Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

RUNTIME CODE GENERATION FOR INTERPRETED DOMAIN-SPECIFIC MODELING LANGUAGES

Tom Meyer
Tobias Helms
Tom Warnke
Adelinde M. Uhrmacher

Institute of Computer Science
University of Rostock
Albert-Einstein-Straße 22
18059 Rostock, GERMANY

ABSTRACT

Domain-specific modeling languages (DSMLs) facilitate concise and succinct model descriptions. DSMLs are commonly realized by defining a custom grammar and executing models in an interpreter. This provides flexibility in language design as well as in the implementation of simulation algorithms. However, this type of implementation can lead to a negative impact on simulation performance in comparison to implementing models in general-purpose programming languages (GPL). To mitigate this problem, we propose using runtime code generation. This allows us to replace specific parts of a model at runtime by compiling generated GPL source code. In this paper, we demonstrate the potential benefit of this concept based on ML-Rules, a DSML for modeling and simulating biochemical reaction networks. Using code generation for arithmetic expressions in ML-Rules' reaction rate functions, we were able to decrease the runtime by up to 40% in complex application scenarios.

1 INTRODUCTION

The development of a novel modeling method typically includes a software implementation of the method. One successful approach is to define domain-specific modeling languages (DSMLs) tailored to a specific application domain (Miller et al. 2010). Carefully designed DSMLs are powerful tools and enable modelers to efficiently develop succinct and accessible implementations of complex models. Especially in comparison to general-purpose programming languages (GPLs), using DSMLs results in more readable and compact model definitions.

Typically, DSMLs are accompanied by software to read, interpret and simulate models defined in the DSML. This enforces a separation of model and simulation algorithm, which enables reusing a simulation algorithm to execute various models, but also executing a model with different simulation algorithms (Himmelspach and Uhrmacher 2007). Compared with a GPL implementation, however, the DSML introduces an additional indirection between model definition and execution. For example, when a simulation model is directly implemented in a GPL and compiled with the GPL's compiler, the compiler can optimize the model code. Many compiler implementations employ sophisticated optimization strategies that can speed up the resulting program significantly (Bacon et al. 1994). However, a model that is defined in a DSML and interpreted by the already compiled interpreter cannot be optimized by such a compiler.

To reduce the gap between the interpreter and the compiler, we propose the use of runtime code generation. The idea is that the interpreter generates source code in the same language it is written in. The generated source code can then be compiled and optimized while the interpreter is running. This way,

978-1-5386-6572-5/18/$31.00 ©2018 IEEE

the interpreter can interpret the model and replace parts of it with generated, compiled, and optimized source code. Runtime code generation in general is a known strategy to improve the performance of interpreters (Würthinger et al. 2012), however, to our knowledge it has not been applied to DSMLs yet. Which parts of the model should be replaced with generated code depends on the concrete DSML.

In this paper, we demonstrate the potential benefit of this concept by applying it to the DSML ML-Rules (Maus et al. 2011) — a rule-based language to model and simulate biochemical reaction networks. Typical for DSMLs for biochemical reaction networks, the dynamics of a model are described by reaction rules consisting of reactants, products and arithmetic rate expressions. We identify these arithmetic rate expressions of reaction rules as a language part for which code generation could be beneficial, since they often produce a significant amount of computational effort and they can directly be mapped to concepts and structures of common GPLs. Therefore, we extend the existing ML-Rules implementation to generate code that replaces the rate expression in reaction rules.

The rest of the paper is structured as follows. In Section 2, we motivate the usage of DSMLs and discuss how applications processing them can be implemented. Afterward, in Section 3, we present possibilities to combine model interpretation and code generation using the programming language Java. Finally, in Section 4, we demonstrate how code generation can be integrated concretely within the model interpreter of the DSML ML-Rules and we show in Section 5 the potential performance benefit on benchmarks and complex models used in simulation studies. Section 6 concludes the paper.

2 DOMAIN-SPECIFIC MODELING LANGUAGES

Martin Fowler defines the term domain-specific language (DSL) as "a computer programming language of limited expressiveness focused on a particular domain" (Fowler 2010, pg. 27). We use the term domain-specific *modeling* language (DSML) to emphasize that the DSLs we consider in this article are specifically designed for specifying simulation models. However, many insights from DSL research are still valuable when working with DSMLs. For example, Fowler emphasizes that processing a DSL should populate a *semantic model*, which is a language-independent representation of the concepts of the DSL.

When developing a DSML, the according semantic model should represent the semantics of a concrete model description. Consequently, the semantic model encodes the meaning of a model and refers to the formal foundation of the DSML. This enables a separation of models and simulation algorithms — an important concept in modeling and simulation (Himmelspach and Uhrmacher 2007). It also facilitates the development of component-based simulation software supporting different simulation algorithms and different model representations.

