configurations. Results from previous case studies in section 1.3 are also detailed here to ease their comparison.
  Results on Ericsson managed workspace, HP EliteBook8540w, Intel Core i5 CPU @ 2.53 GHz, 4 GB RAM, 64 bit Windows 7:

Case study	1000 instances	5000 instances
Compiled to C++	233 ms	1291 ms
Compiled to Java	496 ms	2849 ms
Interpretation in Java	11435 ms	69288 ms
Interpretation in Java with JIT to Java	7295 ms	29796 ms
Interpretation in Java with JIT directly to bytecode	1468 ms	23583 ms

Results on Lenovo T530 (N1B54HV), Intel Core i5 CPU @ 2.60 GHz, 4 GB RAM, 64 bit Linux Mint 13:

Case study	1000 instances	5000 instances
Compiled to C++	304 ms	1821 ms
Compiled to Java	332 ms	1802 ms
Interpretation in Java	5990 ms	57815 ms
Interpretation in Java with JIT to Java	2412 ms	11615 ms
Interpretation in Java with JIT directly to bytecode	2321 ms	9824 ms

Results from both tables are summarized on the chart below. The maximum of the Y axis is limited to 35 seconds. As interpretation with 5000 instances took much more time than this, the two rightmost columns are truncated to make shortest run times displayable. The different characteristics of Java on the two different software configurations are clearly visible, as the hardware is almost the same for both machines. Interpreters with just-in time compilation are converging to the performance of compiled executables. As direct compilation to JVM bytecode is faster than through Java source text, the bytecode JIT converges faster. Compiled Java and C++ have almost the same performance because there were no really high-performance calculations in the tested models, and JVM also uses its JIT while running the bytecode.



1.9 Compilation with EMF-IncQuery and the EMF UML2 API

- **Directory name:** compiler-noDebug-IncQuery
- **Reuse:** This case study is reusing code from compiler-noDebug-javaUML2
- **Short description:** Includes a single-threaded abstract compiler which loads the input EMF UML2 model, finds all the classes inside the model and compiles a single state machine region from each of them into a Java class. The abstract compiler has two implemented realizations, one of which reads the model by the EMF UML2 API and one with the use of EMF-IncQuery. In the case of EMF-IncQuery, this case study aims to examine only the startup speed of the framework in order to see whether incrementality comes with a major cost in time at first compilation. In addition, the case study also compares the memory consumption of the two solutions. No action code processed or executed for state entries, exits or transition effects at all.
- **Technologies used:** Implemented in Java. Uses the EMF UML2 API and EMF-IncQuery.
- **Results:** The test cases were generated with a Java class also included in this case study. In each test case every class consisted of 10 states (plus the initial pseudostate) and 20 transitions (plus one from the initial pseudostate). Measurement results on DELL Inspiron N5110, Intel Core i5 CPU @ 2.50 GHz, 4 GB RAM, 64 bit Windows 7. **Note: This experiment only aims at evaluating the initial round of compilation, therefore IncQuery was not used for incremental translations.**

Number of classes in test case	Compilation time (in seconds)		Memory consumption (in MiBs)	
	EMF UML2 API	EMF-IncQuery	EMF UML2 API	EMF-IncQuery
10	2.5	2.7	20	23
100	12	15	58	80
1000	84	93	340	436
10000	1494 (25 mins)	2217 (37 mins)	713	2193

As the results show, there is no major cost of using EMF-IncQuery on small test cases. With the size of the model increasing, however, the difference also becomes greater, EMF-IncQuery being 1.5 times slower and using up 3 times more memory on the largest test case. The difference in memory consumption is between 30 and 40 percent on the second and third test cases with no significant difference in speed which shows that memory consumption is the more important matter to be taken into consideration when using EMF-IncQuery. This case study also made it clear that in case of a model compiler, incremental techniques must be used because otherwise compilation can become very slow.

1.10 Scheduler Case Studies

- **Directory name:** experiments/SchedulerCaseStudies

- **Short description:** This case study compares different methods to implement simulation of a scheduling mechanism. It compares the performance and capabilities of these methods. Results are described in /docs/slides.pptx. In this case study a simple example is manually implemented in the different methodologies, no code generation is done.

- **Technologies used:** The case study compares several methods:
    - Static scheduling (the order of the instructions is determined at compile-time) in C
    - Static scheduling in Java
    - Interpretation (one instruction is executed at a time)
    - Scheduling by multiple threads with blocking or yielding
    - Scheduling by coroutines in Java with the coroutines package
    - Scheduling by coroutines in C with Duff's device

- **Results:**
    - Interpretation speed is not comparable with the speed of native code execution. Even without action code, it is ~20 times slower than native execution.
    - Scheduling by multiple threads is not usable when running more than a few active objects. Thread pools are not usable when threads become blocked.
    - The coroutines package seems to be unreliable.
    - There is no big difference between the native execution speed of C and Java.

