On the Performance of UML State Machine Interpretation at Runtime

Edzard Höfig and Peter H. Deussen
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
{edzard.hoefig|peter.deussen}@fokus.fraunhofer.de

Ina Schieferdecker
Technical University of Berlin
Franklinstr. 28/29
10587 Berlin, Germany
ina@cs.tu-berlin.de

ABSTRACT

Modelling system behaviour by means of UML Behavioral State Machines is an established practice in software engineering. Usually, code generation is employed to create a system's software components. Although this approach yields software with a good runtime performance, the resulting system behaviour is static. Changes to the behaviour model necessarily provoke an iteration in the code generation workflow and a re-deployment of the generated artefacts. In the area of autonomic systems engineering, it is assumed that systems are able to adapt their runtime behaviour in response to a changing context. Thus, the constraints imposed by a code generation approach make runtime adaptation difficult, if not impossible. This article investigates a solution to this problem by employing interpretation techniques for the runtime execution of UML State Machines, enabling the adaptability of a system's runtime behaviour on the level of single model elements. This is done by devising concepts for behaviour model interpretation, which are then used in a proof-of-concept implementation to demonstrate the feasibility of the approach. For a quantitative evaluation we provide a performance comparison between the proof-of-concept implementation and generated code for a number of benchmark models. We find that UML State Machine interpretation has a performance overhead when compared with static code generation, but found it to be adequate for the majority of situations, except when dealing with high-throughput or delay-sensitive data.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems; D.3.4 [**Software**]: Programming Languages—*Processors*; I.1.3 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation—*Languages and Systems*

General Terms

Model, Interpretation, Runtime, Performance

Keywords

UML state machine, Models at runtime, System adaptation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SEAMS' 11, May 23–24, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0575-4/11/05 ...$10.00

1. INTRODUCTION

Autonomic Systems (AS) engineering research is concerned with the creation, assessment and maintenance of systems that are able to execute management decisions without direct human control.

Our work in this field was mainly inspired by the concept of Autonomic Communication, as outlined in [28]; Autonomic Communication aims at applying a set of computing principles, originally developed by IBM [17], to the management of communication networks and services. The idea is to delegate management decisions from human administrators to devices operating in the network with the goal of reducing the administration complexity and consequently the operational expenditures, as well as to enable the *evolvability* of systems. Designing a system for evolvability implies a scalability not only in terms of resource utilisation, but also in terms of functionality. For achieving such *functional scalability*, it is necessary to design a system in a way that it is able to continue to operate, without human intervention, in the face of changes in its environment. We denote such an ability with the term *homeostasis*, in reference to the concept introduced by cyberneticist W.R. Ashby [2, chapter 5/3]. The major challenge of functional scalability is to enable AS to adapt to environmental changes that are unknown at the design time of the system.

Homeostasis requires that a system is able to modify its own behaviour, extending or re-shaping its functionality to suit an operational context. From a software engineering perspective, this requires employing a suitable format for behaviour representation and an adequate mechanism that executes the behaviour, while allowing for a modification of the underlying representation itself. For the former we propose to use models. Thus, such a behaviour representation format is called a Behaviour Model (BM). For the latter we propose to use interpretation mechanisms. By "interpretation" we refer to the direct evaluation of a BM at runtime, which also includes the ability to dynamically modify the BM. Depending on the formalism that expresses system behaviour, an adaptation of the BM can be easy or more difficult. We restrict ourselves to the study of BMs that are defined by the UML Behavioral State Machine formalism [23, Section 15] and use the terms "BM" and "UML State Machine" as synonyms within this text.

Utilising BMs is a good approach for implementing self-regulating systems. This has been shown for cases where the

system behaviour can be modelled numerically and where only a small number of variables need to be considered e.g. in hardware systems that regulate mechanical processes based on control theory, like a brake assistance system in a car. We think that the general idea of using BMs for the formalisation of runtime behaviour is also applicable to ASs, although there are a number of considerable differences. For example, a numerical modelling of self-regulating systems based on control theory uses a closed-world assumption: the complete parameter space of the system behaviour is supposed to be known at design time. Therefore, a BM using this approach has to be considered static at runtime and it cannot be modified to react to hitherto unknown input events. In ASs, due to the homeostatic property, an open-world assumption has to be made: systems are part of a changing environment and need to be able to change themselves to react to previously unknown events. This requires the ability to dynamically modify the BM at runtime, which is something that has not been thoroughly investigated before.

The main contributions of our work are concepts for adaptive software architectures and the demonstration of implementation mechanisms that support the runtime adaptation of software based on UML State Machines. We also provide a clear identification of the performance issues connected with these.

1.1 Research Hypothesis

Runtime modification of behaviour that is modelled using UML State Machines is not a well researched topic. There is little research available on the feasibility and properties of mechanisms that enable the modification of ASs that rely on models for the specification of behaviour. Such mechanisms are the bare minimum that is necessary for implementing any form of runtime behaviour optimisation for systems, which are designed with an open-world assumption in mind.

We assume that it is viable to execute BMs by interpretation at runtime, instead of using a BM to generate code in a programming language, which would then be compiled and executed. It has only been shown very recently that UML State Machine interpretation is practicable at all [3] and there is no substantial information on the performance or implementation of such mechanisms.

Experience dictates that it is computationally more expensive to employ an interpretation mechanism than to rely on compilation techniques, thus, we believe that a certain performance penalty is incurred by following the interpretation approach. We hypothesise that the innovative benefits of BM interpretation outweigh its performance disadvantages, but will need to determine the incurred performance penalty, to show this. Thus, we will measure an interpretation mechanism's performance using its runtime execution speed and memory consumption characteristics and by comparing these values to the execution characteristics of a mechanism based on compilation we will be able to accurately determine the performance differences between the two approaches. Such a comparison is only possible when based on a common measuring platform and we will use a novel performance benchmark for this purpose.

1.2 Document Structure

This paper is structured in six sections. This first section contains the introduction. The second section discusses related work. Within the third section we establish the key concepts required for BM interpretation. In the fourth section we introduce a generic interpreter architecture and describe the implementation of the UML State Machine interpreter. In the fifth section we are presenting a performance analysis of the interpreter and compare the results to an implementation that uses generated code. Section six contains a summary of the results of our work.

