Statechart Interpretation on Resource Constrained Platforms: a Performance Analysis

Edzard Höfig[1], Peter H. Deussen[1], and Hakan Coşkun[2] *

[1] Fraunhofer Institute for Open Communication Systems, Berlin, Germany
{edzard.hoefig|peter.deussen}@fokus.fraunhofer.de
[2] Faculty IV, Department of Design and Testing of Communication-Based Systems, Technical University of Berlin, Germany coskun@cs.tu-berlin.de

Abstract The statechart formalism allows for the specification of behaviour models of complex, reactive systems. It is employed in the embedded systems domain to specify and verify applications at design time. By enabling the interpretation of formalised behaviour models one earns the favourable abilities of application behaviour inspection, control, and substitution at runtime. One of the major arguments against such an approach concerns poor interpretation performance and high-resource overhead. We are answering this argument by showing that it is feasible to implement a statechart interpreter on a resource-limited platform. We define the utilised statechart formalism and use it as a base for implementing a resource-efficient interpreter on a 8bit microcontroller with 2 kByte RAM. Performance overhead of key aspects of the interpretation engine is evaluated using suitable behaviour models and by comparison with compiled code.

Key words: Statechart, Interpretation, Behaviour Model, Performance Analysis

1 Introduction

Professionals are modelling reactive systems using statecharts for the purpose of system analysis and quality assurance at system design time. Recently there is an interest in using such formalised behaviour models to control and describe system behaviour at runtime. For runtime evolution of software and communication protocols such an approach has potential advantages over the direct generation of system code. Take for example the dynamic re-configuration of embedded systems without a firmware "flashing" procedure or shutdown, the possibility to trace a system state with only minimal performance overhead, or the application of formal validation methods at runtime to assure that the system is in a valid state. Although these properties are of interest for researchers and practitioners

* The authors would like to acknowledge the European Commission for funding the Integrated Project EFIPSANS "Exposing the Features in IP version Six protocols that can be exploited/extended for the purposes of designing/building Autonomic Networks and Services" within the 7th IST Framework Program.

in the autonomic communication and networking field, the resource usage of the interpretation mechanisms is critisised. We often heard the argument that the performance overhead renders the approach unsuitable for resource-constrained platforms. Contrary to this, we believe that such an approach is feasible even for embedded hardware platforms. As we found no hard numbers on interpretation performance, we decided to conduct a study on the performance of a statechart interpreter on a resource-constrained platform. Our challenge is two-fold: Firstly, we created a proof-of-concept implementation of such an interpreter. Secondly, we quantified the performance, enabling us to give a well-grounded answer to the performance argument.
The paper is structured as follows. We describe related work in section 2, and give a formal definition of statecharts in section 3. In section 4 we detail the mapping of our definition to a runtime mechanism. Subsequently, section 5 describes the performance characteristics obtained by measuring the implementation. We conclude with a discussion of our findings in section 6.

2 Related Work

Statecharts were invented more than 20 years ago by D. Harel [1] and are in widespread use as part of UML2 state diagrams. An example for a statechart-based behaviour model of a simplified car door and passenger room light is depicted in Figure 1.

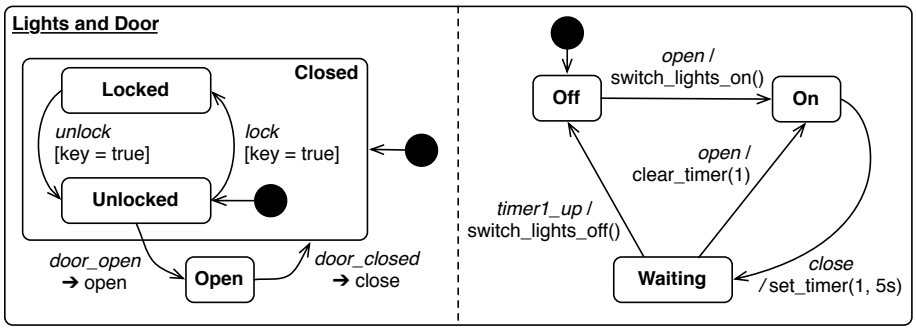

Figure 1. Example behaviour model "Lights & Door"

Employing statecharts for model checking and code generation is common practice [2,3] and the efficient interpretation of statecharts has also been researched by J. Ebert from a theoretical point of view [4]. First practical usage of interpreters for similar formalisms emerged in the last years out of the business process field [5]. An execution standard for statecharts is specified by the W3C under the name "State Chart XML"[6] with two implementations[3] available.

