Visualization of the execution of Real-Time Statecharts

Matthias Tichy and Margarete Kudak
Software Engineering Group
University Of Paderborn
Warburgerstr. 100
33095 Paderborn
[mtt|kudak]@uni-paderborn.de

ABSTRACT
Embedded software systems are used in nearly all of today's industrial products. Statecharts are used for the specification of the reactive behavior of those embedded systems. Since embedded systems have typically no rich user interface to display the current status of the system or even to display debug messages, another way to monitor the execution of the embedded system has to be used. In this paper we describe an extension of the Fujaba Tool Suite to support on-/off-line monitoring of the execution of Statecharts.

Keywords
Statecharts, embedded systems, real-time systems, UML, Fujaba, monitoring, execution traces.

1. INTRODUCTION
Embedded software systems are a huge factor in today's electronics or industrial products. Since a nearly failure-free operation of these embedded software systems is of utmost importance, high-level languages for the design and implementation of the embedded software system are employed. UML Statecharts are one of those high-level languages. They are used to specify the discrete behavior of software systems. Real-Time Statecharts [3] are a variant of UML Statecharts especially geared to the specification of hard real-time systems. Schedulability analysis and Java RT code synthesis are offered to omit error-prone manual implementation of the specification.

In case of a failure the developer of an embedded system wants to know what exactly has gone wrong in the system. Since embedded systems typically have no rich user interface to display its current state or to display debug messages, other means to view the behavioral activities, which lead to the failure, are required.

We propose a monitoring and visualization framework for UML and Real-Time Statecharts in the Fujaba Tool Suite [1]. This framework allows the developer to monitor the execution of the Statecharts. The monitoring data of the executed Statecharts are visualized using UML Sequence Diagrams and Real-Time Statecharts, with special markups. The visualization can either be used in on-line mode visualizing the current behavior of the system or in off-line mode visualizing older monitoring data. In this paper we focus on real-time embedded systems and Real-Time Statecharts. Nevertheless the approach is applicable to non real-time systems as well.

In the next section we give an overview of our approach. In Sections 3-5 we explain in more detail the different steps of our approach. We conclude in Section 6, describe the current state of work, and present some future research directions.

2. OVERVIEW

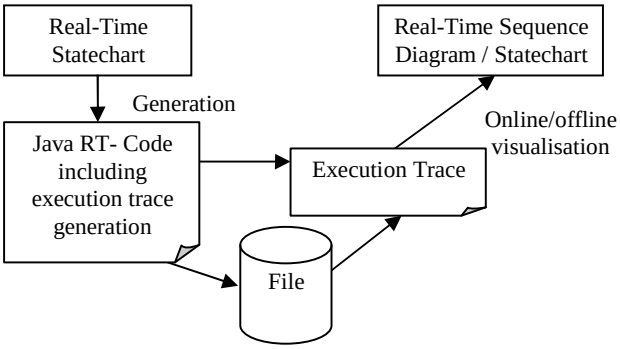

Figure 1. Framework architecture

Our proposed monitoring and visualization framework shown in Figure 1 consists of mainly 3 parts. We have to support the generation of execution traces to gather the data which contain the behavior of the executed Statechart. For each monitored Statechart an execution trace will be created during execution. In Section 3 we show in detail the contents of an execution trace and the different alternatives, which can be used to generate the execution trace.

Since typically more than one Statechart will be visualized, the execution traces of the different Statecharts must be merged prior to visualization. In Section 4 we give a brief overview of how we plan to merge those execution traces.

Finally, the merged execution traces are visualized by means of UML Sequence Diagrams and Real-Time Statecharts. UML Sequence Diagrams show the developer the message flow between the Statecharts as well as time annotations. Additionally, we added a graphical notion to show the current state of an executed Statechart. In the Statechart oriented view of the execution, the developer can see all Statecharts and their current state at a given time. Here, the developer may see the cause for a wrongly fired transition. In Section 5 we give more details and some examples diagrams.

The above mentioned visualization of the Statecharts's execution can be done on-line respective off-line. That means, the visualization can display either past executions by reading the execution traces from a file (off-line) or display the behavior of currently executed Statecharts (on-line). If a monitored Statechart

is changing its state very fast and very often, the visualization may lag in on-line mode.