2. RELATED WORK

2.1 Statecharts

Statecharts have been integrated with the UML since the first version of the standard. Currently, UML State Machines are the most popular, statechart-based BM formalism in use today, which is the reason why we are using it.

The statechart formalism was invented in the 1980's by D. Harel to describe complex, reactive event systems [12]. The formalism builds on higraphs which are defined in [13, Appendix]. In a recent article [16] we proposed a formalisation for statecharts better suited to statechart execution on embedded platforms. Higraphs can be understood as the main innovation that statecharts contribute to the field of automata theory and they contribute two interesting features: composition and concurrency. Composition refers to the support for aggregation of states within a superstate, while concurrency relates to a support of parallel execution of behaviours. D. Harel later concentrated on investigating the modelling of executable objects with statecharts, together with E. Gery [14] and after a year O. Grossman and D. Harel published [11], within which they define syntax and semantics of higraphs in way that minimises (but not circumvents) interpretation ambiguities.

2.2 Interpretation and Execution

We found a usage of the term "interpretation" in conjunction with statecharts in an article from the early nineties by J. Ebert [7]. The paper contains a discussion of the operational semantics of statecharts, as well as a description of algorithms for their interpretation and validation. It does not explore the implementation from a practical perspective, but discusses the subject theoretically. Although the concept of statechart interpretation exists already for some time, it is only in the last two or three years that BM interpretation has become a hot topic. During the work on this thesis we saw the emergence of three other interpreters for statecharts. On the one hand the MOCAS Engine by C. Ballagny [3], which interprets UML State Machines based on EMF in a similar manner to the UML interpreter that we created; on the other hand the Apache [1] and the QT [21] State Chart XML (SCXML) engines. By an inspection of the source code, we found that all of these implementations are maintaining BMs in dynamically modifiable runtime structures. Thus, they could theoretically support the adaptation of an executing BM at runtime, but only the MOCAS system does currently provide support for this feature. Compared to our implementation, the MOCAS engine is better integrated with other UML features, but less sophisticated in regard to the interpretation of conditions and actions. There is also a large body of research on the interpretation performance of programming languages, but we did not find it to be applicable as the discussion of programming language interpretation performance is usually too closely related to a specific hardware or platform.

Code Generation.

Although interpretation of statecharts was discussed early in the 1990s, the vast majority of realisations of executable models have been implemented with automatic or semi-automatic code generation approaches, even when the BMs are employed only for simulation purposes (as done by, e.g. the Rational Rhapsody tool). Code generation from statecharts itself is a well researched problem [18, 20], with applications of the formalism going back to the early 1990s [9] and an active ongoing research, i.e. see [26].

The Shlaer-Mellor Method and Executable UML.

Developed by S. Shlaer and S. Mellor in the late 1980s, the method introduced OO techniques for model-based system analysis and design [27]. Shlaer-Mellor uses Finite State Machines (FSM) for specification of behaviour, but lacks a single, standard expression language for specification of action expressions. This shortcoming was addressed with the specification of *UML action semantics*, which lead to the Executable UML (xUML). Considered a successor to the Shlaer-Mellor method, xUML was created within the MDA research direction as an UML profile. The profile restricts the UML to elements that have clear execution semantics, while adding the missing action semantics. More information can be found in, e.g. the book by S.J. Mellor and M.J. Balcer [19].

Although the employed UML action semantics prescribe the necessary details for integration of the UML with an expression language, they do not specify a concrete syntax [30, Section 2.4]). Regarding its relation to our work, xUML is an approach that employs BMs for specification of system dynamics, with the goal of generating code from these specifications. We later demonstrate the general possibility of interpreting the complete set of features for UML State Machines; thus, an adaptation of the interpretation approach to the restricted set of xUML features should be simple. The approach complements our work, insofar as we did not examine the binding of other UML diagram types (especially static ones for data & structural modelling) to expression language statements – exactly this has been done in the UML action semantics research.

UML State Machine Execution Semantics.

The execution semantics of UML State Machines are commonly criticised for being not adequately formalised, and a substantial amount of effort has been invested into the identification of weak points in the standard, as well as in the creation of execution semantics using alternative formalisms [4]. Using predicate logic, H. Fecher, et al. formulated an execution semantics of UML for model checking purposes [8], although they have omitted some pseudo-states (choice, junction, terminate) and nesting constructs (submachines). X. Than et al. [31] specify execution semantics in the Z language but also examine only core statechart constructs (states, transitions, state containment, concurrency). A treatment of the ambiguous semantics of the history concept within UML State Machines is provided by A. Derezińska and R. Pilitowski [6].

2.3 Performance Assessment

The usage of FSM benchmarks has a long history in the field of logic synthesis, which is concerned with the mapping of a logical expressions to (optimal) connections of transistors using a given hardware technology. The most popular collection of benchmarks for logic synthesis from FSMs is referred to as the ACM/SIGDA[1] benchmarks, which are archived by the North Carolina State University at the Collaborative Benchmarking and Experimental Algorithmics Laboratory [22]. Most of the models are older than 15 years, but they are still widely referenced and used.

There is no commonly accepted benchmark suite that utilises UML State Machines for performance assessment and we found only a very small number of publications documenting the utilisation of single proprietary statecharts for this purpose. Among the employed statecharts is a calculator example from the book on statecharts by M. Samek [25] and the alarm watch example used by D. Harel [12].

For the general performance assessment of computing systems, there are a number of well known standard benchmark suites. For example, the SPEC-cpu2000 benchmark suite [29] or the Dhrystone benchmark [35]. Such benchmarks will asses the performance of a hardware platform and are not suitable for evaluating software execution mechanisms. For this area other benchmarks have been devised, for example the DaCapo benchmark suite [33], which is intended as a tool for benchmarking the runtime performance of the Java programming language in relation to implementation aspects like memory management and the binding to the underlying computer architecture.

