In *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES '99)*,
Rome, Italy, pages 173–177, May 3–5, 1999.

Scheduling Hardware/Software Systems Using Symbolic Techniques

Karsten Strehl and Lothar Thiele
Computer Engineering and Networks Lab (TIK)
Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland

Dirk Ziegenbein and Rolf Ernst
Institute of Computer Engineering (IDA)
Technical University of Braunschweig
Braunschweig, Germany

Jürgen Teich
Computer Engineering Lab (DATE)
University of Paderborn
Paderborn, Germany

Abstract

In this paper, a scheduling method for heterogeneous embedded systems is developed. At first, an internal representation model called *FunState* is presented which enables the explicit representation of non-determinism and scheduling using a combination of functions and state machines. The new scheduling method is able to deal with mixed data/control flow specifications and takes into account different mechanisms of non-determinism as occurring in the design of embedded systems. Constraints imposed by other already implemented components are respected. The scheduling approach avoids the explicit enumeration of execution paths by using symbolic techniques and guarantees to find a deadlock-free and bounded schedule if one exists. The generated schedule consists of statically scheduled basic blocks which are dynamically called at run time.

1 Introduction

One of the major sources of complexity in the design of embedded systems is related to their heterogeneity. On the one hand, the specification of the functional and timing behavior necessitates a mix of different basic models of computation and communication which come from transformative or reactive domains. In addition, we are faced with an increasing heterogeneity in the implementation. This not only concerns the functional units which may be implemented in form of dedicated or programmable hardware, microcontrollers, domain specific or even general purpose processors. In addition, these units communicate with each other via different media, e.g., busses, memories, networks, and by using many different synchronization mechanisms.

This heterogeneity caused a broad range of scheduling policies in hardware and software implementations. Two extreme possibilities are static schedules like those developed for *synchronous dataflow* (SDF) models [9], and EDF (*earliest deadline first*) schedules developed for dynamically changing task structures. Many intermediate possibilities have been developed over the years.

Recently, a methodology has been defined to deal with the modeling problem of complex embedded systems for the purpose of scheduling [16, 17]. The model SPI (*system property intervals*) as defined here is a formal design representation internal to a design system. It combines the representation of communicating processes with correlated operation modes, the representation of non-determinate behavior, different communication mechanisms such as queues and registers, and scheduling constraints.

The present paper is concerned with a scheduling procedure adapted to this kind of internal representation. Problems which are typical for the design of complex embedded systems are, e.g., different kinds of non-determinism such as partially unknown specification (to be resolved at design time), data-dependent control flow (to be resolved at run time), or unknown scheduling policy (to be resolved at compile time), and dependencies between design decisions for different system components. These properties necessitate new scheduling approaches as the number of execution paths to be considered grows exponentially with increasing degrees of non-determinism. Moreover, the complexity of the models of computation and communication greatly increases the danger of system deadlocks or queue overflows, see, e.g., [10].

Results are available which partly deal with above problems. To overcome drawbacks of either purely *static* or *dynamic* scheduling approaches and to combine their advantages, Lee proposed a technique called *quasi-static* scheduling [8]. Similarly to static scheduling, most of the scheduling decisions are made during the design process, providing only few run-time overhead and partial predictability. Only data-dependent choices—depending on the value of the data or resulting from a reactive, control-oriented behavior—have to be postponed until run time. Techniques related to quasi-static scheduling have been developed using, e.g., constraint graphs [7, 4], dynamic dataflow graphs [2], actors with data-dependent execution times [5], and free-choice Petri nets [12].

The approach taken in this paper is based on symbolic techniques which use a combination of efficient representations of state spaces and transition models and *symbolic model checking* principles in order to avoid the explicit enumeration of execution paths. Besides *binary decision diagrams* (BDDs) [1] and their derivatives, *interval diagram techniques*—using *interval decision diagrams* (IDDs) and *interval mapping diagrams* (IMDs)—have shown to be convenient for efficient formal verification of, e.g., process networks like the above-mentioned SPI model [13], Petri nets [14], or timed automata. There exist some approaches to apply symbolic methods to control/data path scheduling for high-level synthesis. BDDs are used to describe scheduling constraints and solution sets either directly [11] or encapsulated in *finite state machine* (FSM) descriptions [3, 6].

