SOFTWARE SYNTHESIS FROM STATECHART MODELS FOR REAL TIME SYSTEMS

Claude Ackad[1]

Formal description techniques like Harel's statecharts offer a great potential to facilitate the design of complex real time systems. The systems are modeled and simulated at a high level. Automatic translation from the abstract models into implementations significantly reduces overall development time. This paper contributes to the optimized code generation from statechart models. The worst-case execution time of the generated code is reduced.

1. Introduction

Real Time (RT) systems gain increasing importance through the widespread use of embedded systems like automative systems. Computations must fulfill RT requirements such as worst case execution times (WCET), i. e., the maximal required computation time with arbitrary input to the system.

Formal description techniques like SDL [Itut93] and statecharts [Hare87] allow the formal and unambiguous description of systems at a high level enabling the simulation of the (informal) system specification.

Statecharts employ parallel, hierarchical state diagrams. Concurrent processing as well as high-level interrupts can be specified appropriately. The semantics are formally defined supporting simulation and analysis of dynamic properties.

To avoid an error-prone manual translation into an implementation and to save design time, automatic code generation is seeked bridging the gap between abstract model and implementation. Although some code generators for statechart models do exist, yet no generator takes into account optimizing the code for small execution times. This paper contributes to the optimized translation of statechart models into implementations w. r. t. a small WCET.

---

1. Technische Universität Braunschweig, Abteilung Entwurf integrierter Schaltungen, Gaussstrasse 11, 38106 Braunschweig, Germany, email: ackad@eis.cs.tu-bs.de

F. J. Rammig (ed.), *Distributed and Parallel Embedded Systems*

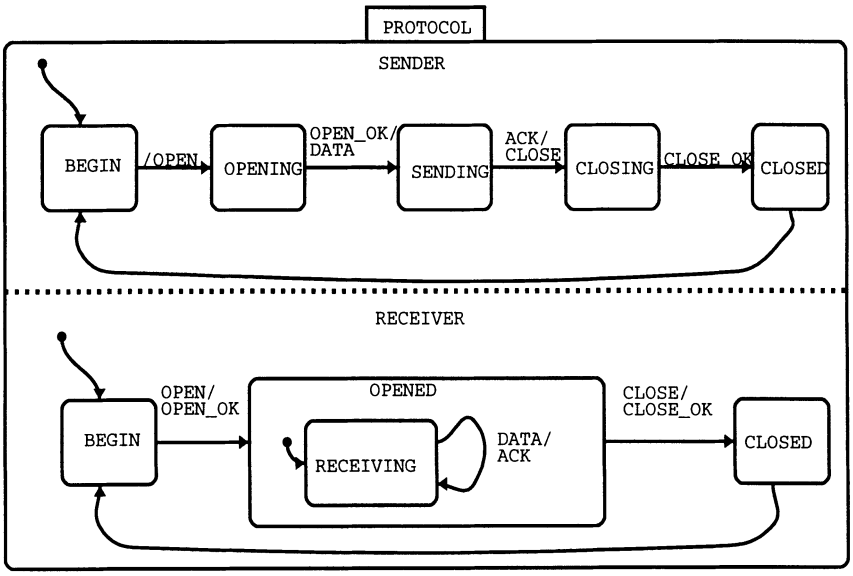

Figure 1: A simplified Transfer Protocol

2. Statecharts as a Modeling Language for RT Systems

Figure 1 shows a statechart example of a simplified transfer protocol consisting of the two parallel finite state machines (FSM) SENDER and RECEIVER. In a typical statechart, concurrent FSMs exchange data through normal variables such as integers or using abstract *events* which remain valid for one step only. In the remainder, we use "variable" for normal variables as well as events.

All variables are global to the whole model. Each FSM may read or write any variable. Multiple write accesses to the same variable in one step are forbidden and considered to be a model error.