We did not find any suitable benchmark suite that will allow us to assess the performance of a statechart execution engine accurately and in the necessary detail. Either the benchmarks are defined too simple, without taking statechart specific features into account (e.g. the ones that are based on FSMs) or they are using statecharts, but only in the form of a single, exemplary model. This is not sufficient for an in-depth study of the various aspects of an execution system. General performance benchmarks are interesting as a documentation of the current best-practices for benchmark design, but we found that such benchmark suites are not specific enough for our purposes. The current state of the art in this area forces us to create our own, more suitable, benchmark scenarios.

3. UML BEHAVIORAL STATE MACHINE INTERPRETATION

We already stated that the interpretation of BMs at runtime is fundamentally different from BM execution that employs code generation. With code generation, a traceable relationship between the executable representation and the original model elements, which were used to generate the executable artefacts, is not given. Therefore, it is difficult to adapt the system at runtime. Often the only solution is to stop the currently running system instance and to restart the system with a different binary executable. With BM interpretation, the runtime format of the executable model is kept and changes can be applied directly to the model.

Figure 1 demonstrates the differences between the two approaches by showing the diverse formats and artefacts employed during BM execution, together with the relations between them. The diagram is structured horizontally and vertically. The grey-shaded, horizontal boxes represent the three different classes of employed formats, the four vertical

[1]Association for Computing Machinery – Special Interest Group on Design Automation

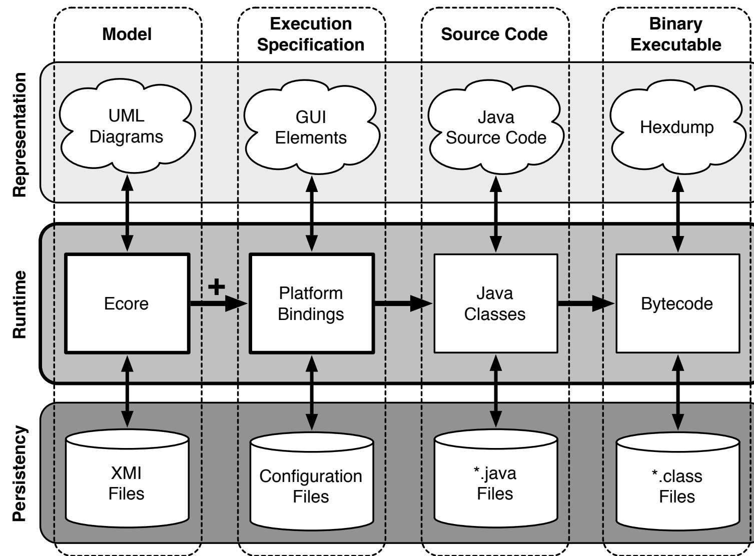

Figure 1: Formats and exemplary artefacts used in BM execution processes

boxes, depicted using dashed lines, represent the four different types of artefacts. For each combination of artefact type and format class, we are providing the concrete artefact format used for the UML interpreter. The three different classes of formats are: **Representation** formats that are used for creation, visualisation and modification of system behaviour; **Runtime** formats that are in-memory formats used to hold the necessary information that allows runtime mechanisms to operate on a behaviour specification; and **Persistency** formats that serve as conventions on how to persistently store system behaviour information on mass storage media.

When using BM execution, the system behaviour description goes through a number of phases from specification to execution and each phase employes specific artefact types. Artefact types of a single phase are kept closely synchronised. A change in one of them will trigger corresponding changes for artefacts of the two other format classes. The four different artefact types are: **Model** artefacts that describe the behaviour encoded by a BM, without specifying how the behaviour relates to an execution platform. The **Execution Specification** artefacts provide the necessary information to bind a BM to a concrete execution platform, they complement the **Model** artefacts and together carry sufficient information to interpret a BM at runtime[2]. The **Source Code** artefacts are created using code generation with the platform-independent **Model** and the platform-dependent **Execution Specification** as input to the transformation mechanism. They are a representation of the BM in a programming language. Finally, the **Binary Executable** artefacts are necessary for executing the transformed BM on a target platform.

The Interpretation of BMs uses only two **Runtime** formats: **Model** and **Execution Specification**. This approach maintains the model structure during execution. In contrast, BM execution that is based on code generation introduces a number of additional artefacts. The code generation process first transforms the model into **Source Code**, which is then compiled into **Binary Executable** artefacts. This is shown in Figure 1 for a Java-based toolchain. Although these transformations result in an execution mechanism that performs better, due to closer bindings to the execution platform, they effectively prevent model adaptation at runtime. Should a modification of the model or the execution specification take place, the code generation transformations need to be reapplied and the resulting **Binary Executable** artefacts would need to be re-deployed, necessitating a system re-start. The benefit of BM interpretation is, that it only relies on the closely linked **Model** and **Execution Specification** artefacts and modifications can thus be directly applied to the executing model during the runtime of the system, without the need for transformations or generation of further artefacts.

3.1 Initialisation from Model Specification

At the start of a BM interpretation process, BM specifications are usually provided in a **Model Persistency** format to the interpreter. The interpreter has to first parse the given format in order to construct a runtime representation of the BM, which is a task that is normally accomplished through libraries, provided as part of the **Runtime** or **Persistency** format implementations. At this stage a check of the BM's syntax is employed, to make sure that a BM conforms to the declared **Runtime** format.

After successfully creating an in-memory, **Runtime** representation of the model, an interpreter has to make sure that all dependencies of the model are met. This includes satisfaction of dependencies regarding required resources, as well as parametrisation of the model and setting up of support for the expression language used in guard conditions and action statements.

To determine the resource dependencies of the model, an interpreter has to analyse the transition labels found in the BM for use of platform resources in the guard condition or action expressions. The interpreter can also make sure that these resources are available. An alternative way for dependency resolution is to explicitly provide the dependencies

[2]This fact is highlighted by the bolder outlines of the corresponding artefacts in the diagram.

as part of the BM specification within the **Persistency format**. An example of this technique is the use of the UML ElementImport feature to define the platform bindings. In this case, the interfaces to the platform and of all of the utilised resources need to be modelled within the UML, as well. If the dependencies are gathered implicitly from the transition labels, a suitable query language for the **Model Runtime** or **Model Persistency** formats provides an advantage. For EMF, there is an in-memory query language provided under the name "Model Query". For the **Persistency** format, an XPath statement could be used to isolate the necessary statements in a file that conforms to the XML Metadata Interchange format (XMI). Once the transition labels have been isolated, they need to be analysed within the context of the employed expression language to identify statements that rely on platform bindings, e.g. the invocation of a method on an object.