In [15], a common representation called *FunState* is presented which unifies many different well-known models of computation, supports stepwise refinement and hierarchy, and is suited to represent many different synchronization, communication, and scheduling policies. Based on this model, we present an approach to symbolic scheduling using interval diagrams techniques. In particular, the following new results are described in the paper:

- A refinement of the SPI model of computation [16, 17] called *FunState* is presented which enables the explicit representation of different mechanisms of non-determinism and scheduling using a mixture of functional programming and state machines.
- A scheduling method for heterogeneous embedded systems is developed which takes into account these different kinds of non-determinism and constraints imposed by other already implemented components and which deals with mixed data/control flow specifications.
- The resulting scheduling automaton is optimized with respect to the length of static blocks and the number of states.
- The approach is illustrated using a hardware/software implementation of a fast molecular dynamics simulation engine.

2 FunState and Scheduling

Mainly in the fields of embedded systems and communication electronics, common forms of representation for mixed control/data-oriented systems have gained in importance. Therefore, the FunState formalism has been developed which combines dataflow properties with finite state machine behavior [15]. It refines the SPI model of computation [16, 17] by introducing internal states, e.g., for modeling scheduling policies. FunState can be used as an internal representation in the design phase.

2.1 The Model of Computation

In this paper, only the aspects of FunState related to scheduling are described. The reader is referred to [15] for a formal introduction. In Figure 1, a simple example FunState model is shown. It consists of three components; each of them has two parts: an upper, data-oriented part—depicting dataflow by functional units (rectangles) and FIFO queues (circles)—and a lower, control-oriented part—described by a finite state machine.

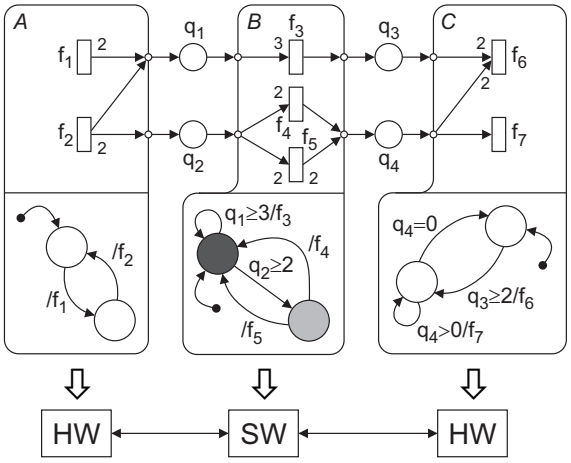

Figure 1: Example FunState model.

The queues in the data-oriented part store data items depicted by tokens, while the functional units perform computations on the data. The functions have consumption and production rates for each connected edge which are depicted only for values different from 1. The execution of the functions of each component is controlled by the corresponding state machine described in a statechart-like manner.

The labels of the state machine transitions indicate combinations of a condition and an action (e.g., "$q_1 \geq 3/f_3$"), meaning that the respective transition, and thereby the action, may be executed only if the condition is satisfied (i.e., if the queue labeled with q_1 contains at least three data items). If the above transition is taken, function f_3 is executed and consumes three tokens from queue q_1 and produces one token for queue q_3.

In the scope of this paper, we use only a simple subset of FunState suitable for scheduling. While the transition predicates in general may be also on values of data items, we allow only predicates on *queue contents*—the numbers of tokens in queues. We ignore explicit timing properties (execution times, timing constraints, etc.). The concurrent execution of state machines of different components is asynchronous and interleaved.

2.2 The Problem

Consider a constellation of components mapped onto different implementational units and communicating via queues in a distributed, parallel setting. The components have both data and control flow properties. Non-determinisms may exist resulting from incomplete specifications or data dependencies resolved only at run time. In this paper, we deal with the problem of finding a *feasible* schedule for the components mapped onto one implementational unit respecting constraints given by other components. In this context, feasible means that the schedule is deadlock-free and guarantees bounded queue contents.

To precise this, we consider a simple example. Assume that component B of the example FunState model of Figure 1 represents a processor transforming data streams between the components A and C. Let A and C be components mapped onto hardware such as an input or output device, respectively, or an interface to a sensor, an actor, or another processor.

Let the behaviors of A and C be specified by the respective state machine. Not considering these additional constraints may lead to less efficient or even incorrect schedules. The state machine of A describes that its functions always are executed in the order $f_1 f_2 f_1 f_2 \ldots$ Hence, it is guaranteed that after each firing of f_1, f_2 is executed and vice versa.

