Scheduling Hardware/Software systems using symbolic techniques

Scheduling Hardware/Software Systems Using Symbolic Techniques

Karsten Strehl and Lothar Thiele

Computer Engineering and Networks Lab (TIK)
Swiss Federal Institute of Technology (ETH)
Gloriastrasse 35, 8092 Zurich, Switzerland
eMail: {strehl,thiele}@tik.ee.ethz.ch
WWW: http://www.tik.ee.ethz.ch

Dirk Ziegenbein and Rolf Ernst

Institute of Computer Engineering (IDA)
Technical University of Braunschweig
Hans-Sommer-Straße 66, 38106 Braunschweig, Germany
eMail: {ziegenbein,ernst}@ida.ing.tu-bs.de
WWW: http://www.ida.ing.tu-bs.de

Abstract

In this report, 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.

Contents

Chapter 1

Introduction

One of the major sources of complexity in the design of embedded systems is related to their heterogeneity in different ways. 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: *synchronous dataflow* (SDF) [LM87], generalized state machines [Har87, BGJ$^+$97], *dynamic dataflow* [Buc93], rendez-vous, queue-based, just to name a few. Typical examples for this are MPEG-2 video encoders or network switching circuits.

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 hardware, programmable hardware, domain specific processors such as DSPs, VSPs (*video signal processors*), and microcontrollers 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 SDF models [LM87], 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 for complex embedded systems for the purpose of scheduling [ZER$^+$98a, ZER$^+$98b]. The model SPI (*system property intervals*) as defined here is a formal design representation internal to a design system. It combines the representation of communicating processes with correlated operation modes, the representation of non-determinate behavior, different communication mechanisms such as queues and registers, and scheduling constraints. It is based on simple basic constructs annotated with information necessary for scheduling across different input language semantics. In [ZER$^+$98a], a common internal representation is presented which integrates the aspects of several models of computation and which is targeted to scheduling and allocation. [ZER$^+$98b] considers classes of applications that feature communicating processes where the functions depend on a finite set of computation modes.

The present report is concerned with a scheduling procedure adapted to this kind of internal representation. Problems which are typical for the design of complex embedded

1

systems are

- 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), and unknown scheduling policy (to be resolved at compile time),

- dependencies between design decisions for different system components,

- correlations between processing modes in different 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., [LP95].

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 [Lee88]. 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 [KD92, CTGC95], dynamic dataflow graphs [Buc93], actors with data-dependent execution times [HL97], and free-choice Petri nets [SLWSV98].

The approach taken in this report 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. Symbolic approaches often turned out to provide advantages regarding computation time and memory resources compared to conventional approaches. Hence, besides *binary decision diagrams* (BDDs) [Bry86] 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 [ST98a], Petri nets [ST98b, ST99], or timed automata [Str98].

There exist some approaches to apply symbolic methods to control-dependent scheduling for high-level synthesis. These exact and efficient symbolic scheduling techniques based on BDDs are used to perform control/data path scheduling by combining the advantages of both heuristic methods and techniques based on *integer linear programming* (ILP). BDDs are used to describe scheduling constraints and solution sets either directly [RB94] or encapsulated in *finite state machine* (FSM) descriptions [CD94, HB98].

In [TTN+98], 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 report:

- A refinement of the SPI model of computation [ZER$^+$98a, ZER$^+$98b] called *Fun-State* is presented which enables the explicit representation of non-determinism and scheduling using a mixture of functional programming and state machines.

- Different mechanisms of non-determinism as occurring in the design of embedded systems are classified.

- 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. It avoids the explicit enumeration of execution paths 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.

- 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.

The report is organized as follows. Section 2 briefly introduces the FunState model and its scheduling concerns. Interval diagram techniques are summarized in Section 3, while Section 4 explains our approach. Finally, Section 5 gives a short summary.

Chapter 2

FunState and Scheduling

During the last years, mainly in the fields of embedded systems and communication electronics (for instance, cellular phones, MPEG, ATM) 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 [TTN$^+$98]. It refines the SPI model of computation [ZER$^+$98a, ZER$^+$98b] by introducing internal states, e.g., for modeling scheduling policies. FunState can be used to model mixed hardware/software systems for analysis and synthesis purposes. It serves as an internal representation in the design phase.

2.1 The Model of Computation

In this report, FunState is described only briefly. Only the aspects related to scheduling are considered. The reader is referred to [TTN$^+$98] for a formal introduction. In Figure 2.1, 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.