Resolution of the required platform dependencies is only a part of the initialisation process. BMs might also require the manual specification of model parameters. For example, if a BM is used to query a remote service that requires user credentials for authentication, these credentials should be provided to the BM interpreter as parameters at initialisation time. The result of the initialisation process is a **Runtime Model** with a matching **Execution Specification** that specifies the resource bindings to the execution platform, along with a binding of values to BM parameters. Before commencing interpretation, an interpreter has to prepare the runtime system by creating a BM instance. The interpreter needs to allocate and initialise context data objects, hook up communication channels for internal and external event transmission, create message queues for buffering events, create the bindings to resources and set up the runtime data structures for managing the active and historical BM state configuration.

4. IMPLEMENTATION

The following section discusses an implementation of a BM interpreter for UML 2 Behavioral State Machines. The intention is to create the necessary mechanisms that we need to determine the performance impact of the corresponding mechanisms, as well as to gain insight into the implementation aspects that dominate the runtime performance characteristics of the approach. Our goal is not to develop a general execution semantics for UML.

For runtime interpretation purposes, we found it useful to extend UML State Machines with a proprietary profile, which we termed the Adaptive Systems Profile (ASP). The ASP contains the additional information that enables the UML interpreter to construct an execution specification of the BM. It currently consists of a number of stereotypes, which are extending the UML meta model. For our discussion, two stereotypes are important: the **RuntimeBehavior** stereotype, which extends the StateMachine class, allowing a BM interpreter to identify which state machines are designed for runtime interpretation and the **ContextImport** stereotype, which is used to reference instance specifications as part of the context data.

4.1 Architecture and Operation

Figure 2 depicts the high-level architecture of a single UML interpreter. It is a system with two interfaces to the environment: the Control Interface and the Event Interface.

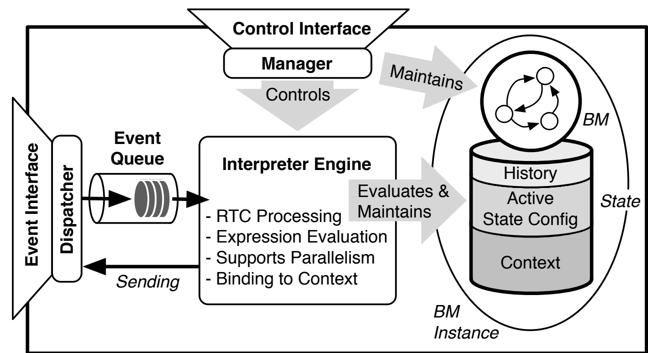

Figure 2: Architecture of the UML interpreter

For illustrative purposes we show only a single BM instance, where instances of several BMs could be used. The following text explains the architecture in more detail.

Event Interface Communication messages that pertain to the behaviour of the BM and that are exchanged with other entities are transmitted using this interface. The interface acts as a gateway between the environment and the locally active BMs and thus, has to to be compatible with the messaging protocols and formats employed.

Dispatcher The Event Interface establishes a connection to the environment, but this is not sufficient where internet-wide communication technologies need to be employed. The Dispatcher subsystem enables the use of routable event messages by taking care of event addressing. When accepting events for sending, the Dispatcher needs to differentiate between internal and external events and transmits the external ones via the Event Interface, whereas the internal ones are added to the interpreter's Event Queue.

Event Queue BMs are interpreted in a step-wise fashion, one event at a time. The event queue buffers external events until they can be processed by the Interpreter Engine.

Interpreter Engine The Interpreter Engine executes the logic for a stepwise evaluation of the BM, by following the Run-To-Completion (RTC) semantics described in [23, p. 565]. It evaluates conditional expressions, action statements and the context data associated with BM instances. It also provides support for runtime parallelism. To conduct BM interpretation using RTC semantics, the Interpreter Engine needs to maintain the state associated with each BM instance. This includes the context data, the active state configuration and the history state configuration.

Control Interface Management access to the BM interpreter is given through the Control Interface, allowing a management authority to control the interpreter. It is understood as a gateway between the environment and the interpreter, which transmits messages that pertain to the management of the interpreter, not the behaviour of the BMs.

Manager The Control Interface establishes a connection to the management authority, but does not implement the concrete logic needed to control the interpreter. This is done by the Manager subsystem, which provides lifecycle management capabilities for the interpreter, as well as handling BMs.

The UML interpreter has been implemented using the Java language (version 6). We employ the UML2 Ecore format provided by the EMF Model Development Tools (MDT) project [34] for the in-memory storage. An Ecore version of the ASP was created to easily identify interpretable BMs, as well as context data imports. The utilisation of Ecore is motivated by its popularity, and the same reason applies for the selection of UML in the first place: it is a standardised representation format, widely understood and supported by a large set of tools. We are employing native Java for the evaluation of expressions using the MVFLEX Expression Language (MVEL) [32]. The motivation behind this is the availability of an on-the-fly bytecode compiler for MVEL. The resulting code executes very fast – the authors of MVEL claim that it is the fastest expression evaluator on the Java platform[3]. The UML interpreter directly binds expression statements to the Java execution platform and a BM invokes Java objects that have been specified using the ASP ContextImport Stereotype.

4.2 Major Challenges

Implementing an UML interpreter is a challenging task and correctly integrating the many features of UML State Machines requires attention to detail regarding potential side-effects of its features. Engineering the BM interpreter implementation forced us to look deeper into the involved concepts and we will now highlight the major lessons learned while creating this proof-of-concept prototype.

Processing Compound Transitions.