Two types of DSLs are typically distinguished: *internal* (or *embedded*) and *external* DSLs (Fowler 2010). Internal DSLs are implemented with constructs of a host programming language. Thus, models implemented in the internal DSL are always also valid programs of the host language. Users of the internal DSL can exploit all existing tools (editors, visualizations, IDEs, ...) with essential features like syntax highlighting or auto-completion available for the host language. Additionally, all features of the host language can still be used when implementing a model. Even if the developer of an internal DSL does not integrate all needed features, a user can exploit all capabilities of the host language to realize missing options. Nevertheless, the designer of an internal DSL cannot freely design the syntax of the language, but is constrained by the syntax, and capabilities of the host language, possibly preventing the usage of common domain metaphors. The design of external DSLs is not constrained in this way, as they do not depend on a host language. Instead, the syntax of external DSLs can be defined by developing formal grammars. The toolchain to process formal grammars is typically generated automatically by parser generators like ANTLR (Parr 2013). However, further auxiliary tools either have to be developed manually or by using existing frameworks, e.g., xText (Eysholdt and Behrens 2010).

Many DSMLs are external DSLs exploiting the independent language design to suitably map concepts and metaphors of the application domain to language constructs. For example, a rule-based metaphor is suitable when dealing with biochemical reaction networks (see Section 4). Therefore, the direct description

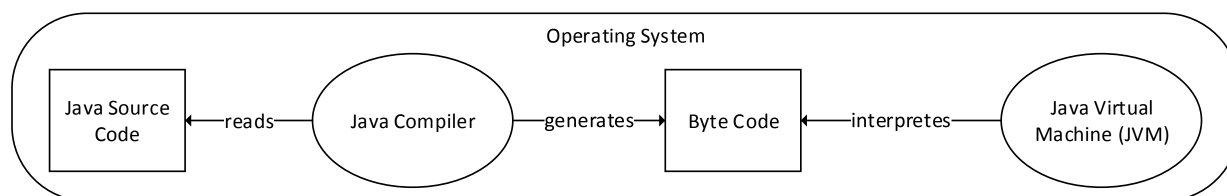

Figure 1: Java source code is compiled to bytecode by the Java compiler. The bytecode is then interpreted by the JVM. The JVM itself is executed on the operating system.

of rules with a tailored syntax improves the usability of DSMLs for this domain, resulting in the development of various external DSMLs for biochemical reaction networks, e.g., BioNetGen (Blinov et al. 2004), Kappa (Danos and Laneve 2004), ML-Space (Bittig and Uhrmacher 2017), and ML-Rules (Maus et al. 2011).

Models defined in an external DSML can be read in and simulated in two ways. Analog to compiling or interpreting general-purpose programming languages (Aho et al. 2006, Chapter 1.1), DSMLs can be executed via *code generation*, or directly in an *interpreter* (Morris et al. 2015). To generate code, the model is translated to source code in a general-purpose programming language (e.g., Java or C++). Afterward, the generated code can be compiled by a compiler of the target language. In contrast, the code executing the model in an interpreter is typically written in the same language as the parser. Here, the model is processed directly after parsing it, without generating another representation of the model.

Both approaches have pros and cons (Brambilla et al. 2012, pp. 28-33), e.g., interpreters allow changing models at runtime, whereby code generation allows a partial generation approach. Further, generated code is considered to be more efficient, since no interpretation is needed during runtime and the code can be optimized during generation time (Topçu et al. 2016, p. 33). Nevertheless, developing a program for code generation is challenging when dealing with abstract and declarative concepts (Iftikhar et al. 2016), possibly leading to the implementation of code generator only for suitable models not using all features of the DSML and an interpreter applicable to all models defined in the DSML. For example, in the simulation software Simulink® (www.mathworks.com/products/simulink) developed by The MathWorks, MATLAB® system blocks support two simulation modes (https://www.mathworks.com/help/simulink/ug/simulation-modes.html): either code generation is used only supporting a subset of features probably resulting in faster simulations; or model interpretation is used supporting all features. Code generation and model interpretation can also be mixed. For example, by first interpreting the model and then compiling selected model parts at runtime, the efficiency gain of compiler optimizations can be exploited while keeping a straightforward, lightweight interpreter for the model.

In general, code generation as well as interpretation can be implemented in all general purpose-programming languages and environments. However, to realize runtime code generation, languages running on virtual machines like Java are advantageous since programs in these languages are also interpreted. Therefore, program adaptation capabilities at runtime can be realized suitably (McKinley et al. 2004).

3 CODE GENERATION IN JAVA DURING RUNTIME

Java is a general-purpose programming language combining compilation and interpretation principles (Aho et al. 2006, Chapter 1.1). The common way to execute Java programs takes two basic steps, see Figure 1. First, the Java source code is compiled into a hardware independent, intermediate representation that is called bytecode. Second, the bytecode is interpreted and executed on a target machine by the Java Virtual Machine (JVM). Since Java programs are therefore executed by interpreting compiled bytecode, they can be adapted and extended during runtime, e.g., exploited by the Just-In-Time compiler to optimize the bytecode during runtime. Further, model interpreters written in Java can therefore be updated and extended during runtime to improve the execution efficiency of a simulation run. To manipulate a Java program at runtime,

one can directly update the bytecode that is executed by the JVM using bytecode manipulation libraries like ASM (Bruneton 2007, Chapter 1.2), ByteBuddy (Winterhalter 2018), or Javassist (Chiba 2000).