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. One major novelty of FunState compared to related models of computation is that the functions are passive. 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). The purpose of such a predicate mostly is to ensure the presence of the input data tokens needed for an execution of the corresponding function. 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 report, we use only a simple subset of FunState suitable for
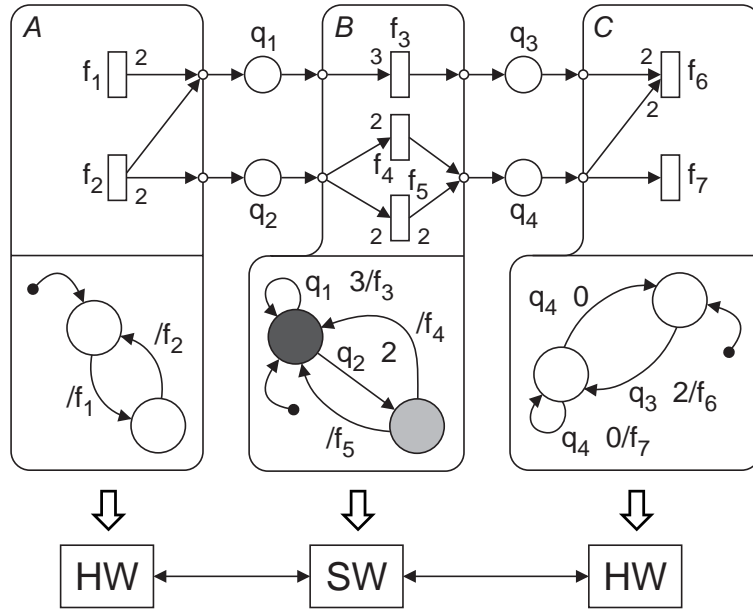
4

Figure 2.1: Example FunState model.

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

Besides conventional scheduling constraints such as data dependencies or resource constraints, even more complex constraints may be imposed to influence a schedule to be developed. For instance, interface protocols between several components or various external constraints may have to be taken into account. Furthermore, the desired scheduling policy may be given partially resulting in an incomplete specification which has to be obeyed during schedule development.

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 report, 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 2.1 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. Their behavior has to be considered with regard to the schedule to be developed. 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 2.1 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. This simple example schedule may be seen as a self-triggered schedule or a trivial case of a quasi-static schedule. Data independent operations are scheduled statically in clusters starting with one data-dependent operation. Only for the beginnings of these clusters dynamic scheduling is necessary.

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. Our approach resulting in a schedule similar to quasi-static obeys constraints given by external specifications and uses this additional information to obtain more efficient schedules. In this report, 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

Symbolic representations of Boolean functions such as BDDs have formed the basis of the breakthrough of symbolic formal verification methods such as symbolic model checking, e.g., [McM93]. In contrast to other verification techniques such as simulation, formal verification considers *all* possible execution traces of a state transition model, not only a part. Hence, formal verification may provide a mathematical proof of the correctness of a system's model.

With regard to formal verification, the techniques for symbolic model checking of process networks based on interval diagram techniques as described in [ST98a] 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.

Apart from this, formal verification may assist during the development of scheduling policies. The system model may be extended to describe one or several dynamic or hybrid scheduling policies, too, of which the behavior is verified together with the system model. Thus, common 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 report, we mainly deal with symbolic scheduling as another task in the area of hardware/software codesign using symbolic methods. In addition to formal verification as described above, symbolic methods based on interval diagram techniques may be used not only to analyze but even to develop scheduling policies for FunState models.

Chapter 3

Interval Diagram Techniques

For formal verification of, e.g., process networks [ST98a], Petri nets [ST98b, ST99], or timed automata [Str98], 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. Especially for models with large numbers of tokens, this approach is reasonable and useful. IDDs are used to represent state sets during computations. In this report, we only give a brief, informal summary of structure and properties of IDDs and IMDs and the methods required.

Due to the similarities between the transition behaviors of FunState and the above-mentioned models of computation, the advantages of symbolic techniques based on interval diagram techniques may be transferred to the area of symbolic scheduling of FunState models—besides their formal verification.

3.1 Interval Decision Diagrams

IDDs are a generalization of BDDs and MDDs—*multi-valued decision diagrams* [SKMB90]—allowing diagram variables to be integers and child nodes to be associated with intervals rather than single values. In Figure 3.1 a), 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 $u, v, w \in [0, \infty)$.