[3] A Java version from the Apache Software Foundation (`http://commons.apache.org/scxml`) and a C++ engine from QT Labs (`http://qt.gitorious.org/qt-labs/scxml`)

In our current work we use statecharts for network and system management [7] with the goal of equipping network routers with the ability to make autonomic decisions on an incoming packet stream by interpreting behaviour models [8]. We also have prior experience with optimisation of Extended Finite State Machines (EFSM) based automatons for analysing large XML data streams [9] and some of the optimisations that we are employing were discovered during previous work.

3 Formal Definition of Statecharts

Our formalism follows the definition specified in the annex of the original research paper [1]. We left out the "History Connector" definition, but apart from that we have a full-fledged statechart, including aggregate states and parallel components.

We define a *higraph* as a structure $H = \langle S, E, s_0, \theta, \sigma \rangle$, where S is a finite set of *symbolic states*, $E \subseteq S \times S$ is a set of *edges*, $\sigma : S \to 2^{2^S}$ is a *substate function*, and $s_0 \in S$ is a *root state*. We call each set of states $Q \in \sigma(s)$ a *component* of s. Distinguished components $Q_1, Q_2 \in \sigma(s)$ are called *parallel* to each other. A state $s \in S$ is called *atomic* if $\sigma(s) = \emptyset$ does hold, otherwise it is called *composed*. Moreover $\theta : \Theta \to S$ assigns a unique *start state* to each component of H, where $\Theta =_{\text{def}} \bigcup_{s \in S} \sigma(s)$ is the set of *components* of H. We assume that $\theta(Q) \in Q$ for each $Q \in \Theta$, i. e., the initial state of a component is a member of that component.

We stipulate a number of restrictions; to this end, let us define $\sigma^+(s)$ to be the smallest set (w. r. t. $\subseteq$) of symbolic states satisfying $\bigcup_{Q \in \sigma(s)} Q \subseteq \sigma^+(s)$ and $s' \in \sigma^+(s) \Rightarrow \bigcup_{Q \in \sigma(s')} Q \subseteq \sigma^+(s)$.

Then we assume that:

1. $Q \in \sigma(s) \Rightarrow Q \neq \varnothing$, i. e., substates of s are non-empty sets (note that this does not imply that $\sigma(s) = \varnothing$, i. e., we allow atomic states).
2. σ is *non-cyclic*, i. e., $s \notin \sigma^+(s)$ for all $s \in S$.
3. The sets in $\sigma(s)$ are pairwise disjoint, i. e., $Q_1, Q_2 \in \sigma(s) \wedge Q_1 \cap Q_1 \neq \varnothing \Rightarrow Q_1 = Q_2$, for all $s \in S$ such that $Q_1, Q_2 \in \sigma(s)$;
4. that the whole higraph has a tree-like structure, i.e., $\sigma^*(s_1) \cap \sigma^*(s_1) \neq \varnothing \Rightarrow s_1 \in \sigma^*(s_2) \vee s_2 \in \sigma^*(s_1)$.
5. Finally, for the root state s_0 we assume that $\sigma^*(s_0) = S$, and s_0 is the only state with this property.

For the sake of notational simplicity we moreover define $\sigma^{-1}(s) =_{\text{def}} s'$ whenever $s \in \sigma(s')$ (note that the expression $\sigma^{-1}(s)$ is undefined for $s = s_0$).