In UML 2, a transition is not limited to consisting of a single edge leading from a single source state to a single target state, but can be made up of an arbitrary number of segments leading from a number of source states to a number of destination states, eventually branching and merging via choice and junction constructs. To determine the correct sets of entered and exited states, one needs to determine a single path (or several paths, should fork or join constructs be involved) through this graph of transition segments.

The UML Transition class exposes an attribute that assigns a kind to the transition, which can be either internal, local or external (see [23, p. 581]). Compound transitions consist of transition segments, where each one corresponds to the Transition class; thus, a compound transition could consist of segments with differing kinds. This is a conflict, as a compound transition can only be of a single kind (necessary for correct determination of exited and entered states).

Completion Events.

The UML standard introduces ε-transitions[4] by means of transitions that trigger on completion events. Completion events are dispatched automatically once a state has been entered and optional do-activities are completed, or once all substates of a compound state have completed [23, p. 574ff]. Understanding state completion as solely represented through events is an elegant approach for combining the state completion mechanisms and the mechanism that processes event triggers on transitions. Unfortunately, this approach is problematic in conjunction with guard conditions. Imagine a transition that is triggered on a completion event and labelled with a guard condition relying on context data. Once the source state of the transition is completely entered, a completion event is generated. At this point in time, the guard condition could evaluate to false: the transition would not fire and the completion event would be discarded. At a later point in time the context changes and the condition could now evaluate to true. As the completion event would already have been discarded, the transition could not be triggered anymore. Such a behaviour is perceived as contrary to the semantics intuitively attached to such a transition.

Determination of Exited States in Transitions Using the Choice Construct.

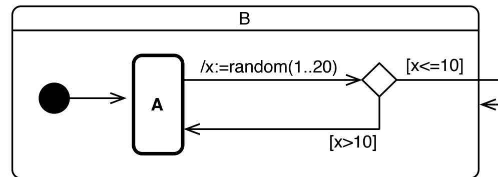

Figure 3: Demonstration of a problem with the UML choice construct

A particular problem with determining the proper set of exited states is encountered when combining choice constructs together with transitions that trigger actions. Figure 3 shows a partial model that demonstrates the issue. Imagine that the system is in state A and exits via the ε-transition, assigning a random value in the range 1 to 20 to the variable x. On the processing of the choice construct, a path is taken depending on the value of the variable x. If the transition is of an external kind, the exited states are either only A or A,B. The UML standard, on the one hand, dictates that states need to be exited prior to executing any transition actions: "Once a transition is enabled and is selected to fire, the following steps are carried out in order: The main source state is properly exited. [...] If a choice point is encountered, the guards following that choice point are evaluated dynamically and a path whose guards are true is selected. [...]" [23, p. 576] (ellipsis added). On the other hand it demands that "In a compound transition where multiple outgoing transitions emanate from a common *choice* point, the outgoing transition whose guard is true *at the time the choice point is reached*, will be taken." [23, p. 574] (emphasis in original text). The problem is that the individual states of a compound main source state might not be determinable before processing the choice – at this point the states would need to have already been exited. This issue has already been reported on the 7 December 2000 issues list; it is tracked under issue number 4110 at the OMG. By convention, our interpreter exits states up to the level of directly nested children of the parent vertex of the choice pseudostate: in the example this is only the state A.

[3] `http://mvel.codehaus.org/Performance+of+MVEL+2.0`

[4] An ε-transition (referred to as a "completion-transition" in the UML standard) is a transition without a specification of the trigger event in the transition label

Fork & Join.

We found that a major obstacle for implementing the UML features that relate to control flow handling is found in the non-exclusive usage of fork and join pseudostates for manipulating the control flow. For example, UML allows us to mix fork / join nodes with transitions that implicitly enter or exit parallel regions: i.e. two regions of a state might be entered through a fork pseudostate, while a third region is entered by means of an initial pseudostate. Such a situation is depicted in Figure 4.

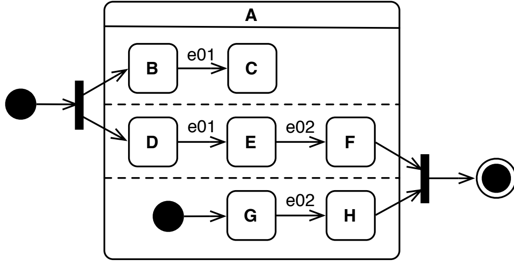

Figure 4: Implicit control flows

The shown BM can be processed from start to end, by triggering the two events e01 and e02. It can be seen that the initial fork pseudostate only specifies entry in the states **B** and **D**. State **G** is entered implicitly by an initial pseudostate. A similar issue occurs when the control flows join: once the BM enters the states **F** and **H**, the superstate **A** is left and the control flow with the active state **C** needs to be terminated implicitly. Due to such combinations of constructs, an interpreter is required to check each exited, orthogonal state for control flows that need to be implicitly terminated and to check each entered, orthogonal state for control flows that need to be implicitly created.

History and Regions.

As recognised by others [6], the semantics of the history pseudostate is not clearly defined with respect to orthogonal states. The UML standard states that "A composite state can have at most one [deep/shallow] history vertex." [23, p. 542][5]. If the respective composite state is also orthogonal, the history pseudostate needs to be contained in one of the regions. This is confusing, as the history functionality refers to the composite state, not the region. There are also no clear specifications on how control flows should be created upon restoration of a previously stored state.

5. PERFORMANCE ANALYSIS

For assessing the performance of the UML interpreter, we created a novel performance benchmark, consisting of ten scenarios for measuring key aspects of this technology. We then exercised the interpreter and collected performance measurements. As these values are somehow useless on their own, we also generated an implementation of a static execution mechanism in C++ using the Rational Rhapsody tool and the GNU Compiler Collection (GCC) tools. This allows us to compare the speed and memory consumption of the two approaches. We choose to use C++ as the language employed for the generated code, because we want to determine an optimal performance as a baseline for comparing the interpretation performance. GCC and C++ fit the bill due to their well known characteristic of being able to create fast runtime code. As the UML interpreter is implemented in Java, we are not only measuring the difference between baseline performance and interpreter performance, but our measurements also include the performance of the underlying Java Virtual Machine.