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. Methods as the *If-Then-Else* operator ITE are defined similarly to their BDD equivalents and may be computed as usual for decision diagram applications using a computed table to improve performance. IDDs are used to represent state sets during scheduling.
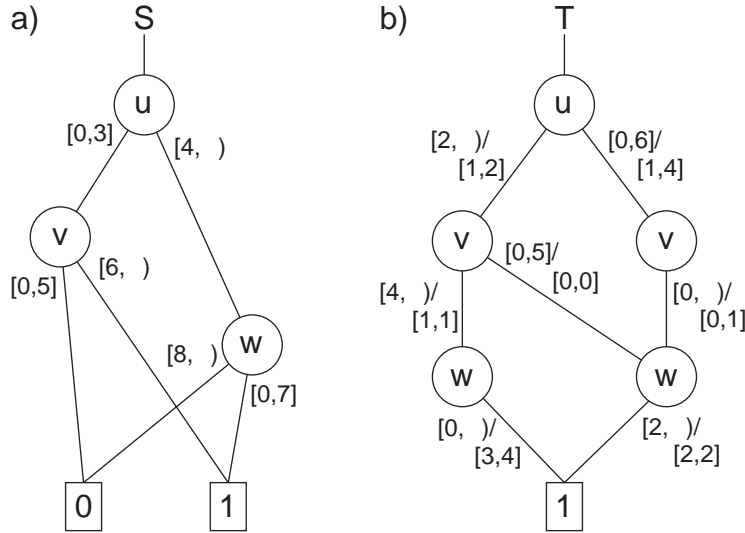
Figure 3.1: Interval decision diagram and interval mapping diagram.

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 3.1 b) 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. The combination of predicate and action interval parameterizes the mapping function and completely defines its behavior.

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. The inverse image $PreIm(S, T)$ represents all states that in one transition *can* reach a state in S.

In [ST98a], 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. This algorithm has been used to perform reachability analysis or symbolic model checking by fixpoint computation and is essential also for symbolic scheduling based on interval diagram techniques. IMDs are dedicated to image computation especially for models like FunState as the state distance

(action interval) combined with the respective firing condition (predicate interval) may be stored more efficiently than many state pairs.

Chapter 4

Symbolic Scheduling

Symbolic methods for control-dependent scheduling have shown to be effective techniques to perform control/data path scheduling, e.g., [HB98]. 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 report, 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., [Lee88, CTGC95], 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 report, 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.

11

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 2.1 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 4.1 a) 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.
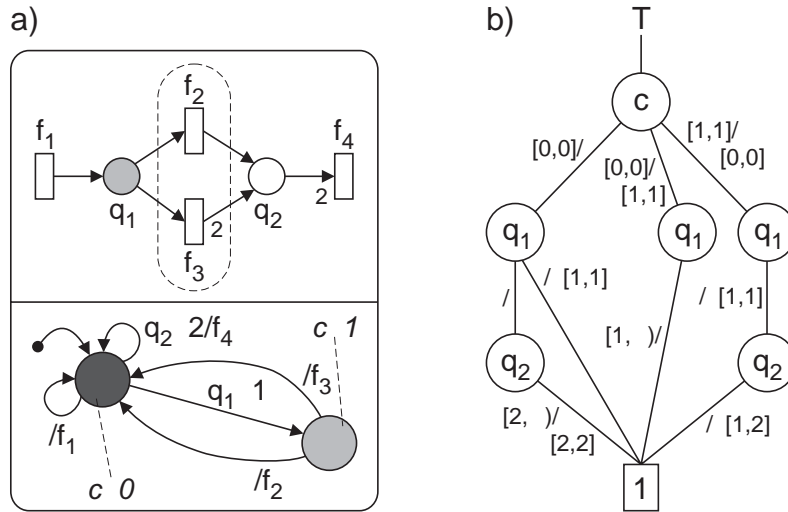

Figure 4.1: FunState model of conflict and transition relation IMD.

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. Hence, besides explicit conflicts incomplete constraint specifications resulting in a non-determinate behavior are modeled using conflict states. Otherwise the non-determinism would be treated as a design alternative and removed during the scheduling process.