The ASM library is a utility to manipulate the bytecode which is used by the JVM. The name is a reference to the `__asm__` keyword in the C programming language, which is used to execute assembler code directly (Bruneton 2007, Chapter 1.2). Analogously, the ASM library provides an application programming interface (API) directly working on bytecode. It is possible to analyze, read, write and transform bytecode, utilizing the whole spectrum of bytecode features with all its optimization potential. Therefore, using ASM requires knowledge of the general concept and structure of bytecode. Nevertheless, implementing directly in bytecode is not really intended and developers are typically not familiar with it.

The code generation and manipulation library ByteBuddy builds on top of the ASM library (Winterhalter 2018). Its goal is to hide the whole bytecode complexity with an easy to use API. The API publishes functions with which one can search and alter certain class features (e.g., functions, member variables etc.) inside existing bytecode. This makes ByteBuddy more usable than the ASM library, since no bytecode is implemented directly. Unfortunately, it also narrows the expressiveness of the library to the features its API provides. It is not possible to consider arbitrary source code with ByteBuddy.

As with ByteBuddy, Javassist implements a reflective API on top of the ASM library (Chiba 2000). However, a design goal was to apply adaptations on source code level. Source code in Java syntax can directly be forwarded as strings to Javassist. The strings are compiled at runtime by a tailored Java compiler, i.e., the expressiveness of supported modifications is close to the expressiveness of Java source code and compiler optimizations are applied automatically to generate optimized bytecode. Since arbitrary strings can be processed by Javassist, neither syntax checks nor type checking can be done during development time, making it difficult to identify and fix bugs. However, in case of simple and short algorithms generated with Javassist, this issue should not be of significant importance.

4 COMBINING CODE GENERATION AND INTERPRETATION FOR BIOCHEMICAL REACTION NETWORKS

When modeling and simulating biochemical reaction networks, DSMLs are an essential tool. Therefore, various external DSMLs interpreting models for this domain have already been developed, e.g., BioNetGen (Blinov et al. 2004), Kappa (Danos and Laneve 2004), ML-Space (Bittig and Uhrmacher 2017), or ML-Rules (Helms et al. 2017). A biochemical reaction network is defined by a state vector $X = (x_1, x_2, \ldots, x_n) \in \mathbb{N}^n$ and a set of reactions $R = \{R_1, R_2, \ldots, R_m\}$ manipulating this state vector. A reaction R_i is defined by a change vector $v \in (v_1, v_2, \ldots, v_n) \in \mathbb{Z}^n$ and a propensity function $a_i : \mathbb{N}^n \to \mathbb{R}^+$. The change vector defines the effect when executing one reaction and the propensity function is used to calculate the rate of a reaction. When simulated deterministically, a reaction network is transformed to a set of ordinary differential equations and a numerical integration method is used to calculate the trajectory. When simulated stochastically, an algorithm based on the *Stochastic Simulation Algorithm* (Gillespie 1977) is typically applied and each reaction firing is calculated individually. For both simulation approaches, maintaining reaction rates is an essential part of the computational simulation effort.

As typical for the simulation of biochemical reaction networks, the calculation of reaction propensities often produces a significant computational effort of a simulation. Since rate expressions typically refer to arithmetic expressions, generating code to calculate rates is straightforward. A common data structure constructed by parsers is an abstract syntax tree (AST) representing the semantic model (Würthinger et al. 2012). When using ASTs, rate expressions are also represented by tree structures. Each node of such a tree structure represents an operation or a value; in our implementation all node classes extend the same class `Node`, see for example Figure 2, enabling the description of complex arithmetic expressions. Each `Node` class implements a method `calc(env)`, which is used to calculate the value of a node based on a given environment. The environment contains all values of constants or values of local variables. For example, calling `calc(env)` of an `Identifier` node returns the value of the given name stored in the environment. The value of a node representing a complex expression can simply be calculated recursively