The state machine of B shown in Figure 1 describes a specification of possible schedules for B. This specification should be used to find a feasible schedule which respects the additional information concerning other components. All transitions starting in a dark-shaded state represent design *alternatives* which may be chosen during schedule development. In contrast to this, a light-shaded state contains a *conflict* concerning its outgoing transitions. The conflict can be resolved only at run time, hence, no design decision is possible. Conflicts occur, for instance, when decisions depending on the value of data or environmental circumstances have to be taken. White states in the FunState model are states which either have only one outgoing transition or of which all transitions have disjoint predicates. Thus, the transition behavior of these states is *determinate*. Note that in component C the state with two outgoing transitions is determinate for this reason.

Suppose that B and C execute sufficiently often (they are "faster" than the preceding component) such that there are no unbounded numbers of tokens simultaneously in q_1 and q_2 or in q_3 and q_4, respectively. An important issue of schedule development are feasibility and correctness of the resulting schedule. A possible schedule of B described by the specification is $(f_4|f_5)(f_4|f_5)\ldots$, where f_4 and f_5 are executed alternatively and iteratively—thus ignoring f_3. But this schedule is not feasible as the queue contents of q_1 and q_4 are not bounded. If we had chosen $f_3(f_4|f_5)f_3(f_4|f_5)\ldots$—$f_3$ is executed, then f_4 or f_5 is executed, etc.—, this would result in an incorrect behavior of C as f_6 could attempt to read too much tokens from q_4 after some time.

In contrast to this, the schedule $(f_4|f_5)f_3(f_4|f_5)f_3\ldots$ is valid with respect to specification and component C, and it is feasible. An implementation of this schedule can profit from the fact that f_3 may be executed only if f_4 has been executed immediately before. From the behavior of A follows that for the execution of f_3 no condition is necessary as q_1 always contains enough tokens. Thus, the resulting schedule may be implemented more efficiently by considering only necessary conditions as less queue contents have to be determined.

Using the symbolic scheduling techniques proposed below, the above issues are taken into account. Intuitively, the scheduling is performed by replacing dark-shaded states by white states—taking decisions and thus removing design alternatives. In this paper, we consider only software scheduling using a uniprocessor. Extensions for hardware scheduling under resource constraints or scheduling for several processors are easily possible.

2.3 FunState and Symbolic Methods

With regard to formal verification, the techniques for symbolic model checking of process networks based on interval diagram techniques as described in [13] are directly applicable to FunState as the transition behavior of FunState is very similar to that of the considered models of computation. Thus, using FunState to model a mixed hardware/software system enables its formal verification comprising the whole well-known area of symbolic model checking concerning the detection of errors in specification, implementation, or scheduling. Properties as the correctness of a schedule may be affirmed by proving the boundedness of the required memory and the absence of artificial deadlocks. In the scope of this paper, symbolic methods based on interval diagram techniques are used not only to analyze but even to develop scheduling policies for FunState models.

3 Interval Diagram Techniques

For formal verification of, e.g., process networks [13], Petri nets [14], or timed automata, interval diagram techniques—using interval decision diagrams (IDDs) and interval mapping diagrams (IMDs)—have shown to be a favorable alternative to BDD techniques. This results from the fact that for this kind of models of computation, the transition relation has a very regular structure that IMDs can conveniently

represent. While BDDs have to represent explicitly all possible state variable value pairs before and after a certain transition, IMDs store only the *state distance*—the difference between the state variable values before and after the transition. In this paper, we only give a brief, informal summary of structure and properties of IDDs and IMDs and the methods required for scheduling.

3.1 Interval Decision Diagrams

IDDs are a generalization of BDDs and MDDs—*multi-valued decision diagrams*—allowing diagram variables to be integers and child nodes to be associated with intervals rather than single values. In Figure 2a), an example IDD is shown. It represents the Boolean function $s(u,v,w) = (u \leq 3) \wedge (v \geq 6) \vee (u \geq 4) \wedge (w \leq 7)$ with $u,v,w \in [0,\infty)$.