Next, we define a *statechart* as a structure $C = \langle H, V, D, I, \iota, \omega, \gamma, \alpha \rangle$, such that $H = \langle S, E, s_0, \theta, \sigma \rangle$, is a higraph called the *skeleton* of C. V is a finite set of *variables*, D is a set of *data*, I is a set of *events* including the *empty event* ϵ, $\iota, \omega : E \to I$ are mappings assigning a *triggering event* $\iota(e)$ and an *output event* $\omega(e)$ to each edge of H, respectively. Moreover $\gamma : E \to (V^D \to \{0, 1\})$ assigns a predicate to each edge of H. Here, V^D denotes the set of total mappings from V to D, i. e. all assignments of values from D to variables from V. Finally

$\alpha : E \to (V^D \to V^D)$ defines the effect of executing an edge $e \in E$ to an assignment $\rho \in V^D$.

A statechart describes a set of concurrent processes, where parallel processes are syntactically distinguished as substates $Q \in \sigma(s)$ of some high-level state $s \in S$. Hence we first need to define what a run-time state of statechart is. An *aggregated state* of a statechart C is a minimum (w. r. t. set inclusion) set $R \subseteq S$ such that

1. $s \in R \;\;\&\;\; s \neq s_0 \Rightarrow \sigma^{-1}(s) \in R$, i. e., if a state s is a member of an aggregated state, then its corresponding high-level state $\sigma^{-1}(s)$ is also;
2. $s \in R \wedge \sigma(s) \neq \varnothing \Rightarrow (\forall Q \in \sigma(s))|R \cap Q| = 1$, i. e., if s is a member of R, then R contains exactly one state from each component of s.

Note that by definition we have $s_0 \in R$ for each aggregated state R. Moreover, there is a uniquely defined initial aggregated state for each statechart C, namely the aggregated state R_0 with $\theta(Q) \in R_0$ for all $s \in R_0$ and $Q \in \sigma(s)$.

In order to fully describe the run-time behaviour of a statechart, we further need to take into account its current variable vector. Hence, *run-time states* are tuples of the form $\langle R, \rho \rangle$, where R is an aggregated state and $\rho \in V^D$ is a variable assignment. Let us denote the set of run-time states of C by Σ.

Now we are ready to define the behaviour of a statecharts in terms of transitions leading from one run-time state to another. To this end, we define a partial transition relation $\xrightarrow{a,b} \subseteq \Sigma \times \Sigma$ for each pair of events $a, b \in I$:

$$\begin{aligned}
&\langle R_1, \rho_1 \rangle \xrightarrow{a,b} \langle R_2, \rho_2 \rangle \\
&\Leftrightarrow_{\text{def}} (\exists e = \langle s_1, s_2 \rangle \in E)\big[s_1 \in R_1 \wedge s_2 \in R_2 \\
&\qquad\qquad \wedge\, \iota(e) = a \wedge \omega(e) = b \wedge \gamma(e)(\rho_1) = 1 \wedge \alpha(e)(\rho_1) = \rho_2 \\
&\qquad\qquad \wedge\, (\forall s \in R_2 \setminus R_1)(\forall Q \in \sigma(s))\big\{Q \cap R_1 = \varnothing \Rightarrow \theta(Q) \in R_2\big\}\big]
\end{aligned}$$

This means, a transition from a run-time state $\langle R_1, \rho_1 \rangle$ to another run-time $\langle R_2, \rho_2 \rangle$ if R_1 and R_2 contain symbolic states s_1 and s_2, respectively, connected by an edge $e = \langle e_1, e_2 \rangle$ labelled with the input event a and the output event b. Moreover, the predicate $\gamma(e)$ applied to ρ_1 yields true, and ρ_2 is the result of applying the $\alpha(e)$ to ρ_1. Finally, the last line in the formula above ensures that if a component Q is newly introduced into R_2 by the transition, then R_2 contains its start state. Using these definitions we can now discuss the statechart interpreter implementation.