In the example, the sender opens a connection by generating the event OPEN, the receiver acknowledges with OPEN_OK, enters the hierarchical state OPENED and its sub-state RECEIVING. Data are transmitted generating DATA. The transmission is terminated by sending CLOSE in which case a transition occurs from OPENED to CLOSED (high-level interrupt).

Parallel activities, inherent in most real world systems, can be modeled appropriately at different levels in the statechart model. Parallel FSMs may be started dynamically. Similar behavior of states can be expressed using high-level transitions leading to more comprehensible models.

3. Design Flow of RT System Design using Statecharts

Figure 2 shows the design flow using statecharts. The abstract (textual) specification for the system often contains contradictions and ambiguities. It cannot be simulated because the specification is not formal, thus leaving errors unrevealed.

The specification is modeled using statecharts. This model is on the one hand "near" the specification by means of the abstract nature of the statecharts and on the other hand simulateable because the statechart semantic is formal and unambiguous. Specification errors are discovered quickly and different design architectures are explored in short time at this abstract level.

After validation of the model, an implementation must be developed. A manual translation is error-prone as well as time consuming violating a short time-to-market. Therefore, an automatic transformation into an implementation is required.

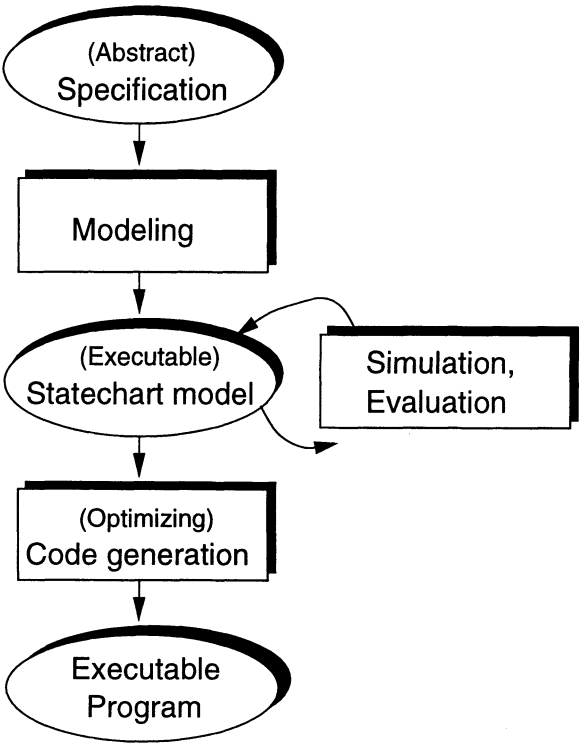

Figure 2: Design Flow using Statecharts

4. Software Synthesis of Statechart Models

After designing the statechart model, an *implementation* is derived automatically which essentially exhibits the same functionality as the statechart.

The real time systems we consider must fulfill constraints on the WCET, e. g. for scheduling calculations. Each piece of implementation code takes some amount of *execution time* depending on the actual state and the input. The implementation must be structured in a way to provide for an automatic execution time analysis to determine the WCET of the program. Therefore, recursive function calls and unbounded loops are prohibited.

The WCET of the implementation should be as small as possible, that is, the implementation should be optimized w. r. t. a small WCET.

Of course, there are many valid implementations with different WCETs for a given statechart. Our contribution lies in the optimized code generation from statechart models into implementations for single processor target platforms. We consider only a special class of implementations derived from *parallel execution graphs* as explained below.

The execution of statecharts is step-based. In each step, all parallel FSMs synchronously execute transitions, generating events or changing variables.

Statechart models describe reactive systems which never terminate. Therefore, the implementation is a *cyclic executive* containing a loop construct at the top level. The time for the execution of the whole system is infinite. Each loop cycle corresponds to a statechart step. The longest execution time for one cycle is the worst-case execution time (WCET) of the implementation, which is the longest processing time of a response to a stimulus.

4.1. Influence of the Statechart Semantics on the Execution Time

We focus on implementations to be executed on a single processor. Starting with a statechart model, an executable program must be generated which has the statechart functionality and a small runtime system (RTS) to keep the statechart semantics.