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 4.1 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 4.1 b) shows the interval mapping diagram representing the transition relation of the FunState model of Figure 4.1 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 4.2, shows the *regular state transition graph* of the FunState model in Figure 4.1. 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.
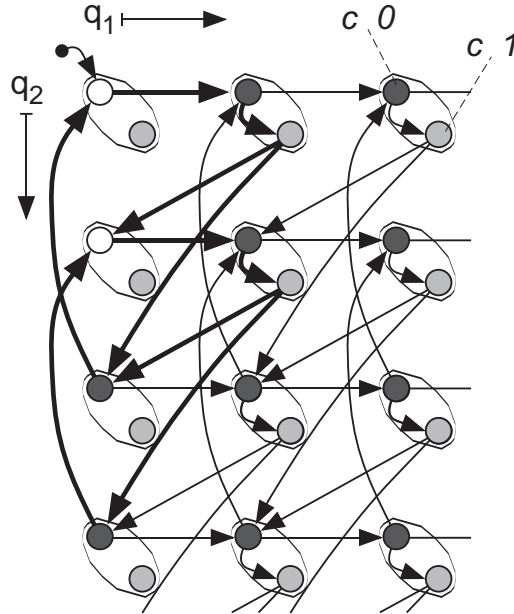

Figure 4.2: Regular state transition graph with schedule.

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 4.1 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 [ST98a].

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 4.2.

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.

4.4.1   Algorithms

In Table 4.1, the algorithm of $determineShortestPath(A, B)$ is described where A and B are sets of states which do not have to be disjoint. The result is one—out of possibly several—shortest path $x_0, x_1, \dots, x_n$ with respect to a transition relation T from any element of A to any element of B with $n \geq 1$.

The image operator $Im(S, T)$ and its inverse $PreIm(S, T)$ are introduced in Section 3.3. The choosing of one state out of a set of states in Table 4.1 preferably is done by selecting non-conflict states. This is a heuristic criterion to reduce the number of conflicts to be considered and, hence, the size of the search space and the resulting schedule.

The syntax $< x_0, x_1, \dots, x_n >$ represents a list P of elements x_i, starting with x_0. In the following, common functions for list manipulation such as $P.size()$, $P.head()$, $P.tail()$, or $P.elements()$ are used.

An outline of the recursive algorithm of $determineStateSchedule(a, B)$ is sketched in Table 4.2 where a is a state and B a set of states which may include a. The function call is by reference, hence, modifying the value of parameter B. The result is a directed graph of states representing the state transition graph of the schedule as explained below. The recursion is initiated by calling $determineStateSchedule(x_0, \{x_0\})$ where x_0 is the

$$determineShortestPath(A, B) :$$

```
    S_0 = A; n = 0;
    do
        S_{n+1} = Im(S_n, T);
        n = n + 1;
    until  S_n ∩ B ≠ ∅;
    choose a state x_n ∈ S_n ∩ B;
    for  i = n − 1 downto 0
        choose a state x_i ∈ S_i ∩ PreIm({x_{i+1}}, T);
    return  list < x_0, x_1, … , x_n >;
```

Table 4.1: Algorithm for shortest path search.

initial system state.

```
determineStateSchedule(a, B) :
    list P = determineShortestPath({a}, B);
    B = B ∪ P.elements();
    add P as subgraph to graph G;
    while  P.size() > 1
        x = P.head(); P = P.tail();
        if  isConflictState(x)
            for each  y ∈ Im({x}, T)
                add < x, y > as subgraph to G;
                if  y ∉ B
                    add subgraph determineStateSchedule(y, B) to G;
    return  G;
```

Table 4.2: Algorithm to determine the state transition graph of the schedule.

In Table 4.2, the construction of the resulting graph by adding subgraphs is not de-scribed in detail for the sake of clearness. The function value of $isConflictState(x)$ is $true$ iff x is a conflict state.

The algorithm of $determineStateSchedule(a, B)$ is sketched only very roughly. Several extensions are necessary not described in Table 4.2. For instance, often no cy-cle exists which includes the initial state x_0. Thus, beginning with x_0, the regular state transition graph has to be traversed until finding a cycle which forms the basis for a valid schedule. This may be regarded as an initialization phase at the beginning of the resulting schedule. The algorithm of $determineStateSchedule(a, B)$ has to be modified such that the first shortest path search is replaced by searching a path from the initial state to any state visited during this search.