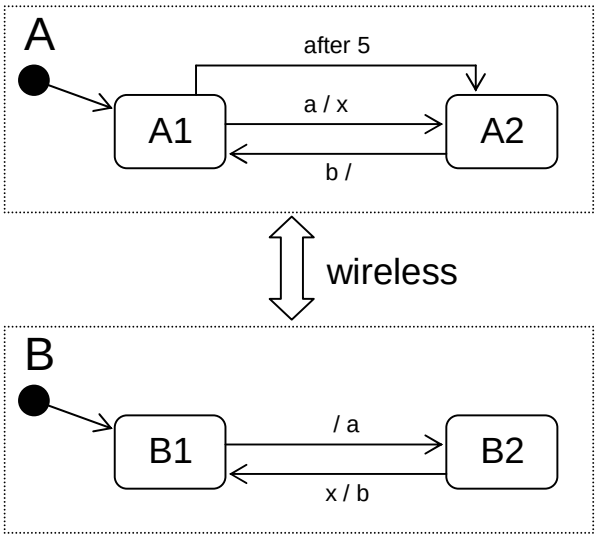

Figure 2. Example

In the example above (Figure 2) two Real-Time Statecharts are depicted which communicate via a wireless lan. In this small example it may be obvious (but not for larger ones), that both Statecharts may reach a deadlock. Initially Real-Time Statechart A is in state A1 and is waiting for the message a to change into state A2. Being in A1, A can also fire the transition "after(5)" and then go into state A2. Real-Time Statechart B changes from state B1 to B2 and sends the message a to A. Sending this message by wireless lan can take longer than 5 ms, so A may go into state A2 without having received the message from B. The consequence of this is, that Real-Time Statechart A is in A2 and is waiting for the message b, while Real-Time Statechart B is in B2 waiting for the message x to send the message b.

3. GENERATION OF EXECUTION TRACES

A trace of the execution of a Statechart consists of several execution activities. Each execution activity describes an activity of the Statechart, for example firing a transition.

Generation of such execution traces during runtime is time-critical, since it influences the temporal behavior of the Statechart. For hard real-time systems the goal is to minimize the temporal change.

For the generation of the execution traces several alternatives are available. At first the Java Debug Interface (JDI) [7] can be used to monitor the executed system. This would be a fairly easy solution, since no change of the monitored software system is required. Unfortunately, the runtime costs of JDI are way too high. The measurements in [6] indicate an increase of runtime by factors 100-300,000. Thus, JDI is completely out of the question.

A second solution is to instrument the compiled software. The Byte Code Engineering Library (BCEL) [8] can be used to change Java software after compilation. The negative effects of JDI can be avoided by using instrumentation. Though, it is difficult to find the correct code position where to add the data generation code. Since we are generating the source code for the Statecharts anyway, we may add the data generation code for the execution activities as well. Additionally, then the code for generation of the execution activities can be taken explicitly into account during scheduling analysis.

Additionally, we need to decouple the generation of the execution trace from everything else (file write, network write, etc.). We use a queuing approach. Each execution activity will only be written into a queue in the time-critical part of the system. During idle time, the queue entries will be read and further processed for writing to disk or network. If the queue becomes to large for the monitored system, the developer must take appropriate actions.

For an individual execution activity it is important to know what states, transitions and properties a Real-Time Statechart has. In the execution trace four activities are mainly listed: the change of a state, when a transition fires and the incoming and outgoing messages between Real-Time Statecharts, as can be seen in Figure 2. Additionally, an execution activity may include information about the different clocks used in the Real-Time Statechart and may include data about local variables which are used in transition guards.

4. MERGING OF DIFFERENT EXECUTION TRACES

A typical embedded system is comprised of more than one active system. These active systems are communicating with each other. For our approach we do not only need to generate and visualize the execution of just one Statechart. We have to display all concurrent Statecharts of the embedded system and their communication. Thus, the developer has the ability to see the interaction of the Statecharts. This interaction is a typical cause of complex errors in software.

To achieve this goal, we need to merge the execution traces of all involved Statecharts for a coherent view of the system. Embedded systems are often deployed on a distributed system and therefore are executed on different computers with different local clocks. Thus, clock differences and clock drift must be taken into account for the merging of the execution traces.

The differences between local clocks can be tolerated by the use of relative time stamps based on the known clock differences. Dealing with clock drifts is more difficult. If the clock drift is small and the absolute drift over the monitoring period is not too large (fraction of seconds), it may be possible to ignore it. Otherwise the approach of [4] has to be used. This approach uses causal relationships, e.g. between incoming and outgoing messages, to merge the execution activities in a correct order. In contrast to [4] we know the originating Statecharts and thus establishing a causal relationship is a lot easier.

5. VISUALIZATION

After merging the execution traces according to the last section, we can visualize them in two different ways.

5.1 Sequence Diagrams

UML Sequence Diagrams are used to display the communication between the participating Statecharts. Each Statechart is represented by an active object in the Sequence Diagram. The messages between the Statecharts are displayed as arrows between the lifelines. As an addition, the current state of the Statechart is shown as a special graphical object on the lifeline after a state change (see Figure 3). Time information has been added to the Sequence Diagrams to show the timing behavior. In real-time systems the timing behavior can often be the cause of problems,

which are difficult to find. For the sake of a clearer presentation only some time annotations have been added to the figure.