In each statechart step, many parallel FSMs may switch; these parallel activities have to be sequentialized. The more FSMs are active in parallel the longer the execution time. It is not determined in which order parallel FSMs are processed in the implementation. This leads to some degrees of freedom (DOF), which can be exploited to improve the WCET.

Like in most formal techniques, the formal semantics result in overhead regarding execution speed and/or code size of the implementation. The statechart semantics require that changes of variables occur at the beginning of the next step. It is essential that the implementation exhibits the same behavior as the statechart model, thus variable changes must be buffered until the end of the current step (*variable buffering*). This buffering consumes execution time delaying writes after all reads to the same variable are done. Often, no buffering is needed; the execution time may be improved in these cases.

Events are valid for only one step; after generating the event, it is cleared in the following step. Because non-trivial statecharts contain quite a few events, an efficient RTS event handling is crucial.

4.2. Existing Code Generators

Statemate [Ilog97] is a toolkit containing a statechart editor, a simulator as well as a code generator. Unfortunately, the generated code is not optimized for a small WCET; furthermore, the implementation contains unbounded loops and calls to library functions like *malloc*, such that execution time is unpredictable.

Procors [Spre96] is a code generator developed at BMW AG, Germany. Although the code is optimized w. r. t. a small WCET, the generator imposes severe restrictions on the allowed statechart models. For example, only one event is consumed by the model at each step. Furthermore, the statechart semantics are altered in that variable changes occur immediately, even "inside" a step, resulting in behavioral differences between the statechart model and its implementation.

5. SCOT[2] Project Description

Because of the lack of an optimizing code generator for statecharts, we launched the project SCOT. The main project objective is the efficient translation of statechart models into implementations. The generated machine code should exhibit a small and computable WCET, which is to be optimized for a single 68000 processor system.

Statemate is our statechart modeling tool. It was successfully applied to the rapid prototyping of airplanes [RJJC95], automobiles as well as specification of home automation systems [ScSc96].

5.1. Transformation Flow Using SCOT

Figure 3 shows the transformation flow using SCOT. After designing the statechart model with Statemate, we transform it into an intermediate format, a *parallel execution graph* (PEG). The PEG is an acyclic directed graph, which is an abstract scheme for the final implementation. The rather complex statechart semantics (evaluation of transition trigger expressions, calculation of priorities) are significantly reduced to simple branches depending on variable values, and assignments. The time required for one step can be estimated more accurately because of the simpler constructs.

The PEG execution contains some degrees of freedom (DOF), for example, the FSM execution order, which are represented using *parallel blocks*; segments contained in parallel blocks can be executed in an *arbitrary* order. In an iterative optimization cycle, this freedom is eliminated.

The WCET estimator analyzes each node in the graph for its type, data-dependency to other nodes, number and type of operands etc., and estimates the execution time depending on timing information of the target platform. To all nodes containing variable read or write references, an overhead for the double buffer handling is added. This overhead may be reduced later upon fixing the execution order.

The longest path through the PEG is determined and optimized yielding a new PEG with a smaller WCET and less DOF. This step is repeated until no more optimizations are possible.