4 Mapping of the Formalism to a Runtime Mechanism

We implemented the interpreter using the C programming language on an Arduino Duemilanove test board with a 16MHz ATmega328P microcontroller. There are 32 KByte Flash and 1 KByte EEPROM non-volatile memory available, as well as 2 KByte of volatile SRAM. We are using the most simple mapping that still allows to show a working approach. Introduction of more complex features would greatly improve the usability of the devised mechanism, but add nothing substantial in terms of evaluating the runtime performance overhead.

4.1 The Behaviour Model

We abstained from defining a syntax for behaviour models and work directly with an Abstract Syntax Tree (AST) in-memory representation, which we suppose can be generated from any suitable representation format (e.g., UML2 state diagrams, or SCXML). For each model the complete AST data is allocated as a single chunk of memory and the AST structure is constructed with single-byte references to this data. Prior to interpretation, an additional *executor* structure is allocated that holds input and output event queues, as well as data structures for processing parallel components, and a reference to s_0 as the initial starting point for execution.

4.2 States and Data Space

We restrict S to contain up to 256 symbols encoded by the numbers 0..255. Each state is represented by a data structure containing fields that allow to bidirectionally navigate the substate tree spanned by σ. For performance reasons we separate the state data structure into a substate set, a set of parallel components, and an additional reference to a superstate. Additionally, the structure contains a set of references to outgoing edges and a so-called *flag* byte used to indicate state properties, e.g., θ is implemented as a single bit in the flag byte. Sets are generally implemented as byte arrays with an additional field that holds set size. For aggregated states, and states containing parallel components, it is necessary to evaluate θ to identify the start state of contained component(s), and to additionally create data structures that allow for pseudo-concurrent processing of parallel components.

The variables V are limited to a maximum of 246 read- and writeable entries per behaviour model and 10 additional global entries shared between all executing models. Variables are referenced by the numerical values 0..255, where the values 0..9 refer to global values and 10..255 refer to local ones. The data set D is limited to 8 bit integer numbers. There is no type system. When data values are evaluated within boolean expressions, we follow C conventions for assigning logical values: 0 corresponds to a logical "false", other values are "true".

4.3 Edges and Event Processing

E is implemented as a set of data structures, which contain a reference to a destination state, the triggering event assignment ι, and the output event assignment ω. There can be a maximum of 256 edges. Events are numbered from 0..255 and identified by their numerical value – 0 is the special "empty" event symbol ε. The edge structure also contains references to a guard condition predicate γ and an action mapping α. Due to parallel processing of edges it is possible for multiple events to be received during a single step of a model. Events are buffered for input and output in ring-buffers, limited to 10 elements.
The guard condition predicates γ need to be evaluated to decide if an edge should

be traversed (hence the name "guard condition"). They are specified within the model AST and can be constructed from variable or constant references (notationally depicted using a $ sign), boolean operators (!, $\wedge$, $\vee$), and comparison operators ($=$, $<$, $>$, $\leq$, $\geq$). Evaluation precedence is implicitly given through the AST hierarchy.
The action bindings α are implemented as code that is statically bound to the runtime mechanism before the interpretation of a behaviour model commences. An action binding is a conventional function call with an arbitrary number of input and output parameters, and represents fixed capabilities of a device that are orchestrated using statechart logic. It is implemented by a structure holding a function pointer plus an ordered set of variable references. Parameters need to be de-referenced inside of the action function and can be used to read or write the variable value. A specific set of actions considers timers. We created three timers that can be set with a delay value using `set_timer(id,delay)` to deliver the specific events 8..10 once the delay time passes. Timers can be cleared using `clear_timer(id)` which suppresses dispatching of the timer event.
Explaining the processing algorithm goes beyond the scope of this paper. For an understanding of the intricacies refer to the paper by J. Ebert [4]. In a nutshell: For each input event all active components in a statechart are evaluated for triggered edges. If a triggered edge has a matching guard condition, the assigned action is executed and an output event send. The state(s) are then changed and another evaluation iteration is run with the next input event. These steps are repeated until all active components reach end states. It is worth mentioning that all ε edges are traversed before the next input event is taken from the input queue. Also, specific handling functionality needs to be executed on entering and exiting parallel states to maintain data structures for active components.