4.5 Schedule Controller Generation

The resulting schedule consists of paths of the regular state transition graph as shown in Figure 4.2. The corresponding subgraph in Figure 4.3 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.
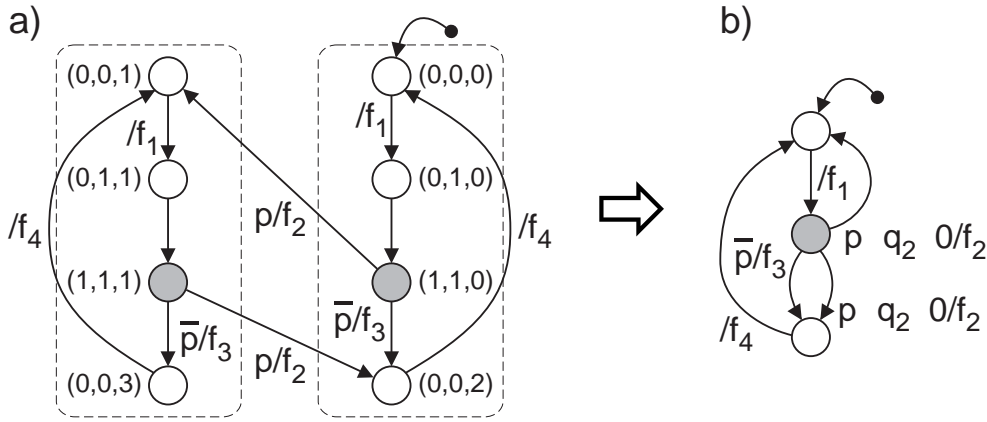

Figure 4.3: State transition graph of schedule for (c, q_1, q_2) and resulting controller automaton.

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, see, e.g., [Mic94]. The ambiguity of the next states in the case of dynamic states is resolved by adding predicates to the outgoing edges.

Figure 4.3 b) shows the controller automaton as the result of this process. It may be transformed into program code as shown in Table 4.3 as pseudo code.

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

Table 4.3: Controller program code.

4.6 Conflict Queues and Compositions

Up to here, a scheduling methodology has been explained which makes use of a schedule specification automaton given explicitly. In this section, a different methodology is described. It is based on an incomplete FunState model which does not contain a schedule specification automaton. For the component to be scheduled, only the dataflow part is given in a manner similar to a conventional Petri net. Based on this model, the schedule specification automaton is generated automatically. This is the advantage of this methodology because the explicit construction of the schedule specification automaton is not always an easy task. This strategy is sufficient for the conflict behaviors of many kinds of system models with data-dependent conflicts. However, the example model described in Section 4.7, for instance, requires an explicit schedule specification. Figure 4.4 shows this modified methodology for our symbolic scheduling approach. New terms used in the following explanations are defined below.
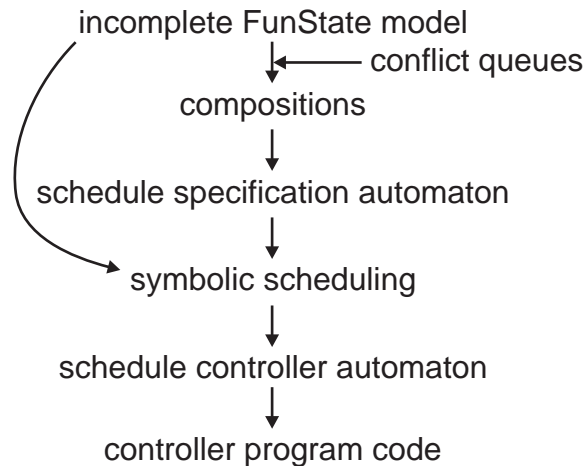

Figure 4.4: Symbolic scheduling methodology.

4.6.1 Preliminaries

As shown in Figure 4.4, an incomplete FunState model of the system to be scheduled including scheduling constraints is the basis of our approach. For this model, the user has to specify those queues involved in a conflict of functions as explained above. Based on these *conflict queues*, *compositions* of conflicting functions are determined. Then the schedule specification automaton is generated automatically. This extended FunState model is used to perform the symbolic scheduling procedure as described above.

Definition 4.1 (Conflict queue) *A* conflict queue *is a multi-reader queue that may contain tokens which only some of the queue's readers consume (depending, e.g., on the token data) but the others do not.*

Queues have to be specified explicitly by the user as conflict queues because this is a semantical property.

Definition 4.2 (Composition) *A* composition *is the set of all functions which are conflicting with respect to the same conflict queues.*

These conflicting functions are dependent on each other in the sense that they are connected via at least one common conflict queue in their presets.

 The dataflow part of the FunState model shown in Figure 4.1a) may be regarded as an incomplete specification in the above sense. Let the light-shaded queue q_1 be a conflict queue. Then the conflicting functions f_2 and f_3 form the corresponding composition.