Unfortunately we need to forfeit a detailed description of the benchmark, as it would be far too substantial for this paper[6]. Instead, we will only discuss the results and add explanatory comments about the employed measurement scenarios were necessary. The benchmark uses measurements of the processing speed and the operational memory consumption. It was executed on a Apple MacBook Pro computer with a single Intel Core 2 Duo Processor running at 2.5 gigahertz and 4 GB of Random Access Memory (RAM). The employed software platform was the Sun Java HotSpot 64-Bit Server VM running on Mac OS X 10.6.3. For conducting speed measurements, we relied on using a specific `timestamp` action in the corresponding BMs. We determined the resolution of the employed system timer at approximate 1 microsecond (µs) and the employed overhead of the timestamping method incurred an average delay of 3.64 µs.

Transition matching.

Matching of a single ε-transition is fastest with a value of ∼25 µs[7] for the interpreter and ∼2.25 µs for the static code. For determination of more realistic values for transition matching, we need to take transition selection into account. Transition selection is used to select a single transition that will be activated, from a set of potential candidates. At first we are interested in selecting a transition only by matching an event trigger specification for a transition label. No guard condition or action specification is used. This scenario has an interesting result, which is why we will discuss it more in-depth. The benchmark uses the BM shown in Figure 5.

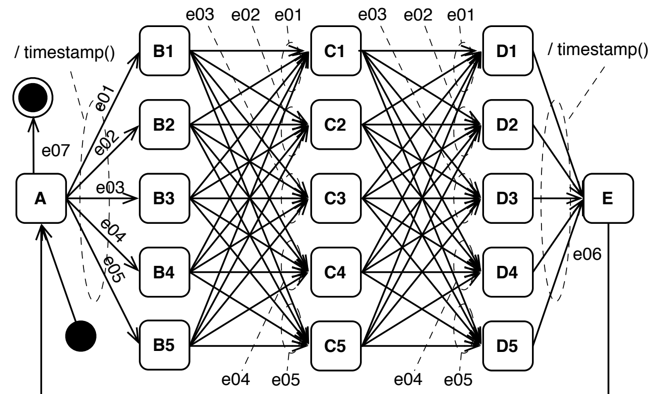

Figure 5: BM for measuring transition selection

Although the model might look complicated, the composition is simple: a control flow always starts at state **A** and continues along a path through states $\mathbf{B}_x$, $\mathbf{C}_y$, $\mathbf{D}_z$, finally arriving at state **E**. The values of x, y and z are determined

[5]Square braces added, these are two separate statements

[6]The complete benchmark will be published soon

[7]The symbol ∼ is shorthand for "approximately"

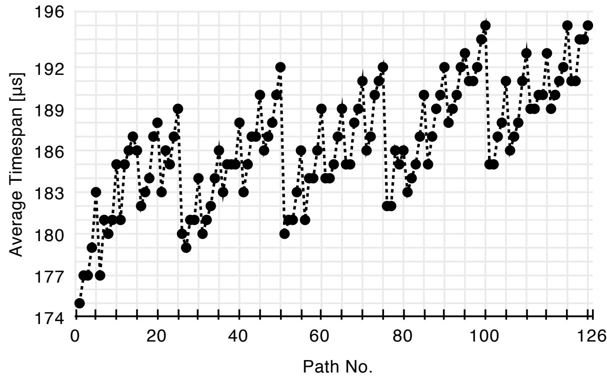

Figure 6: UML interpreter guard condition evaluation performance (per path)

by the received corresponding events e_x, e_y and e_z. They can range from 1 (triggered by `e01`) to 5 (triggered by `e05`). At state **E**, a reception of `e06` returns the flow to the beginning of the BM in state **A**, where the behaviour can also be ended through the reception of an `e07` event.

The benchmark scenario is executed by going through all possible paths in the model using events for all permutations of x, y and z. For example a sequence of `e01`, `e04`, `e03`, `e06` will lead to the following path of entered states: **A**, **B1**, **C4**, **D3**, **E** and **A**. There are 125 possible paths ($5 \times 5 \times 5$) and for each path a timestamp pair is recorded. The difference between these two values indicates the overall time that the interpreter spent on that path. We found that on average, it took the interpreter ∼186 µs to complete a single path. When plotting the average measured time per path, one can directly see the effect that the employed data structure for storing the BM has on the transition selection delay. Figure 6 shows the matching delays of the 125 different paths.

For understanding the diagram, one needs to be aware that each transition label needs to be checked against an incoming event. When using code generation, such a test is implemented with a constant runtime complexity, e.g. with a `switch` statement in the C++ programming language. This is not possible with an interpretation approach as the set of outgoing transitions of a state could change during runtime. Thus, the interpreter iterates over the set of outgoing transition and tests each label against the incoming event individually. Once a match is found, the iteration stops. This is visible in the plotted data: the measurement values have been connected by a dotted line, forming a curve with five large jags — these correspond to the transitions leaving state **A**. Each of the jags is again made up of five smaller spikes, reflecting the influence of the transitions that leave the **B** states. Lastly, each of the smaller spikes is made up of five data values, related to the selection of the transitions leaving the **C** states. For the generated code, no such patters are observable, it always executes with an average time between 22 and 25 µs, independent of the used path.

Processing of transitions that traverse nesting levels within the containment hierarchy of compound states is slower than the processing of simple transitions. This performance reduction depends on the number of exited and entered states and we determined an average factor between ∼2.5 and ∼3. The primary influence for the processing time difference is the number of nesting levels that need to be considered during processing of a transition (this can be either 1, 2 or 3 in the employed scenario employed).

Expression Evaluation.