---

2. Statechart Compilation for Optimized worst-case execution Time

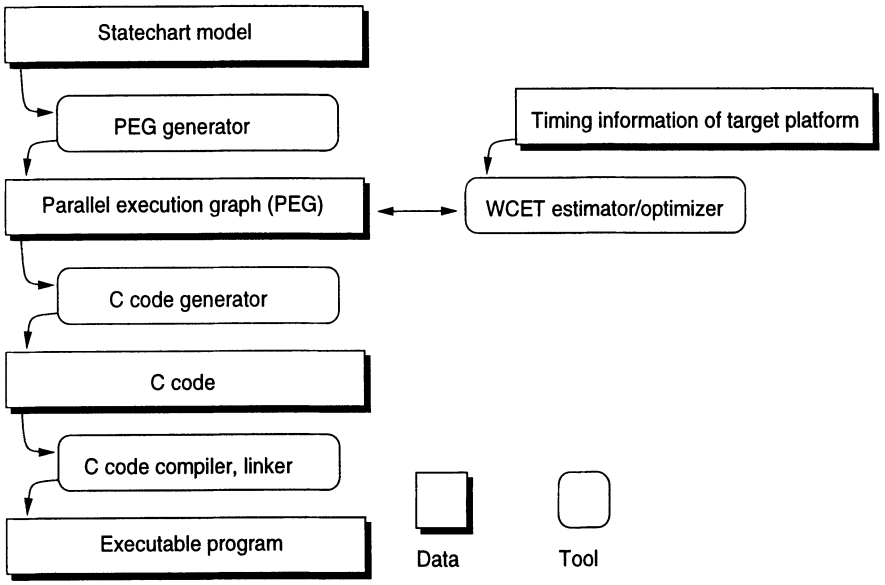

Figure 3: SCOT Transformation Flow

The C code generator produces C code for the PEG together with a tailored RTS which can be compiled, linked, and executed.

5.2. PEG Optimization

The PEG consists of conditional branches, variable assignments, parallel blocks, and join nodes to collect several path segments. Each path through the PEG corresponds to one possible statechart step. Thus optimizing the longest PEG path means optimizing the WCET of the implementation.

To calculate the WCET, the execution time of each node must be estimated. The estimation is based on the analysis of the node execution time on a 68000 platform.

Optimizations are possible for variables, when a variable is buffered and the buffering can be eliminated. Remember that in this case, the execution time of all nodes referencing this variable, is shortened.

Figure 4 shows a PEG example. Each path leads from START to END. Both path

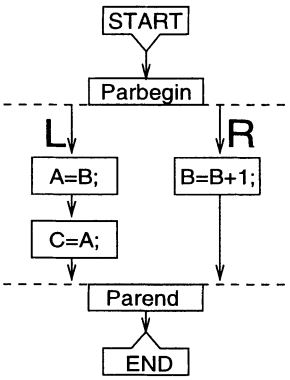

Figure 4: PEG Example

segments between Parbegin and Parend must be contained in all paths through the PEG, but the order is arbitrary. Because of the statechart semantics, changes to variables must not be observable by other statements before the next step. In all paths, no write to a variable may be followed by a read, otherwise, the variable must be buffered (changes are applied upon reaching END). In the figure, variable B needs no buffering if segment L is executed before R. Thus, selecting the order "L before R" reduces variable buffering, hence execution time of the nodes referencing B.

On the other hand, A needs buffering, since it is read after being written in segment L.

For each optimization step, the path through the PEG with the longest WCET is determined. All referenced variables with buffering along the path are collected. For each variable all references are checked w. r. t. data dependencies to path segments before and after the reference as well as parallel segments.

The variable contributing most to the execution time of the path is selected for optimization. If an execution order of segments is possible so that the buffering is unnecessary, this segment order is fixed, the buffering is deleted, and the execution time is fixed. In some situations, no order can be found because of cyclic data dependencies. In this case, one variable is marked as unoptimizable breaking the cycle. If there is still some DOF left, the next iteration is started.

5.3. WCET-Optimized Runtime System

Generated events must be cleared. Both generating and clearing is associated with an RTS overhead, in that the event must be generated during step n and cleared at the end of step $n+1$. Generating an event contributes to the execution time of the PEG nodes; the RTS is responsible for clearing events. Good *average* execution times can be achieved using an event clear list like in the Statemate-generated code, where all generated events are inserted for clearance after the next step. Unfortunately, the length of the list contributing to the *WCET* depends on the number of generated events in the

execution of the implementation, which cannot be determined at compile time. Counting all existing events as maximal length would result in an unrealistic WCET estimation.

To alleviate this problem, we selected a timestamp approach. Generating an event means setting the timestamp to the *actual step counter* (ASC), which is incremented in each step. Checking whether an event has been generated in the last step results in a comparison between its timestamp and the ASC.