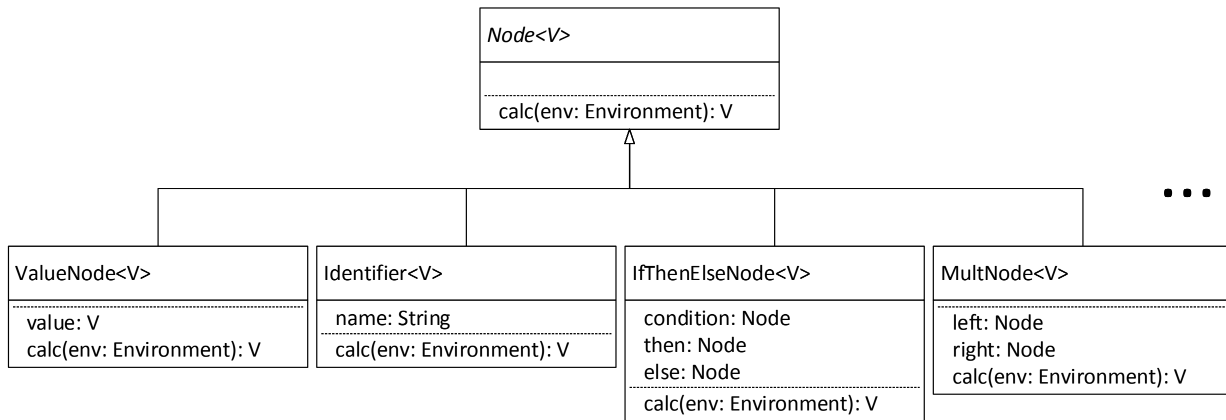

Figure 2: Class diagram of the node architecture used to represent expressions, e.g., see Figure 3.

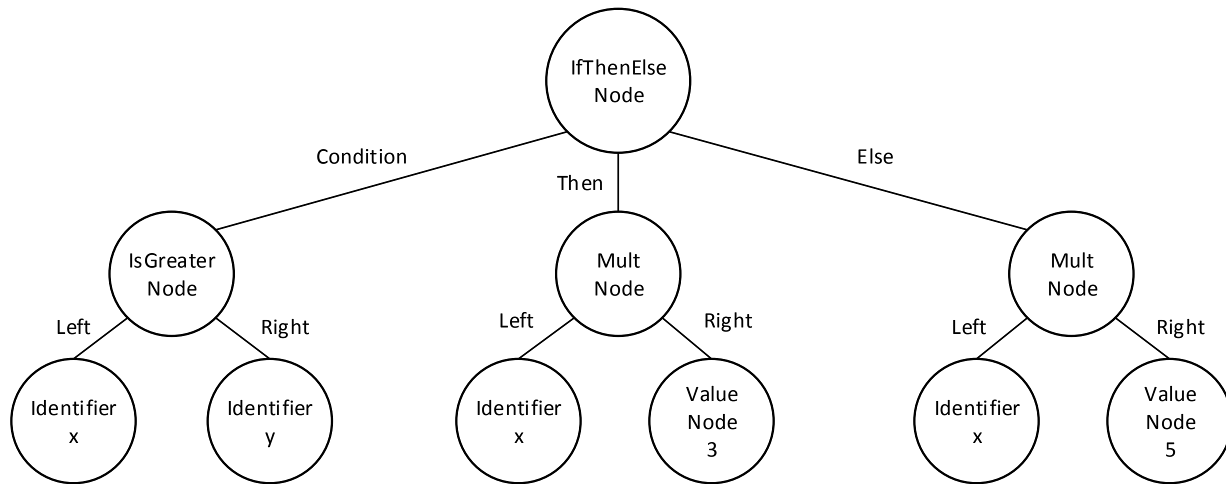

Figure 3: Tree structure representing the expression `if x > y then x · 3 else y · 5` using existing node classes.

by calculating the values of its sub nodes. However, since each value and each operation is represented by an object, multiple method calls are needed to calculate the value of a complex expression resulting in a computational overhead. For example, calculating the value of the tree representing the expression `if x > y then x · 3 else y · 5` shown in Figure 3 needs seven `calc(env)` calls (if either the `then` part or the `else` part are evaluated based on the result of the `condition`). By executing Java source code directly representing this expression as one tailored node, see Figure 4, this overhead can be reduced.

Such tailored nodes can be generated automatically during runtime using code generation. Therefore, after generating the semantic model, an additional post-processing is executed firstly checking whether nodes of rate expressions can be converted to code and secondly conducting AST rewriting (Würthinger et al. 2012) to apply newly compiled tailored nodes, see Figure 5. The idea is to iterate over all rate expressions and to try to generate new tailored node classes with Javassist during runtime representing rate expressions in the `calc` method. Initially, a whitelist of supported expression node classes for code generation has to be defined. In our implementation, the following arithmetic and logical expressions are supported: `AddNode`, `SubNode`, `MultNode`, `DivNode`, `IsSmallerNode`, `IsGreaterNode`, `IsEqualNode`, `AndNode`, `OrNode`, `IfThenElseNode`. Based on the whitelist, the code for a tailored node is produced

```
public class DN_XYZ extends DynamicNode {

  public Double calc(Environment env) {
    double x = env.getValue("x");
    double y = env.getValue("y");
    if (x > y) {
      return x * 3;
    } else {
      return y * 5;
    }
  }

}
```

Figure 4: An exemplary tailored node representing the calculation of the expression `if x > y then x · 3 else y · 5`. Calculating the value of this node should be more efficient than calculating the value of the corresponding node tree shown in Figure 3.