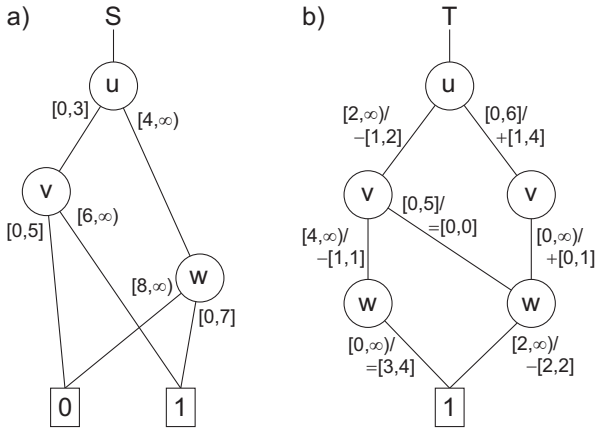

Figure 2: Interval decision diagram and interval mapping diagram.

Equivalent to BDDs, IDDs have a reduced and ordered form, providing a canonical representation of a class of Boolean functions—which is important with respect to efficient fixpoint computations often necessary for formal verification, and also for the symbolic scheduling techniques considered here. IDDs are used to represent state sets during scheduling.

3.2 Interval Mapping Diagrams

IMDs represent valid state transitions, for instance, the execution of functions depending on predicates on queue contents. IMDs are represented by graphs similar to IDDs. Their edges are labeled with functions mapping intervals onto intervals. The graph contains only one terminal node. Figure 2b) shows an example IMD. With regard to transition relations, IMDs work as follows. Each edge is labeled with a condition—the *predicate interval*—on its source node variable and the kind and amount of change—the *action operator* and the *action interval*—the variable is to undergo. Each path represents a possible state transition which is executable if all edges along the path are enabled.

3.3 Image Computation

Similarly to formal verification like symbolic model checking, an operation named *image computation* is fundamental for symbolic scheduling techniques. The image $Im(S,T)$ of a set S of system states with respect to transition relation T represents the set of all states that may be reached after exactly one valid transition from a state in set S. In [13], an efficient algorithm is described to perform forward or backward image computation using an IDD S for the state set and a IMD T for the transition relation, resulting in an IDD S' representing the image state set.

4 Symbolic Scheduling

Symbolic methods for control-dependent scheduling have shown to be effective techniques to perform control/data path scheduling, e.g., [6]. They often outperform both ILP and heuristic methods while yielding exact results. Furthermore, all possible solutions to a given scheduling problem are computed simultaneously such that additional constraints may be applied to find optimal schedules. In this paper, we present a symbolic approach to the scheduling of systems represented as FunState models. The approach based on interval diagram techniques avoids the explicit enumeration of execution paths by using these symbolic techniques.

4.1 Conflict-Dependent Scheduling

As mentioned in Section 1, quasi-static and related scheduling approaches, e.g., [8, 4], try to combine the advantages of static and dynamic scheduling methods. To achieve this, the resolution of data or environment dependent control is done at run time whereas the tasks that need to be executed as a consequence of a run-time decision are scheduled statically. The aim is to make most of the scheduling decisions at compile time, leaving at run time only choices that, e.g., depend on the value of data. As mentioned in Section 2.2, we call this latter kind of run time choices *conflicts* and the corresponding scheduling techniques *conflict-dependent*. The former design decisions at compile time are named *alternatives*. As we ignore explicit timing properties in the scope of this paper, the resulting schedule—similarly to scheduling of, e.g., marked graphs—consists of sequences of function executions.

Initially, the given FunState model contains a *schedule specification automaton* which extends the FSM part such that all possible schedule behaviors are modeled. This FunState model represents a totally dynamic scheduling behavior and is used to perform the symbolic scheduling procedure as described below. The result of this procedure is the *schedule controller automaton* which restricts the scheduling behavior to be only conflict-dependent. This automaton may replace the specification automaton of the original FunState model, e.g., for analysis purposes such as verification. Finally, the controller automaton may be transformed into program code to implement the controller.

4.2 Conflicts and Alternatives

A conflict in our understanding is a non-determinism in the specification which may not be resolved as a design decision, but of which all possible execution traces have to be taken into account during the schedule. Thus, the multi-reader queue q_4 in Figure 1 does not represent a conflict as both following functions may read all tokens of q_4 independent of their value or possible external circumstances.