1.11 Compilation of UML2 model to C code with different scheduling strategies, without debugging

- **Directory name:** experiments/compiler-noDebug-javaUML2-schedule
- **Reuse:** This case study is reusing code from experiments/compiler-noDebug-javaUML2-file
- **Origin:** This case study is based on the results of experiments/SchedulerCaseStudies
- **Short description:** This case study loads a UML2 model and extracts a single state machine without action code. It compiles the state machine into a C class. Four different executables are built from the generated C class:
  - Pure evaluation – just evaluates the code without scheduling. State machines are run sequentially.
  - Non-deterministic evaluation – evaluates the code but switches between active state machines with a given probability.
  - Tracer evaluation – same as non-deterministic, but logs each executed instruction.
  - Follower evaluation – loads the output of the tracer and performs exactly the same sequence of instructions.
- **Technologies used:** The model is loaded using the UML2 API. C code is generated, make program is used to build the C files, the C code follows the coroutine paradigm with Duff's device.
- **Results:**
  - The performance of the scheduled code is comparable to the performance of the pure code. It slightly deteriorates with the increasing number of context switches, but the worst performance, when there is a context switch after every executed instruction is still only 10 times slower than native code, that is far better than the performance of the interpreted code, even without executing action code.
  - The performance of non-deterministic evaluation and follower evaluation were similar.

1.12 Compilation of UML2 model and Alf code with nondeterministic and pure scheduling to C code without debugging

- **Directory name:** experiments/compiler-noDebug-javaUML2-C-schedule-alf-ast

- **Reuse:** This case study is reusing code from experiments/interpreter-noDebug-javaUML2-alf-ast and a models from experiments/test-models

- **Origin:** This case study is based on the results of experiments/compiler-noDebug-javaUML2-schedule

- **Short description:** This case study loads a UML2 model and extracts a single state machine with action code specified in a comment in Alf language. It compiles the state machine into a C class. Two different executables are built from the generated C class:
  - Pure evaluation – just evaluates the code without scheduling. State machines are run sequentially.
  - Non-deterministic evaluation – evaluates the code but switches between active state machines with a given probability. Switch can occur after each instruction generated from Alf code.

- **Technologies used:** The model is loaded using the UML2 API, the Alf code is parsed using the org.eclipse.papyrus.alf.sandbox.editor.feature package. C code is generated, make program is used to build the C files the C code follows the coroutine paradigm with Duff's device.

- **Results:**
  - It is hard to generate code according to the Alf AST. It is mostly a technical problem. First, the AST is generated from an XText grammar, and parsed by an LL parser. The grammar had been manually left-factored to eliminate left-recursion, and the AST reflects this. There are lot of ambiguity in the grammar, that generates nodes like "`Feature_Or_SequenceOperationOrReductionOrExpansion_Or_Index`". I found no type resolution for the AST, so the types of the generated C fields are assumed to be int.
  - The performance benefit of code generation is even more obvious compared to the interpreted version of the case study found in experiments/ interpreter-noDebug-javaUML2-alf-ast.

1.13 Compilation of UML2 model to Java with action code, running as a Moka execution engine with Java Debug Interface support

- **Directory name:** compiler-moka-jdi-javaUML2

- **Reuse:** This case study is reusing code from compiler-noDebug-javaUML2-C-Java-alf-ast

- **Short description:** The implemented behavior is semantically equivalent with the case study documented in section 1.3. It is an Eclipse plugin, which adds a new execution engine into Moka. The user can launch a single state machine, which is first compiled to Java, then started in a separate

debugger process. When a Moka breakpoint on a state is hit, it is highlighted on the diagram, but does not suspend the execution.

- **Technologies used:** Implemented in Java. Reads the model by the EMF UML2 API. Generates Java code using Java Emitter Templates. Compiles the code using javac. Moka serves as an adapter on the top of Eclipse Debugging Framework, and also responsible for visualization. Breakpoint support is implemented with Java Debug Interface. JSR-045 is used to map source locations between the generated java code and the original model.

- **Results:** No technical problems found. However, as the implementation is partial the integration with Moka is not fully tested.

2 Summary

Follows the summary of the most important results of the case study experiments:

- EMF-UML2 is slow. Around 10 times speedup can be achieved when using a model representation optimized for interpretation.
- Alf Xtext grammar is left-factored, and very complicated in some cases. It was very hard and inefficient to implement even a basic interpreter on this representation.
- Direct compilation to JVM bytecode is more efficient than generating Java source and compiling it with javac, however this is a lot harder and more error-prone task as there is no real Java compiler in the process.
- In case external process is to be connected to the debugging framework: As using a TCP connection is very reliable, easy to use and relatively fast, it is recommended to connect an external process to a debugger. It is the most common solution used in many existing products.
- Integration with Eclipse required a lot of additional code compared to the initial code base size, but most of this code is general and reused from a tutorial. Writing the domain-specific parts of the debugger code was simple after the EDF cooperating layer was established.
- Comparing to the non-EDF version, the execution with debug support is slightly slower, because it checks whether to stop at possible breakpoints after every transition. The difference in the compilation time is insignificant.
- There is no major cost of using EMF-IncQuery on small test cases. With the size of the model increasing, however, the difference also becomes greater, EMF-IncQuery being 1.5 times slower and using up 3 times more memory on the largest test case used. Memory consumption is the more important matter to be taken into consideration.
- In case of a code generation, incremental techniques must be used because otherwise compilation can become very slow.

- Interpretation speed is not comparable with the speed of native code execution. Even without action code, it is ~20 times slower than native execution.
- Scheduling by multiple threads is not usable when running more than a few active objects. Thread pools are not usable when threads become blocked.
- The Java coroutines package seems to be unreliable.
- There is no big difference between the native execution speed of C and Java.
- The performance of the coroutine scheduled code is comparable to the performance of the pure code. It slightly deteriorates with the increasing number of context switches, but the worst performance, when there is a context switch after every executed instruction is still only 10 times slower than native code, that is far better than the performance of the interpreted code, even without executing action code.
- No technical problems found with the integration of generated code with the Moka framework.