Determination of the performance for the evaluation of expressions was conducted using seven different expressions that contain statements for boolean logic, arithmetic, method invocation and the sending of an event. The measurements have one thing in common: the first conducted measurement always took longest, when compared to all measurements. The worst case is found for the expression that invokes a method: the maximum was determined at ∼1259 µs, whereas the average was ∼66 µs, a difference of a factor of ∼19. We found, that these exceptional values are caused by one-time initialisation processes within the MVEL library. After initialisation, the expressions are evaluated with a lower variance: invocation time varies in the range of ∼15 µs, event sending varies within ∼20 µs and condition evaluation within a ∼30 µs range. The evaluation of guard conditions tends to be moderately slower for more complex conditions, ranging from ∼30.67 µs (one term with single operator) to ∼36.51 µs (multiple operators, four terms, nested parentheses). In the generated code all expressions are evaluated very quickly within the timeframe of ∼1 µs (this is the minimal timer resolution). This is independent of their complexity. The only exception is the sending of events, which takes an average of ∼3 µs.

Concurrency.

The scenario for measuring interpreter performance when handling concurrent control flows uses a number of compound states with fork constructs to split the original control flow into four separate ones. These individual control flows each pass through three states, before being combined to a single one through a sequence of join statements. A timestamp is taken before and after the process. The BM involves a number of states (21) and transitions (28) and the details do not matter here, it is sufficient to say that the BM took an average of ∼785.96 µs for processing. With code generation the concurrent control flows can be mapped to static structures (e.g. a single state that represents control flow 1 being in state A and control flow 2 being in state B), which saves the implementation from maintaining any additional runtime control flow information. The BM consequently executes in an average ∼3.92 µs. On a side note, we did also show that a similar optimisation is possible at runtime [15].

Configuration & Lifecycle Management Issues.

The interpreter performance for both instantiating and starting, as well as stopping and removing models, is steady. The starting time, which is the time it takes the interpreter to process the first transition of a BM is ∼1 Millisecond (ms), while the time for stopping and removing a BM takes only a fraction ($\sim\frac{1}{7}$) of that. This is due to the more complex processes needed when creating an initial BM instance. In comparison, the static code already contains a deployed BM once it has been loaded and there are only minimal initialisation processes (allocation of a single new object). The average time that it took the C++ implementation to process the first transition was ∼6.04 µs and a completion of the complete BM was executed on average in ∼19.06 µs, which

shows that exiting a BM takes the Rhapsody implementation more than twice as long as starting the model. Retrieval of the active state configuration was measured at an average of ~48 µs. By default, Rhapsody generates no methods for retrieval of the active state configuration. Instead there are macros provided that can be used to determine if a single state is active. We added code that executed each of the state macros of a certain BM[8] to determine the active states. The state configuration was then returned as a string containing the active state ids. On average, this code used ~8 µs to complete.

After the interpreter has been initialised the consumed memory stays on average at roughly the same values around 27–28 megabytes (MB) throughout the benchmark. The impact of the BM size on the overall memory consumption was determined using a set of BMs, each with a larger number of states than the previous one. We found that heap memory increases with ~1 kilobyte (KB) per additional state in a model. This is different for the statically generated code, where the BM size impacts the binary executable size, but not the dynamic memory consumption. We determined, that the binary executable size grows with ~0.1 KB per additional state. In general, memory consumption does not seem to be a relevant factor for BM interpretation, unless the interpreter is executed with a large number of models, with extremely big models or on a severely resource constrained hardware (e.g. an embedded system).

6. CONCLUSION

We were able to systematically determine the performance penalty incurred by such an approach. The results of the performance analysis confirm our assumptions; the performance of the interpretation approach is ~3 to ~460 times slower than generated code, depending on the utilised features of the BM and the employed technology for the runtime system. The average speed for the benchmark is ~20 times slower and the average consumed memory is ~60 – ~80 times more. For an interpreted language, such an overhead is considered normal, e.g. compare with Romer et al. who determine the overheads of a variety of general-purpose interpreted languages at a similar magnitude [24]. As a general rule of thumb it can be said that adding a level of interpretation slows down the execution time of a program by at least a factor of ten [10]. This is the cost to pay for the advanced runtime adaptation features provided by an interpretative approach.

Our research is mainly motivated by the requirements of AS engineering, namely the need for runtime adaptation of system behaviour in response to changes in a system's environment. In addition to enabling changes at runtime, the use of BM interpretation has a lot more advantages, which has been recently pointed out by J. den Haan et al [5]. BM interpretation enables a much faster turnaround time for applying system changes, as it does not require conducting code generation or build steps. This also increases the ease of deployment, because the same artefact used for designing a behaviour can also be used for deployment at the runtime system. Similar to a Virtual Machine, an interpreter supports the portability of applications by decoupling the concrete execution hardware from the application logic contained in the BM. Using interpreted BMs not only supports functional scalability, but also helps with scalability of resource utilisation, as one can create more processing capability by simply utilising additional interpreter instances for executing a given BM and where required, adapting employed BMs to account for the newly created instances. Our approach also enables the easy migration and persistency of running BMs by maintaining a separation of BM State and execution logic, as well as through the reactiveness of the employed BMs. BMs are only active in reaction to input events and always return to a quiescent state after RTC processing, which makes it easy to adapt them during these dormant phases. Furthermore, BM interpretation enhances the security of a platform, as the BM usually cannot access resources (e.g. the file system) directly, but only through the interpreter. The interpreter forms an abstraction layer on top of the execution system, essentially providing a Platform-as-a-Service (PaaS) infrastructure. Another interesting idea is the possibility of debugging BMs at runtime by using breakpoints at model level.

To this list of features we can add the support of a more understandable monitoring of application behaviour, which can be achieved by inspecting the active state configuration of BM instances. With appropriately designed models, the active state configuration of a BM instance can easily provide valuable runtime information for system management (e.g. a system being in a "fallback" state or in a state "waiting for end of calculation").

To summarise our contribution to the scientific community: We implemented a runtime interpreter for UML State Machines and assessed its performance using benchmark scenarios. We were able to show that the interpretation of BMs at runtime is a viable approach. The performance assessment shows an average overhead of a factor of ~20 when using a Java-based BM interpreter, compared with an implementation that relies on statically generated C++ code. We believe that the features offered by this approach outweigh its performance penalty, making it a good choice for systems that benefit from an adaptability of their behaviour at runtime.

6.1 Future Work

Building upon the presented foundation, we will pursuit several other paths. We are currently looking at a comparison between the performance of the UML Interpreter and generated Java code. This will allow us to determine the overhead of the approach specifically on the Java platform.