In contrast to that, the queue q_1 in Figure 3a) is a multi-reader queue that may contain tokens which only one of the queue's readers f_2 and f_3 consumes (depending, e.g., on the token data) but the other one does not. Besides such data-dependent conflicts, conflicts depending on environmental circumstances may occur.

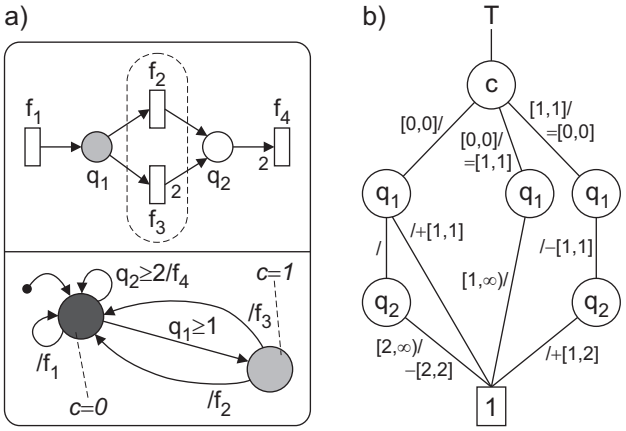

Figure 3: FunState model of conflict and transition relation IMD.

The states of the FSM part of FunState models are divided into three types. According to Section 2.2, light-shaded states are called *conflict*

states, dark-shaded states are *alternative state*, and *determinate states* are white. While the property of a state to be determinate is derived directly from its transition predicates, the non-determinate states have to be divided explicitly into conflict states and alternative states as both are semantical properties. All transitions leaving an alternative state represent design choices which may be made during the schedule development. In contrast to that, all transitions leaving a conflict state represent decisions which may not be taken at compile time, but which keep their non-determinate character until run time.

Determinate states with only one outgoing transition are called *static* as there exists only one possibility to quit them. Determinate states with more than one transition, alternative states, and conflict states are named *dynamic* because they represent a dynamic execution behavior with several traces depending, e.g., on queue contents or data.

4.3 Schedule Specification Automaton

To model the above-mentioned conflicts, a schedule specification automaton is built which represents all possible conflict behaviors and thus specifies all valid schedules. The lower part of Figure 3 a) shows the specification automaton used to describe the above-mentioned conflict behavior concerning f_2 and f_3 with regard to q_1. When one of the functions is enabled—q_1 contains at least one token—, the automaton can make a transition from the initial alternative state to the conflict state. Then, after executing either f_2 or f_3 it returns to the alternative state.

Besides the variables for the queue contents, a state variable c for the FSM states has been introduced. Figure 3 b) shows the interval mapping diagram representing the transition relation of the FunState model of Figure 3 a). This IMD is used for symbolic state traversal as explained below.

4.4 Performing Symbolic Scheduling

The aim of the described scheduling process is to sequentialize functions specified as concurrent while preserving all given conflict alternatives. The resulting schedule has to be deadlock-free and bounded as mentioned in Section 2.2.

Figure 4, shows the *regular state transition graph* of the FunState model in Figure 3. It represents all valid state transitions of the FunState model with regard to the total state space consisting of the queue contents of the dataflow part and the discrete system states of the FSM part. At each coordinate pair of (q_1, q_2), both possible states of the FSM part are shown.

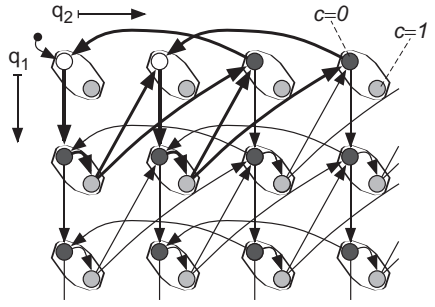

Figure 4: Regular state transition graph with schedule.

Using interval diagram techniques, the regular state transition graph is traversed symbolically without constructing it explicitly. This is achieved by iterative image computations as explained in Section 3. An interval mapping diagram such as shown in Figure 3 b) represents the transition relation, while interval decision diagrams are used to store intermediary state sets. The efficiency of these techniques has been shown in [13].

In the following, the scheduling procedure in its simplest form is explained with this graph. First, a symbolic breadth-first search is performed to find the shortest paths from the initial state to itself or any state already visited during the search. One of these (possibly multiple) shortest paths—representing or at least containing a cycle—is selected as the basis of the following scheduling procedure.