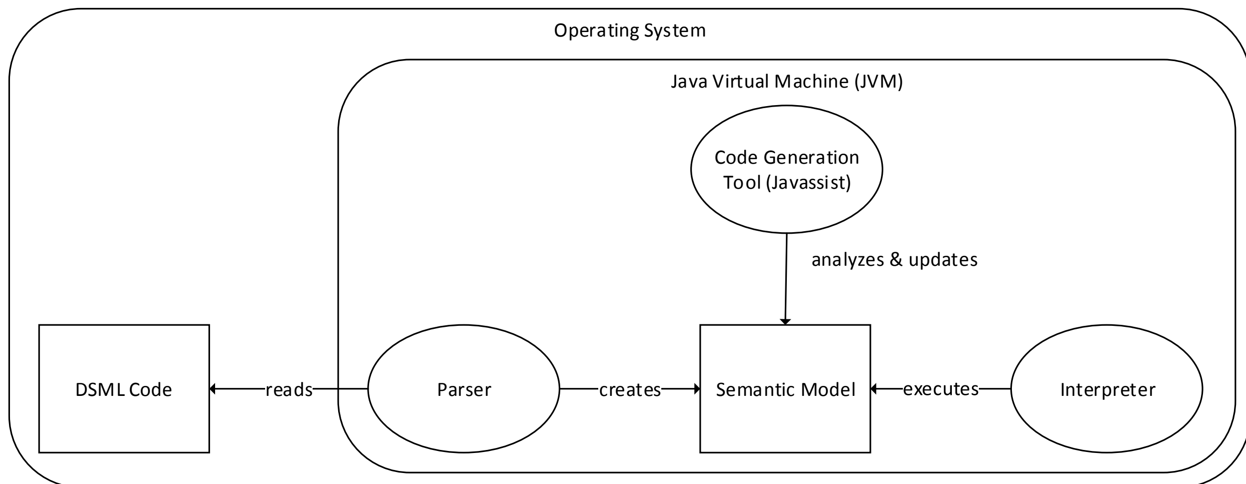

Figure 5: A parser, which is interpreted by the JVM, reads a DSML model and creates the semantic model. The semantic model is manipulated by the code generation tool using Javassist. The interpreter executes the updated semantic model.

by recursively generating code representing the calculation of a value of a node. Consequently, for each node class that shall be supported by the code generation, a transformation method has to be implemented. If a rate expression contains a node that is not contained in the whitelist, the process is stopped and no tailored node is created for this rate expression. Consequently, our approach currently does not support the creation of tailored nodes for parts of a rate expression. Based on the generated code, a new class for a tailored node can be created, compiled and linked to the JVM by Javassist methods. Thereby, the compiler automatically applies standard optimization features, e.g., to replace an expression with a constant result directly with this result.

Class names of generated classes are made unique using a universally unique identifier (UUID) to avoid class name conflicts when generating multiple node classes. To easily identify objects of generated

classes in existing code, all generated node classes extend the abstract class `DynamicNode` that is only extended by these classes. After generating a new node class for a rate expression, an object of this class is instantiated and the original node object is simply replaced by the new node object. Thereby, this adaptation is transparent for the rest of the software, i.e., to calculate the value of a newly generated node object, still only the `calc` method has to be called. All in all, the effort to implement a code generator for the rate nodes has been acceptable since arithmetic nodes like the `MultNode` or the `IfThenElseNode` map nicely to concepts and structures of Java.

5 EVALUATION WITH ML-RULES

We evaluate the benefit of runtime code generation of arithmetic rate expressions for biochemical reaction network simulation on the DSML ML-Rules (Helms et al. 2017), which is an internal DSML focusing on dynamically nested reaction networks, i.e., models with a dynamic state vector and a dynamic set of reactions. Nested and attributed entities are supported, rule schemes (patterns for reactions) can be defined supporting nested reactants. Further, propensity functions can be described by complex expressions, for example considering attributes of entities, content of entities or local properties of the model state. The grammar of ML-Rules is defined and implemented in ANTLR 4 (Parr 2013) and the ML-Rules parser is generated in Java. The semantic model calculated by the ML-Rules parser, which represents the meaning of the simulation model, consists of 1) an environment containing a map of all defined constants, functions, and entity types, 2) the initial state of the system, and 3) the set of rule schemes. After generating a semantic model of an ML-Rules model, an interpreter implemented in Java is applied calculating the simulation run with the semantic model.

To evaluate the benefit of code generation for arithmetic expressions during runtime for ML-Rules, we executed microbenchmarks and macrobenchmarks (Oaks 2014, Chapter 2). In Section 5.1, the results of microbenchmarks focusing on the benefit of code generation for isolated arithmetic expressions are shown. Section 5.2 presents the results of macrobenchmarks executing simulation runs with complex ML-Rules models. We use the benchmark framework `Java Microbenchmark Harness` (JMH, http://openjdk.java.net/projects/code-tools/jmh/) for the performance measurement. To consider best practices (Horký et al. 2015), warm up iterations are always executed. To avoid performance biases based on parallelism, all iterations of the experiments are executed sequentially. However, we explicitly do not disable the JIT compiler, since we want to consider whether the JIT compiler might equalize the benefit of the manually implemented code generation. All experiments are executed on a Lenovo Thinkpad T460 with Intel® Core™ i7-6600U CPU @ 2.60GHz, 16GB RAM, deactivated frequency scaling, Windows 7 Professional (64-bit), and Java 8 (Version 1.8.0_161).