5 Performance Analysis

We found that the experimental platform has sufficient resources for the statechart interpreter code, which uses less than 8 KBytes of Flash memory. In this section we describe the evaluation results using the experimental platform.
Latency measurements have been conducted using in-line timestamps, the slight delay that has been introduced by this is negligible for the overall result. The employed timestamping mechanism has an accuracy of approx. 4 μs. Stack memory measurements were conducted by dumping the stack pointer during runtime. Performance of these routines is uncritical as such experiments only measured memory consumption, not latency.

5.1 Memory Overhead

To analyse stack performance, we exercised the behaviour model shown in Fig. 1. We used the following sequence to measure the normalised[4] stack allocation as shown in Fig. 2: $key \leftarrow false$, *door_open*, *door_close*, *lock*, *door_open*,

[4] Showing only the additional bytes consumed during interpretation of the model

door_close, *key* ← *true*, *lock*, *door_open*, *key* ← *false*, *unlock*, *door_open*, *key* ← *true*, *unlock*, *door_open*, *door_close*, *wait for the lights to turn off*. The interpreter executes a single *step* method to iteratively advance the statechart. This method uses 25 bytes stack when processing input events and 23 bytes when processing ε events. Peaks in the stack usage are due to evaluation of the *key* guard conditions on the edges between the *Locked* and *Unlocked* states.

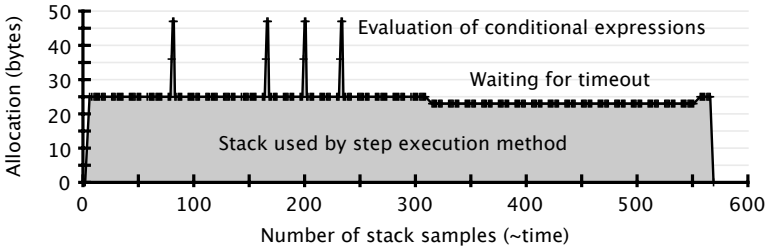

Figure 2. Stack usage during interpretation of Light & Door model

5.2 Conditional Expression Evaluation

The expression evaluator is implemented as a tree walker that recursively traverses a binary tree of statement tokens (variables, constants, and operators). We used three expressions to measure performance "$\$0 < 15$" (interpreted in 24 μs), "$(\$0 < 15) \wedge (\$1 = \$2)$" (56 μs), and "$((\$0 < 15) \wedge (\$1 = \$2)) \wedge ((30 > \$4) \vee (\$3 = \$5))$" (116 μs). This approach has a remarkable performance overhead: A hard-coded C version of any of these expressions executes in less than 4 μs. As seen in Fig. 2 the evaluation of conditional expressions is depicted as peaks in the stack usage. By sampling stack size during evaluation we found that the expression evaluator uses an additional 11 bytes per recursive iteration, e.g., Expression C uses a total of 44 bytes stack memory during evaluation.

5.3 Simple Edge Matching

We are measuring the time that our implementation needs to react with a single output event to a single input event using the traversal of a single edge. There can be more than one outgoing edge assigned to a single state, so we are also interested in the latency of the interpreter when processing multiple edges. We are using 30 behaviour models with an increasing number of edges for a single state. Each edge is triggered by a specific event 1..30 and sends a corresponding output event 101..130. Each model is then supplied with exactly one event, activating the edge that is triggered by the highest event. This is done to force the interpreter into exhibiting worst-case behaviour (it checks each edge before finding the edge that matches). The results, along with an illustration of the experimental models, can be seen in Fig. 3. To put the measurements into perspective, we also added the time that a conventional "switch" statement needs to deliver the same result.