All states of the selected path corresponding to conflict states need further investigation as no conflict decision may be taken during the schedule design. Hence, beginning with the successor states of the conflict states again a breadth-first search is performed until reaching any state visited yet. Additional conflict states visited during this search are also treated as described above.

The schedule is complete when each successor state of each visited conflict state has been considered. Thus, it is guaranteed that any conflict alternative during run time may be treated by providing a static schedule until the next conflict to be resolved. The resulting schedule is marked by bold arcs in Figure 4.

If no schedule has been found while traversing one of the conflict paths, another shortest path is selected to repeat the scheduling procedure. If all shortest paths have been checked without finding a complete schedule, longer paths are selected. By introducing a bounding box on the state space, the search space may be restricted. Thus, the termination of the algorithm is guaranteed. Furthermore, if a deadlock-free and bounded schedule exists, the above procedure will find it.

4.5 Schedule Controller Generation

The resulting schedule consists of paths of the regular state transition graph as shown in Figure 4. The corresponding subgraph in Figure 5 a) is the basis for the generation of the controller automaton. As a consequence of the scheduling process, all alternative states have been been replaced by determinate states—taking decisions and thus removing design alternatives. The predicate p identifies the run-time decision associated to the conflict node.

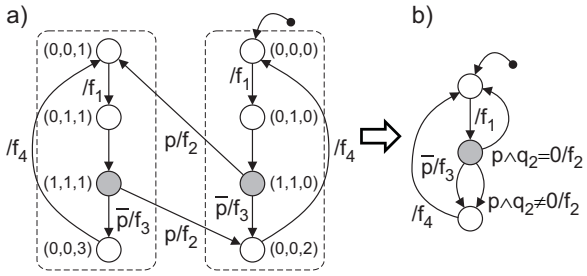

Figure 5: State transition graph of schedule for (c, q_1, q_2) and resulting controller automaton.

In order to reduce the implementation effort, this state transition graph may be simplified. Obviously, this process can be driven by many different objectives, for instance, minimizing the number of states in the schedule automaton or keeping sequences of static nodes.

As an example, a procedure is described which minimizes the number of states under the condition that sequences of static nodes are not partitioned. This way, the number of dynamic decisions (at run time) is not increased in any execution trace. The optimization procedure is based on well-known state minimization methods and uses the following equivalence relation:

- Two static states are equivalent iff for any input they have identical outputs and the corresponding next states are equivalent.
- Two dynamic states are equivalent iff they are of the same type (conflict, alternative, or determinate) and they correspond to the same node in the non-scheduled state machine, i.e., they have the same state name but different queue contents associated.

This definition can be used to perform the usual iterative partitioning of the state set until only equivalence classes are obtained. The ambiguity of the next states in the case of dynamic states is resolved by adding predicates to the outgoing edges. Figure 5 b) shows the controller automaton as the result of this process. It may be transformed easily into program code as shown in Table 1 as pseudo code.

```
a: f1;
   if p then
      f2;
      if q2 = 0 then goto a;
   else f3;
   f4;
   goto a;
```

Table 1: Controller program code.

4.6 Molecular Dynamics Simulation Example

The introduced approach has been applied to perform conflict-dependent scheduling for a molecular dynamics simulation system. As shown in Figure 6, the simplified fundamental algorithm has been mapped onto a host workstation (*Host*) linked to a special purpose hardware accelerator serving as a coprocessor (*CoPro*). In the figure, the circles containing a square represent registers storing data. Therefore, they do not introduce additional dependency constraints. The transition labels $l_1, \ldots, l_4$ are depicted separately for reasons of space.

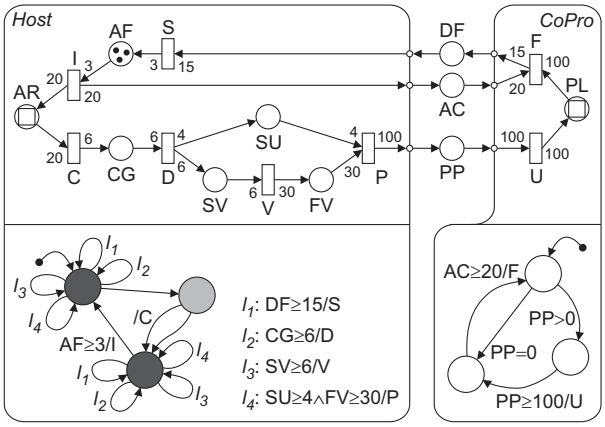

Figure 6: Molecular dynamics model with specification automaton.