5.1 Microbenchmarks

Microbenchmarks are executed to test an isolated small part of a software, e.g., the execution of a method call. Therefore, the performance results are only influenced by the software components which should explicitly be considered. Here, we measure the effort required to calculate the result of expressions once represented by node trees and once represented by `DynamicNodes` created by code generation. We use the following four expressions for these microbenchmarks:

1. `1 + 2`: An expression with a constant result. This expression should automatically be optimized by the compiler directly representing the result (`3`) to avoid repetitive calculation of the same result.
2. `x + y`: A simple expression calculating the sum of two variables `x` and `y`, i.e., an optimization analog to the first expression `1 + 2` is not possible.
3. `if x > 5 then 1 else 2`: A simple conditional expression. Generated Java code executing such a condition should be much faster compared to the execution of the corresponding node tree.
4. `(x1 + x2) - ((x3 / x4) + x2 + x4)`: A complex expression resulting in a node tree consisting of eleven nodes.

For the variables, we use the following values: $x = 5$, $y = 3$, $x1 = 1$, $x2 = 2$, $x3 = 3$, $x4 = 4$. The performance results to calculate the values of these expressions base on 500 iterations (100 warm up iterations) and are shown in Table 1. The results show that significant runtime savings are possible when applying code generation for the expressions. Referring to the optimization of the expression `1+2`, the compiler is optimizing this expression as expected and generates optimized bytecode directly using the result `3`. Clearly, such simple optimizations could also be implemented manually, but we argue that using existing established compiler capable of complex and sophisticated optimization methods are preferable to self-implemented solutions. For all other expressions, the runtime could be reduced by more than 80%. This demonstrates the overhead of interpreting nodes caused by additional method calls, object handling etc.

Table 1: Microbenchmark average runtime results in microseconds (standard deviation in brackets).

Expression	Node Tree	`DynamicNode`	Savings
`1 + 2`	0.081 (0.010)	0.001 (0.001)	99%
`x + y`	0.118 (0.035)	0.020 (0.006)	83%
`if x > 5 then 1 else 2`	0.094 (0.008)	0.012 (0.003)	87%
`(x1 + x2) - ((x3 / x4) + x2 + x4)`	0.414 (0.041)	0.072 (0.020)	82%

5.2 Macrobenchmarks

Macrobenchmarks are useful to evaluate the benefit of an optimization when executing a software as a whole. Therefore, we execute whole simulation runs including model compilation, code generation and all simulation step executions for the macrobenchmarks and the following runtime results consequently also consider the overhead of the runtime code generation. We use the following three models for these macrobenchmarks:

1. Wnt/β-catenin pathway model (Mazemondet et al. 2012): Represents the Wnt/β-catenin pathway including the production, degradation and phosphorilation of Axin proteins and the shuttling of β-catenin proteins between the cytosol of a cell and its nucleus. Cells and nuclei in this model do not change.
2. Dictyostelium discoideum model (Beccuti et al. 2015): A spatial model of Dictyostelium discoideum amoebas illustrating the aggregation process of those amoebas in space with multi-scale dynamics, i.e., amoeba internal dynamics are much faster than amoeba movements.
3. Eastern Baltic cod model (Pierce et al. 2017): An individual-based model of eastern Baltic cod in the Bornholm Basin including its metabolism and its behavior in dynamic, stratified water volumes.

In addition to different models, we also use different ML-Rules simulation algorithms (Helms et al. 2017) for the macrobenchmark experiments. Since the compartments (cells and nuclei) of the Wnt/β-catenin pathway model are fixed, the reaction network in this model does not change and therefore, a tailored simulation algorithm (`StaticSim`) for static reaction networks can be applied. Most of the computational effort in this model is needed by the reaction selection mechanism and rate updates. Due to the multi-scale dynamics of the Dictyostelium discoideum model, we select a hybrid simulator (`HybridSim`) combining deterministic and stochastic concepts to improve runtime efficiency (Helms et al. 2018). For this model, the numerical integration method calculating the deterministic part of the model producing most of the computational effort of a simulation run again mostly caused by rate updates. Finally, for the cod model, we use the default ML-Rules simulator (`StandardSim`). Rate updates do not dominate this model, however, the code generation is also applicable. Generally speaking, since in ML-Rules a semantic model is created by the parser and the code generation acts on the semantic model, the benefit of code generation can be exploited by all existing ML-Rules simulation algorithms — supporting the concept of a clear separation of

models and simulation algorithms. The simulation end times for the simulations are model-dependent and have been chosen based on executed simulation studies.