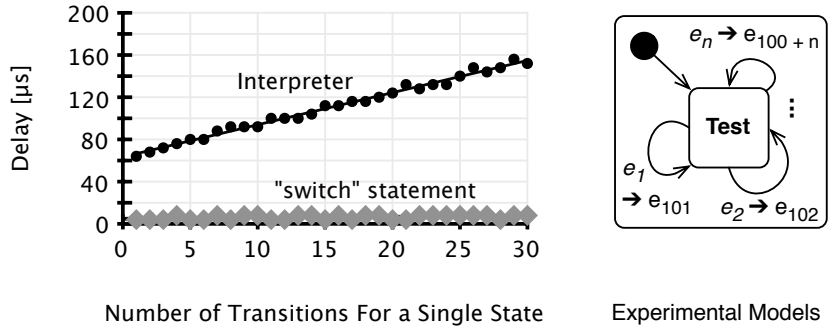

Figure 3. Delay of edge matching processes

The latency for a simple edge transition is approx. 64 μs, which includes event processing, timer handling, edge selection, and edge execution. It is approximately a factor 10 slower than a conventional switch statement which executes at around 6 μs. Latency increases linearly with approx. 3 μs for each edge up to 152 μs for 30 edges. The "switch" statement has a constant delay independent of the given event. The reason for the linear increase is the need to check each of the edges for a possible match.
The usage of dynamic action bindings instead of static function calls also has an impact on the latency of action execution due to the way parameters are passed to function code. We created models that trigger an action using a single edge from a single state and altered the number of parameters (0..10) passed to the action. The additional delay introduced amounts to an average of approx. 3 μs per additional parameter. For conventional function calls we believe that an additional delay exists as well, but we found that the measured latency differences are within the precision range of the employed timing mechanism for the number of arguments studied (delay differs < 4 μs).

5.4 Processing of Aggregates and Parallel Components

The two major features that differentiate statecharts from EFSM are aggregation and the ability to specify parallel components. To measure performance of aggregation handling we used a series of models with an increasing number of nested states (from a single state to an aggregate with a nesting depth of 30) where the most deeply nested state had an edge that matched on a given input event. The parallel component processing was analysed using 30 models which contained a superstate with an increasing number of parallel components, each triggering on the same input event. The results are displayed along with the experimental models in Fig. 4. We found it necessary to differentiate between the first input event and subsequent events[5] processed in the same state. This is due to additional functionality executed when entering an aggregated state or a state that contains parallel components. Fig. 4(A) shows an average delay of 12.5

[5] in the diagram labelled as 2^{nd} event, representative for all subsequent events

μs per additional nested state for an event that triggers entering the aggregate. Once the aggregate has been entered, the delay for processing subsequent events is independent of the nesting level. This is different for parallel components, as shown in Fig. 4(B). Entering a state with parallel components has an average latency of approx. 52 μs per parallel component. There is an average overhead of approx. 26 μs per active component for each subsequent event. To compare the latency overhead with conventional constructs, we also show the delay of a "for-loop" sequentially processing the input event. In this case, the overhead is at approx. 2 μs per additional iteration.