The simulation mainly consists of repeated computations in the feedback loop distributed among both processors where atom forces (*AF*) are computed (*F*), added up (*S*), and integrated (*I*) to calculate new atom coordinates (*AC*, *AR*). After a variable number of iterations, the central coordinates of slowly moving sub-molecules called charge groups (*CG*) are updated (*C*). Then, a new list of neighbors called pair list (*PL*) is computed (*D*, *V*, *P*, *U*).

As the moment when to start this pair list computation is unknown until run time, this fact represents a conflict which is modeled using a conflict state. The major issue of the schedule specification is that there exists no cycle in the corresponding state transition graph which does not contain the conflict state. This is ensured by the fact that the transition executing *I* cannot be reached without visiting the conflict state. The result of the symbolic scheduling process—the schedule controller automaton—is shown in Figure 7. It replaces the FSM part of the *Host* component of Figure 6. It consists of two static cycles and a conflict state switching between them. The schedule is respecting the specification of *CoPro*. Note that even the schedule of *CoPro* is not static as it depends on the content of queue *PP*.

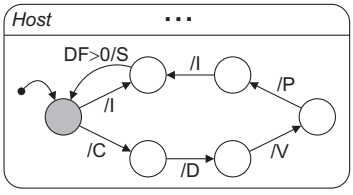

Figure 7: Resulting controller automaton.

5 Summary and Conclusion

An approach for symbolic scheduling of mixed hardware/software systems has been presented. It is based on a FunState model of the system and the scheduling constraints. Further work concentrates on extending the approach to hardware scheduling under more complex resource constraints and on considering the timing behavior of the system to allow for the specification of timing constraints.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):667–691, August 1986.

[2] J. T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, University of California, Berkeley, 1993.

[3] C. N. Coelho Jr. and G. De Micheli. Dynamic scheduling and synchronization synthesis of concurrent digital systems under system-level constraints. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-94)*, pages 175–181, 1994.

[4] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli. Software synthesis for real-time information processing systems. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 260–279. Kluwer Academic Publishers, 1995.

[5] S. Ha and E.A. Lee. Compile-time scheduling of dynamic constructs in dataflow program graphs. *IEEE Transactions on Computers*, 46(7):768–778, July 1997.

[6] S. Haynal and F. Brewer. Efficient encoding for exact symbolic automata-based scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, 1998.

[7] D. C. Ku and G. De Micheli. Relative scheduling under timing constraints: algorithms for high-level synthesis of digital circuits. *IEEE Transactions on Computer-Aided Design*, 11(6):696–718, June 1992.

[8] E. A. Lee. Recurrences, iteration, and conditionals in statically scheduled block diagram languages. In R. W. Brodersen and H. S. Moscovitz, editors, *VLSI Signal Processing III*, pages 330–340. IEEE Press, New York, 1988.

[9] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[10] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.

[11] I. Radivojević and F. Brewer. Ensemble representation and techniques for exact control-dependent scheduling. In *Proceedings of the 7th International Symposium on High-Level Synthesis*, pages 60–65, 1994.

[12] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli. Quasi-static scheduling of embedded software using free-choice Petri nets. In *Proceedings of the Workshop on Hardware Design and Petri Nets (HPWN '98)*, 1998.

[13] Karsten Strehl and Lothar Thiele. Symbolic model checking of process networks using interval diagram techniques. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, pages 686–692, San Jose, California, November 8–12, 1998.

[14] Karsten Strehl and Lothar Thiele. Interval diagram techniques for symbolic model checking of Petri nets. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE99)*, Munich, Germany, March 9–12, 1999.

[15] Lothar Thiele, Jürgen Teich, Martin Naedele, Karsten Strehl, and Dirk Ziegenbein. SCF—state machine controlled flow diagrams. Technical Report TIK-33, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, January 1998.

[16] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign (Codes/CASHE '98)*, pages 9–13, Seattle, Washington, March 1998.

[17] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Representation of process mode correlation for scheduling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD-98)*, San Jose, California, November 8–12, 1998.