4.6.2 Conflict Resolution

The conflict of a composition is resolved by binding exactly one of the conflicting functions and storing this binding. This is represented by internal states of the schedule specification automaton. Basically, there are two possibilities of resolving a conflict: as soon as possible or just before executing one of the involved functions. The former is called *early* and the latter *late conflict resolution*. In this section, we describe how to generate a schedule specification automaton which models one of these resolution behaviors.

 First, we concentrate on early conflict resolution. A conflict is to be resolved as soon as possible, hence, at the very beginning of the schedule or immediately after executing one of the conflicting functions of the corresponding composition. Figure 4.5 shows a FunState model including the schedule specification automaton which represents this resolution behavior. For simplicity, concurrent state machines are used in the FSM part.

 The concurrent execution of partial state machines within one component and their communication via events is totally synchronous as described in [TTN$^+$98]. Some transition predicates contain the in-state operator $M\ in\ s$ of which the result is $true$ iff the partial state machine M is in its state s. Note that function calls in transition actions such as "$\ldots/f_2$" serve also as events for communication and thus may be part of a transition predicate such as "f_2" as shown in the right FSM part of Figure 4.5.

 The queues q_1, q_2, and q_4 have been specified as conflict queues. This results in the compositions C_1 and C_2. The schedule specification automaton has been generated automatically. Each of the conflicts results in a partial automaton that performs the binding
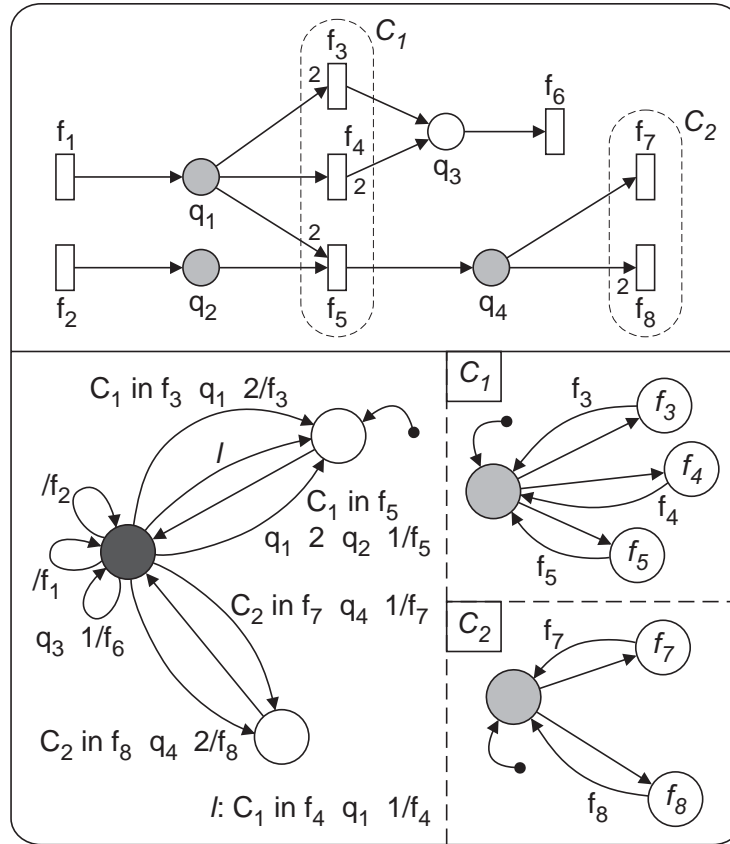
Figure 4.5: Early conflict resolution.

of the conflicting functions. Here, each function is represented by a state reachable from a conflict state. An additional partial automaton is used to control the total schedule. It contains the only alternative state.

A drawback of early conflict resolution is that the corresponding regular state transition graph and the resulting schedule contain many similar paths that proceed "in parallel" and of which the visited states differ only slightly. This results from the fact that the execution paths are splitted very early. It is likely that this increases the size of the search space and the resulting schedule.

Late conflict resolution avoids splitting the execution paths too early as this is done only as late as possible. The conflict is resolved only when at least one of the functions could be executed. Figure 4.6 shows the schedule specification automaton for late conflict resolution replacing that of Figure 4.5.

The events $tryC1$ and $tryC2$ are defined explicitly for communication between the concurrent automata. The transition predicates for generating these events in the left part of Figure 4.6 are based on the disjunction of the firing predicates of all conflicting functions of the respective composition. As long as a conflict is unresolved, the corresponding partial automaton stays in its state $free$.