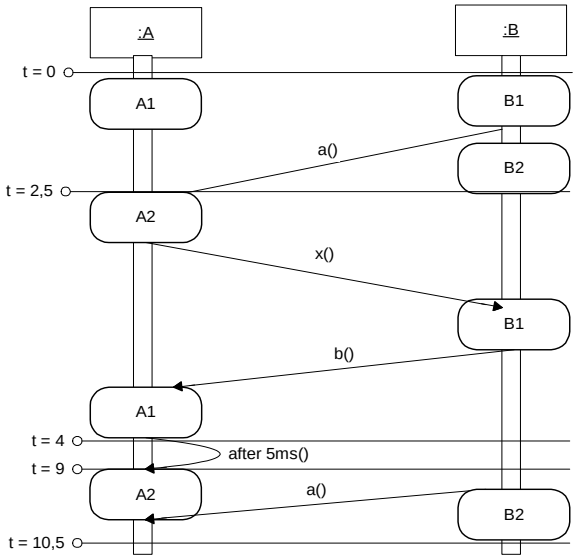

Figure 3. Sequence Diagram

The Sequence Diagram of Figure 3 shows the visualization of the Real-Time Statechart (Figure 2). Using this form of visualization the user can easily see, that Real-Time Statechart A fires the transition "after 5" and only receives message a from B afterwards. At this point of time the deadlock is reached, because A is waiting for message b while B is waiting for message x to send message b to A.

5.2 Statechart snapshots

For each point on the timeline of the above described Sequence Diagram a snapshot of all Statecharts can be displayed. Such a snapshot of a Statechart, displays the Statechart and an additional markup of its current state. The Statechart view may be easier for the developer to realize what the cause of a problem is. For example in this view the developer can see why a certain transition did not fire as expected and what the difference from expected behavior is. In Figure 4 the developer can see that the after 5 transition fired, since the message a has not been received.

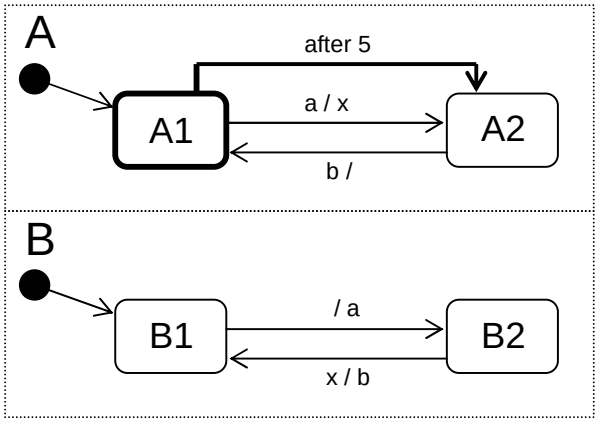

Figure 4. Real-Time Statechart with special markups

5.3 Navigation

The user has different possibilities to navigate in the visualization. Using a timeline, he can slow up and speed up the visualization. If for example states are changing very quickly (fraction of seconds), it is not possible to see exactly what happened. Due to this, it makes sense to reduce the speed of the visualization. Another navigation possibility is to pause and resume the execution. Additionally, the user is able to specify a particular point of time and to start the visualization from this timestamp.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a framework for the visualization of the behavior of embedded systems. This framework provides visualization in form of UML Sequence Diagrams and Real-Time Statecharts. The systems can be monitored off-line, i.e. a former execution is displayed. Additionally the systems can be monitored during runtime (on-line). Currently, we are at the implementation stage of the proposed framework.

We focus on real-time systems specified by Real-Time Statecharts. Nevertheless, the approach can be used to monitor non real-time Statecharts as well. The Fujaba Tool Suite includes support for StoryCharts [5]. StoryCharts are an extension of UML Statecharts which may include StoryPattern as do-activity. Our proposed approach may be extended to show the application of the StoryPattern as well (in [2] a related approach for the debugging of StoryDiagrams is described). Statistical evaluation of a number of execution traces may be used to discover potential faults of the system, which otherwise may be unnoticed.

7. REFERENCES

[1] The Fujaba Tool Suite. http://www.fujaba.de, September 2003.

[2] Leif Geiger, Albert Zündorf. Graph Based Debugging with Fujaba. In Proc. of the Workshop on Graph Based Tools, International Conference on Graph Transformations, Barcelona, Spain, October 6 - 12 2002.

[3] Holger Giese and Sven Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Computer Science Department, University of Paderborn, June 2003.

[4] C.E. Hrischuk and C.M. Woodside. Logical Clock Requirements for Reverse Engineering Scenarios from a Distributed System. IEEE Transactions on Software Engineering, 28(4):321-339, April 2002.

[5] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, pp. 241-251, ACM Press, 2000.

[6] Katharina Mehner. Zur Performanz der Überwachung von Methodenaufrufen mit der Java Platform Debugger Architecture (JPDA). Java Spektrum, Ausgabe Nov./Dez. 2003 (in German).

[7] Sun Microsystems. Java Platform Debugger and Java Debug Interface. http://java.sun.com/products/jpda, September 2003.

[8] The Jakarta Project. Byte Code Engineering Library. http://jakarta.apache.org/bcel/, September 2003.