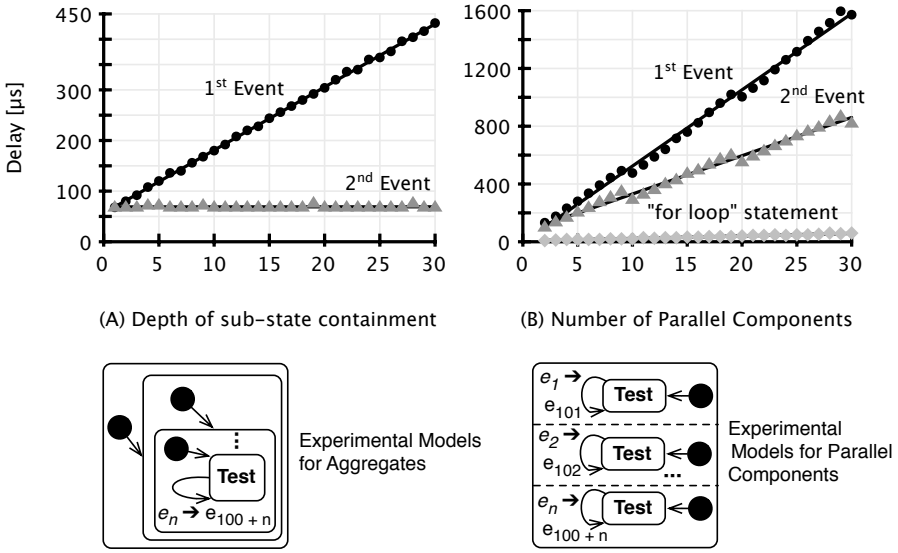

Figure 4. Event delay characteristics for processing of aggregated states (A) and parallel components (B) with the employed experimental models

6 Conclusion

The results confirm our initial assumption: It is possible to implement a statechart interpreter on a severely resource-constrained platform. We had no problem fitting the interpreter into non-volatile memory, though the available heap memory establishes clear constraints on the complexity of the behaviour models. Heap memory was large enough to hold any of the experimental models we applied for performance assessment, but we found that models with about 100 states are the limit. Stack memory is unlikely to be exhausted: Models would need to use a very deep conditional expression token tree.
On the performance side, we found that the interpreter clearly adds a processing overhead. In the best case execution latency is about a factor 10 longer than with compiled code. Performance depends largely on the structure of the interpreted

models, main factors are: the number of edges leaving a state, the nesting depth for aggregates, the number of parallel components, and the usage of guard conditions. Also, the ratio between the time the interpreter spends in action functions and the time spent in statechart interpretation plays an important role: If action functions are sufficiently complex, the overhead caused by the statechart engine is much smaller. On the other hand, if system behaviour is completely modelled using a statechart, the interpretation overhead becomes very large.

The interpreter performance can still be improved, mainly by using a better expression evaluator, but also by optimisation of the edge evaluation code (e.g., grouping triggering events, combining guard conditions). Even with ingenious optimisations, some overhead cannot be purged: In the worst case, any statechart interpreter needs to evaluate all outgoing edges for a single state, including the outgoing edges of parent states, and there will always be an overhead for event processing and handling aggregates, as well as parallel components. Therefore we conclude that our approach is adequate for reactive systems, which are idle most of the time. It does not seem to be suitable for systems that need the fastest possible reaction time due to the introduced interpretation delay, which can easily amount to 1 ms. Such a value is unacceptable for most real-time applications. Regarding high-throughput systems, successful applications should be possible but will depend on the underlying platform performance and the utilised behaviour model complexity.

References

1. Harel, D.: On visual formalisms. Communications of the ACM **31**(5) (May 1988)
2. Gnesi, S., Mazzanti, F.: On the fly model checking of communicating uml state machines. ACIS Int. Conf. on Software Engineering Research, Management and Applications (2004) 331–338
3. Raghunathan, B., Hartrum, T.: The automated transformation of statecharts from a formal specification to object-oriented software. 48th Midwest Symposium on Circuits and Systems (2005) 319– 322
4. Ebert, J.: Efficient interpretation of state charts. Fundamentals of Computation Theory **710** (Jan 1993) 212–221
5. Sánchez, M., Barrero, I., Villalobos, J., Deridder, D.: An execution platform for extensible runtime models. Proc. models@run.time workshop (2008) 107–116
6. W3C: State Chart XML (SCXML): State machine notation for control abstraction. http://www.w3.org/TR/scxml/
7. Höfig, E., P.H.Deussen: Document-based network and system management. Proc. 2nd International Conference on Autonomic Computing and Communication Systems (Jun 2008)
8. Höfig, E., Coskun, H.: Intrinsic monitoring using behaviour models in ipv6 networks. to be presented at the IEEE Modelling Autonomic Communication Environments (MACE) Workshop (October 2009)
9. Hinnerichs, A., Höfig, E.: An efficient mechanism for matching multiple patterns on xml streams. Proc. IASTED Int. Conf. on Software Engineering (2007) 164–170