The combination of interpretation and dynamic compilation is also interesting. We are already employing just-in-time (JIT) compilation for the expression statements, but might extend this approach to BMs (or parts thereof), e.g. depending on the adaptation frequency.

While our work covers a basis for runtime adaptation of system behaviour, we did not investigate the adaptation process for BMs, itself. This is a highly relevant topic in current research [36, 37] and one of our ongoing research activities is concerned with the formalisation and consistent transformation of Statechart based BMs for runtime system adaptation.

7. REFERENCES

[1] Apache Software Foundation. Commons SCXML v0.9, December 2008. http://commons.apache.org/scxml.

[8]The one used for determination of the performance of concurrent control flows

[2] W. R. Ashby. *Design for a Brain*. Chapman and Hall, 1960. ISBN 0-412-20090-2.

[3] C. Ballagny, N. Hameurlain, and F. Barbier. MOCAS: A State-Based Component Model for Self-Adaptation. In *Proc. 3rd IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems*, pages 206–215, August 2009.

[4] M. L. Crane and J. Dingel. On the Semantics of UML State Machines: Categorization and Comparison. Technical Report 2005-501, School of Computing, Queen's University, Kingston, Ontario, Canada, 2005.

[5] J. den Haan. Model Driven Development: Code Generation or Model Interpretation? "Birds of a Feather" session at the Code Generation 2010 conference, June 2010.

[6] A. Derezińska and R. Pilitowski. Interpretation of History Pseudostates in Orthogonal States of UML State Machines. In *Proc. 7th Conf. on Next Generation Information Technologies and Systems*, pages 26—37, 2009.

[7] J. Ebert. Efficient Interpretation of State Charts. In *Proc. 9th Int. Symposium on Fundamentals of Computation Theory*, pages 212–221, December 1993.

[8] H. Fecher, M. Kyas, and J. Schönborn. Semantic Issues in UML 2.0 State Machines. Technical Report 0507, Christians-Albrecht-Universität Kiel, Institut für Informatik und Praktische Mathematik, June 2005.

[9] A. G. F. Filho and H. Liesenberg. Transforming Statecharts into Reactive Systems. In *Proc. 19th Conferencia Latinoamericana de Informatica*, pages 501–509, February 1993.

[10] P. Graham. The Hundred-Year Language. Keynote at PyCon DC, March 2003.

[11] O. Grossman and D. Harel. On the Algorithmics of Higraphs. Technical Report CS97-15, The Weizmann Institute of Science, December 1997.

[12] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, December 1987.

[13] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5), May 1988.

[14] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1996.

[15] A. Hinnerichs and E. Höfig. An Efficient Mechanism for Matching Multiple Patterns on XML Streams. In *Proc. of the IASTED Int. Conference on Software Engineering*, pages 164–170, February 2007.

[16] E. Höfig, P. H. Deussen, and H. Coşkun. Statechart Interpretation on Resource Constrained Platforms: a Performance Analysis. In *Proc. 4th Int. Workshop models@run.time*, page n.p., October 2009.

[17] IBM Corporation. An Architectural Blueprint for Autonomic Computing. White Paper, 4th edition, June 2006.

[18] A. Knapp and S. Merz. Model Checking and Code Generation for UML State Machines and Collaborations. In *Proc. 5th Workshop on Tools for System Design and Verification, Technical Report*, pages 59–64, 2002.

[19] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002. ISBN 0-201-74804-5.

[20] I. A. Niaz and J. Tanaka. Code Generation from UML Statecharts. In *Proc. 7th IASTED Int. Conf. on Software Engineering and Application*, pages 315–321, 2003.

[21] Nokia Qt Labs. SCXML Importer for the Qt State-Machine Framework, January 2010. `http://qt.gitorious.org/qt-labs/scxml`.

[22] North Carolina State University. The Benchmark Archives at CBL. `http://www.cbl.ncsu.edu:16080/benchmarks`.

[23] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2. OMG Specification, February 2002.

[24] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The Structure and Performance of Interpreters. *Proc. 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, October 1996.

[25] M. Samek. *Practical Statecharts in C/C++*. Newnes, 2nd edition, 2008. ISBN 0-75068-706-1.

[26] M. Sánchez, I. Barrero, J. Villalobos, and D. Deridder. An Execution Platform for Extensible Runtime Models. In *Proc. 3rd Int. Workshop on models@run.time*, pages 107–116, September 2008.

[27] S. Shlaer and S. J. Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon Press, 1991. ISBN 0-13-629940-7.

[28] M. Smirnow. Autonomic Communication: Research Agenda for a New Communication Paradigm. White Paper from Fraunhofer FOKUS, October 2004. `http://www.autonomic-communication.org/publications/doc/WP_v02.pdf`.

[29] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmark Suite. `http://www.spec.org/cpu2006`.

[30] G. Sunyé, A. L. Guennec, and J.-M. Jézéquel. Using UML Action Semantics for Model Execution and Transformation. *Information Systems*, 27(6):445–457, 2002.

[31] X. Than, H. Miao, and L. Liu. Formalizing the Semantics of UML Statecharts with Z. In *Proc. 4th Int. Conf. on Computer and Information Technology*, pages 1116– 1121, September 2004.

[32] The Codehaus. MVFLEX Expression Language, October 2009. `http://mvel.codehaus.org`.

[33] The DaCapo Project. DaCapo Benchmark Suite. `http://www.dacapobench.org`.

[34] The Eclipse Foundation. UML2 Implementation v3.0.1 from the Model Development Tools Project, August 2009. `http://www.eclipse.org/modeling/mdt`.

[35] R. P. Weicker. Dhrystone: a Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, October 1984.

[36] J. Zhang and B. H. Cheng. Model-Based Development of Dynamically Adaptive Software. In *Proc. 28th Int. Conf. on Software engineering*, pages 371–380, May 2006.

[37] J. Zhang and B. H. Cheng. Using Temporal Logic to Specify Adaptive Program Semantics. *Journal of Systems and Software*, 79(10):1361–1369, 2006.