Table 2 shows the performance results of the simulation runs based on 20 iterations (5 warm up iterations). Since updating rate expressions is an essential part of the simulation runs for the Wnt/β-catenin pathway model and the Dictyostelium discoideum model, the needed runtime for the simulation runs could be reduced significantly by code generation for arithmetic expressions. Nevertheless, as expected, the benefit for the Eastern Baltic Cod model is not significant — most of the computational effort is produced by the maintenance of the reaction set and reaction rate updates are not central in this model. Besides, the overhead of the code generation and compilation at the beginning of a simulation run has been around 200 ms for all three models, so that it could worsen the runtime when executing simulation runs with lower runtimes. However, this issue could be solved at least for the execution of multiple simulation runs of the same model by reusing generated node classes.

Table 2: Macrobenchmark average runtime results in seconds (standard deviation in brackets). The last column shows the number of created `DynamicNode` classes per replication.

Model	Simulator	Node Tree	`DynamicNode`	Savings	#`DN`
Wnt/β-catenin pathway	`StaticSim`	2.17 (0.16)	1.72 (0.08)	21%	12
Dictyostelium discoideum	`HybridSim`	30.11 (2.18)	17.26 (1.55)	43%	24
Eastern Baltic cod	`StandardSim`	11.11 (1.17)	10.52 (0.79)	5%	8

6 CONCLUSION

In this paper, we demonstrate the potential benefit of runtime code generation for interpreted DSMLs illustrated on ML-Rules— a DSML for dynamically nested biochemical reaction networks. Domain-specific modeling languages enable modelers to efficiently develop succinct implementations of complex models. Models implemented in an external DSML are executed by either using code generation or by applying an interpreter. Code generation is considered to produce code that is more efficient to be executed than interpreting models. However, implementing interpreters can be more suitable in case of abstract and declarative language concepts. To combine the effectiveness of code generation for suitable parts of an interpreted DSML, both approaches can be coupled, e.g., by using reflective concepts typically provided by interpreted programming languages like Java running on a virtual machine. We show the benefit of this combination by introducing code generation at runtime for arithmetic rate expressions in ML-Rules. To deal with arithmetic expressions in DSML interpreters, they are typically translated into node trees, thereby introducing a calculation overhead caused by additional method calls, object handling etc. Code generation at runtime can be used to mitigate this overhead by replacing a complex node tree with one semantically equivalent tailored node. This can be easily realized for arithmetic expressions, since they map nicely to language concepts and structures of general purpose languages, such as Java. The performance evaluation demonstrates the potential of code generation at runtime. The runtime of simulation runs of complex models used in simulation studies could be reduced by up to 40%. Clearly, the effectiveness of code generation at runtime depends on the model. However, since the concept can be applied straightforwardly and with little effort to interpreted DSMLs, it appears worthwhile to analyze and exploit its potential, possibly beyond arithmetic expressions for interpreted DSML.

The runtime code generation will be part of one of the next ML-Rules releases. Further, an artifact containing the source code and applications to reproduce the performance results is available at https://doi.org/10.18453/rosdok_id00000109.

ACKNOWLEDGMENTS

This research is supported by the German Research Foundation (DFG) via research grants ESCeMMo (UH-66/14).

REFERENCES

Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools*. 2nd ed. Boston, MA, USA: Addison-Wesley.

Bacon, D. F., S. L. Graham, and O. J. Sharp. 1994. "Compiler Transformations for High-performance Computing". *ACM Computing Surveys* 26(4):345–420.

Beccuti, M., M. A. Blätke, M. Falk, S. Hardy, M. Heiner, C. Maus, S. Nähring, and C. Rohr. 2015. "Dictyostelium discoideum: Aggregation and Synchronisation of Amoebas in Time and Space". *Dagstuhl Reports: Multiscale Spatial Computational Systems Biology (Dagstuhl Seminar 14481)* 4(11):195–214.

Bittig, A. T., and A. M. Uhrmacher. 2017. "ML-Space: Hybrid Spatial Gillespie and Particle Simulation of Multi-Level Rule-Based Models in Cell Biology". *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 14(6):1339–1349.

Blinov, M. L., J. R. Faeder, B. Goldstein, and W. S. Hlavacek. 2004. "BioNetGen: Software for Rule-based Modeling of Signal Transduction Based on the Interactions of Molecular Domains". *Bioinformatics* 20(17):3289–3291.

Brambilla, M., J. Cabot, and M. Wimmer. 2012. *Model-Driven Software Engineering in Practice*. 1st ed. Morgan & Claypool Publishers.

Bruneton, Eric 2007. "ASM 4.0 A Java bytecode engineering library". http://download.forge.objectweb.org/asm/asmguide.pdf, Accessed: January 10, 2018.

Chiba, S. 2000. "Load-Time Structural Reflection in Java". In *Proceedings of the 14th European Conference on Object-Oriented Programming*, 313–336. Berlin, Heidelberg: Springer.