A disadvantage of late conflict resolution is that for certain kinds of models the generated schedule specification automaton is quite complex, resulting from complex tran-
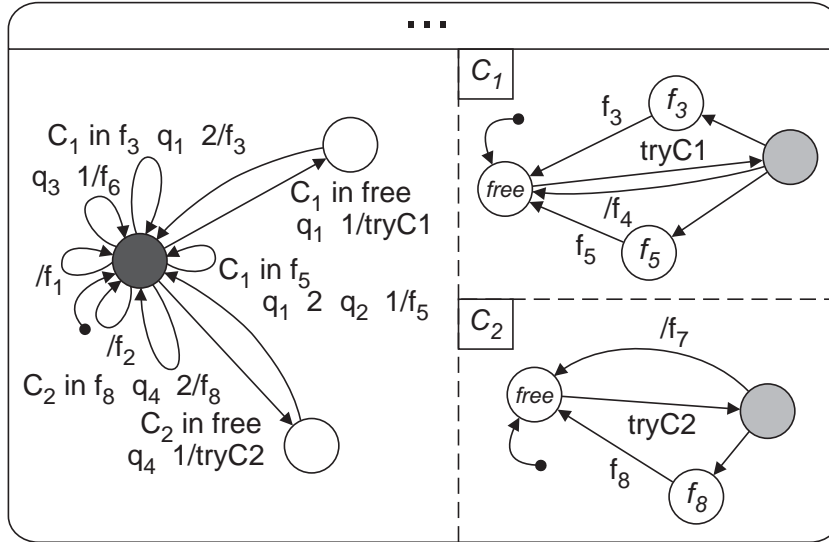
Figure 4.6: Late conflict resolution.

sition predicates and many similar transitions. Investigations comparing the described drawbacks of early and late conflict resolution still have to be made.

4.6.3 Strong Firing Conditions

The schedule specification automaton can be greatly simplified by introducing modified firing conditions for conflicting functions within a composition. By abandoning some freedom during scheduling, the size of the search space may be reduced significantly. The constraint *strong firing conditions* allow for conflict resolution and thus for function execution only when all conflicting functions within a composition are enabled—thus, all predicates on the conflict queue contents with respect to the functions' consumption rates are satisfied.

The effect of strong firing conditions is that the execution of the function chosen is performed immediately after the conflict resolution. Hence, the binding does not have to be stored using an internal state. Figure 4.7 shows the schedule specification automaton representing the strong firing conditions with respect to Figure 4.5.

The predicates of the transitions reaching a conflict state are the conjunctions of the conventional firing predicates of the respective conflicting functions. As some design freedom has been given up, using strong firing conditions may result more often in non-schedulable models than otherwise. Furthermore, resulting schedules may be less efficient.

A special class of models obeying these strong firing conditions are equivalent to free-choice Petri nets where the corresponding consumption and production rates of all conflicting functions within a composition have to be equal. Hence, there is a tight relationship between our approach and that of [SLWSV98] which still has to be investigated in detail.
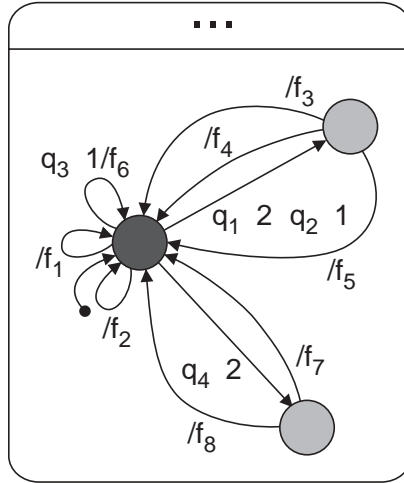
Figure 4.7: Strong firing conditions.

4.7   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 4.8, 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.