Since statechart models are reactive systems with continuous use, a simple incrementation of the ASC would not be possible because of the overflow due to the limited range of integers. At special re-initialization steps, the ASC and the timestamps must be set to zero. This special step would require an extraordinary amount of time resulting in a loose WCET prediction.

Therefore, we partition all existing events in p *event groups*. Each group is associated with a *individual* ASC. In each step, one of the p groups is re-initialized together with its ASC. Using this approach, the overhead for the re-initialization is distributed equally to p steps.

6. Application Example

As an example for the optimization process, a system for an elevator control is used. Figure 5 shows the statechart model; it contains parallel and hierarchical con-

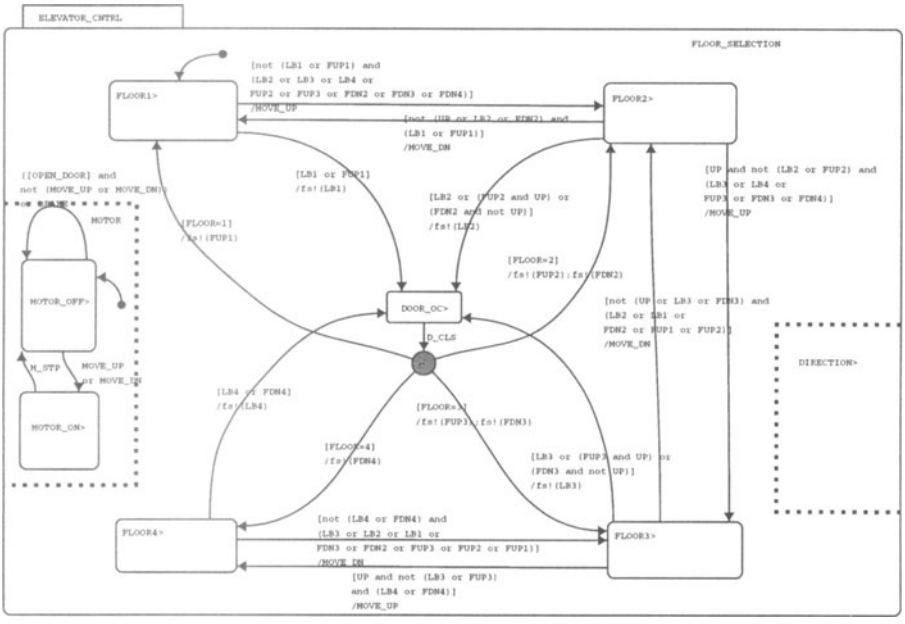

Figure 5: Application Example: Elevator Control System

structs as well as complex transition triggers. 19 variables and 8 events are used; a standard code generator would use double buffering for all of them.

After model analysis and selecting an appropriate execution order, the SCOT code generator needs buffering for only three variables and three events reducing the execution time from 1142 to 842 clock cycles (26 % less).

References

[Hare87] D. Harel; Statecharts: *a Visual Formalism for Complex Systems*; Science of Computer Programming, Vol. 8, August 1987.

[Ilog97] *Statemate Magnum Reference Manual*; i-Logix Inc., Andover, USA, 1997.

[Itut93] *CCITT Specification and Description Language*; Recommendation Z.100, ITU-T, Geneva, Switzerland, 1993.

[RJJC95] M. Romdhani, A. A. Jerraya, A. Jeffroy, P. de Chazelles, A.-El-K. Sahraoui; *Modeling and Rapid Prototyping of Avionics Using STATEMATE*; 6th IEEE International Workshop on Rapid Systems Prototyping, Chapel Hill, North Carolina, USA, 1995.

[ScSc96] S. Schulz, M. Schütze; *Modeling and Simulating an Home Automation System Using STATEMATE* (in German); 4th German User Meeting for STATEMATE, Munich, Germany, 1996.

[Spre96] M. Spreng; *Rapid Prototyping of electronic Control Systems in the Car Development* (in German); Thesis, University of Karlsruhe, Germany, 1996.