Danos, V., and C. Laneve. 2004. "Formal molecular biology". *Theoretical Computer Science* 325(1):69–110.

Eysholdt, M., and H. Behrens. 2010. "Xtext: Implement Your Language Faster Than the Quick and Dirty Way". In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, 307–309. New York, NY, USA: ACM.

Fowler, M. 2010. *Domain-specific languages*. Upper Saddle River, NJ, USA: Addison-Wesley.

Gillespie, D. T. 1977. "Exact Stochastic Simulation of Coupled Chemical Reactions". *The Journal of Physical Chemistry* 81(25):2340–2361.

Helms, T., T. Warnke, C. Maus, and A. M. Uhrmacher. 2017. "Semantics and Efficient Simulation Algorithms of an Expressive Multi-Level Modeling Language". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 27(2):8:1–8:25.

Helms, T., P. Wilsdorf, and A. M. Uhrmacher. 2018. "Hybrid Simulation of Dynamic Reaction Networks in Multi-Level Models". In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, 133–144. New York, NY, USA: ACM.

Himmelspach, J., and A. M. Uhrmacher. 2007. "Plug'n Simulate". In *Proceedings of the 40th Annual Simulation Symposium (ANSS'07)*, 137–143. Washington, DC, USA: IEEE.

Horký, V., P. Libič, A. Steinhauser, and P. Tůma. 2015. "DOs and DON'Ts of Conducting Performance Measurements in Java". In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 337–340. New York, NY, USA: ACM.

Iftikhar, M. U., J. Lundberg, and D. Weyns. 2016. "A Model Interpreter for Timed Automata". In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, edited by T. Margaria and B. Steffen, 243–258. Cham, Switzerland: Springer International Publishing.

Maus, C., S. Rybacki, and A. M. Uhrmacher. 2011. "Rule-based Multi-level Modeling of Cell Biological Systems". *BMC Systems Biology* 5(166).

Mazemondet, O., M. John, S. Leye, A. Rolfs, and A. M. Uhrmacher. 2012. “Elucidating the Sources of β-Catenin Dynamics in Human Neural Progenitor Cells”. *PLoS ONE* 7(8):1–12.

McKinley, P. K., S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. 2004. “Composing Adaptive Software”. *Computer* 37(7):56–64.

Miller, J. A., J. Han, and M. Hybinette. 2010. “Using Domain Specific Language for modeling and simulation: ScalaTion as a case study”. In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson et al. Piscataway, New Jersey: IEEE.

Morris, K. A., M. Allison, F. M. Costa, J. Wei, and P. J. Clarke. 2015. “An Adaptive Middleware Design to Support the Dynamic Interpretation of Domain-specific Models”. *Information and Software Technology* 62:21–41.

Oaks, S. 2014. *Java Performance: The Definitive Guide*. 1st ed. Sebastopol, CA, USA: O’Reilly Media.

Parr, T. 2013. *The Definitive ANTLR 4 Reference*. 2nd ed. Frisco, TX, USA: Pragmatic Bookshelf.

Pierce, M. E., T. Warnke, U. Krumme, T. Helms, C. Hammer, and A. M. Uhrmacher. 2017. “Developing and Validating a Multi-level Ecological Model of Eastern Baltic Cod (Gadus morhua) in the Bornholm Basin - A Case for Domain-specific Languages”. *Ecological Modelling* 361:49–65.

Topçu, O., U. Durak, H. Oğuztüzün, and L. Yilmaz. 2016. *Distributed Simulation: A Model Driven Engineering Approach*. Cham, Switzerland: Springer International Publishing.

Winterhalter, Rafael 2018. “ByteBuddy Website”. http://bytebuddy.net/, Accessed: January 10, 2018.

Würthinger, T., A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. 2012. “Self-optimizing AST Interpreters”. In *Proceedings of the 8th Symposium on Dynamic Languages*, 73–82. New York, NY, USA: ACM.

AUTHOR BIOGRAPHIES

TOM MEYER holds a bachelor in computer science and is currently a master student. His studies focus on models and algorithms. His email address is tom.meyer@uni-rostock.de.

TOBIAS HELMS holds a PhD in Computer Science from the University of Rostock and is currently a post-doc at the Institute of Computer Science of the University of Rostock. His research focuses on the development of adaptive and efficient simulation algorithms for dynamically nested biochemical reaction networks. His email address is tobias.helms@uni-rostock.de.

TOM WARNKE holds a master in computer science and is currently a PhD student at the Institute of Computer Science of the University of Rostock. His research focuses on the development of domain-specific languages for modeling and simulation. His email address is tom.warnke@uni-rostock.de.

ADELINDE M. UHRMACHER is professor at the Institute of Computer Science of the University of Rostock and head of the research group Modeling and Simulation. She holds a PhD in Computer Science from the University of Koblenz and a Habilitation in Computer Science from the University of Ulm. Her email address is adelinde.uhrmacher@uni-rostock.de.