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 schedule specification automaton has been given explicitly. The specification automata of more complex systems may be given by a concurrent representation for clearness (omitted here for the sake of simplicity). The result of the symbolic scheduling process—the schedule controller automaton—is shown in Figure 4.9. It replaces the FSM part of the $Host$ component of Figure 4.8. 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.
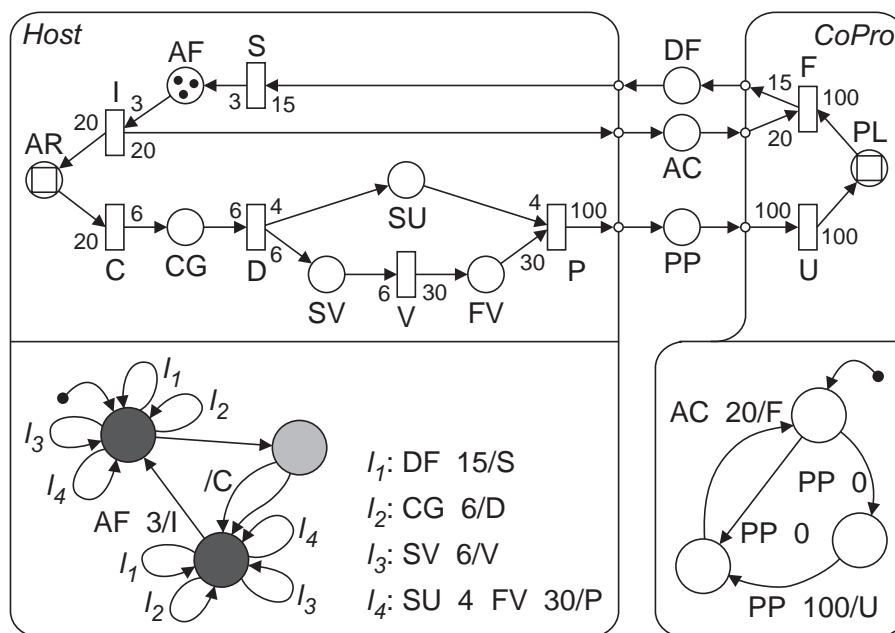
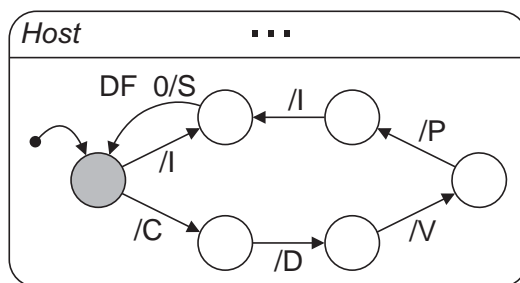Figure 4.8: Molecular dynamics model with specification automaton.

Figure 4.9: Resulting controller automaton.

Chapter 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. The result is a scheduling policy which may be implemented, e.g., as a software controller on a uni-processor.

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. Furthermore, performance investigations are necessary. In particular, the performances of early and late conflict resolution and of the strong firing conditions have to be compared to each other.

Bibliography

[BGJ+97]   F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tab-
          bara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, and
          K. Suzuki. *Hardware-Software Co-Design of Embedded Systems: The Po-
          lis Approach*. Kluwer Academic Press, Boston, 1997.

[Bry86]   R. E. Bryant. Graph-based algorithms for boolean function manipulation.
          IEEE Transactions on Computers, C-35(8):667–691, August 1986.

[Buc93]   J. T. Buck. *Scheduling dynamic dataflow graphs with bounded memory
          using the token flow model*. PhD thesis, University of California, Berkeley,
          1993.

[CD94]   C. N. Coelho Jr. and G. De Micheli. Dynamic scheduling and synchroniza-
          tion 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.

[CTGC95]   M. Cornero, F. Thoen, G. Goossens, and F. Curatelli. Software synthe-
          sis 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.

[Har87]   D. Harel. Statecharts: a visual formalism for complex systems. *Science of
          Computer Programming*, 8, 1987.

[HB98]   S. Haynal and F. Brewer. Efficient encoding for exact symbolic automata-
          based scheduling. In *Proceedings of the IEEE/ACM International Confer-
          ence on Computer-Aided Design (ICCAD-98)*, 1998.

[HL97]   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.

[KD92]   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.

[Lee88]   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.

[LM87]   E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[LP95]   E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.

[McM93]   K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Mic94]   G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill International Editions, New York, 1994.

[RB94]   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.

[SKMB90]   A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1990.

[SLWSV98]   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.

[ST98a]   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.

[ST98b]   Karsten Strehl and Lothar Thiele. Symbolic model checking using interval diagram techniques. Technical Report TIK-40, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, February 1998.

[ST99]   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.

[Str98]   Karsten Strehl. Using interval diagram techniques for the symbolic verification of timed automata. Technical Report TIK-53, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, July 1998.

[TTN+98] 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.

[ZER+98a] 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.

[ZER+98b] 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.