

Symbolic Quasi-Static Scheduling of Actor-Oriented SystemC Models

Jens Gladigau, Christian Haubelt, and Jürgen Teich
 Hardware/Software Co-Design
 Department of Computer Science
 University of Erlangen-Nuremberg, Germany

Abstract—In this paper, we propose a *quasi-static scheduling* (QSS) method applicable to actor-oriented SystemC designs. QSS determines a schedule where several static schedules are combined in a dynamic schedule to reduce runtime overhead. This is done by performing as much static scheduling as possible at compile time, and only treating data-dependent control flow as runtime decision. Our approach improves known quasi-static approaches in a way that it is directly applicable to real world designs, and has less restrictions on the underlying model. The effectiveness of the approach based on symbolic computation is demonstrated by scheduling a SystemC design of a network packet filter.

I. INTRODUCTION

Actor-oriented modeling is commonly used in modern design of embedded systems [10]. In actor-oriented modeling, a system is described by concurrently executed entities (called actors) which communicate among each other only via dedicated channels. SystemC is a system description language convenient for implementing actor-oriented models and widely used when developing embedded systems, typically starting with an abstract model. Further in the design flow, the abstract model is partitioned and individual actors or clusters of actors are mapped to resources of a target architecture. These architectures often consist of processors, capable to execute software, hardware accelerators, memories, etc. When generating software for a processor from actor-oriented designs, a software scheduler is needed. This scheduler is obtained either by computing static schedules (e.g., [1]) or by using dynamic scheduling policies. However, as only limited models such as synchronous or cyclo-static data flow models can be scheduled statically [9], [11], and dynamic approaches (e.g., round-robin) introduce enormous overhead, *quasi-static scheduling* (QSS) seems to be a possible remedy.

Static schedules can be repeated infinitely often without the need of runtime decisions, while dynamic schedules imply runtime decisions before each execution step. In QSS, at *compile time* analyses are applied to reduce runtime decisions to the minimum of data dependent decisions, which naturally are only decidable at runtime. A QSS consists of static scheduling sequences and unavoidable runtime decisions, so called *conflicts*, which select from alternative static sequences. I.e., only inevitable data dependent scheduling decisions are decided at runtime

and static decisions are made a priori (at compile time) where possible. While this naturally results in more complex scheduling code, the runtime overhead is minimized. Besides, guarantees on memory requirements and better execution prediction can be given, which is difficult (or impossible) in presence of dynamic schedules.

To subsume, important properties of quasi-static schedules are: (1) minimized runtime overhead, (2) deadlock freeness, and (3) memory boundness. To further reduce runtime overhead, checking for availability of input data and buffer space can be omitted. In QSS, this is already considered at compile time. In this paper, we propose a quasi-static scheduling approach for actor-oriented SystemC models. More precisely, we will focus on the scheduling of these parts of the model implemented as software on a processing unit. As the result of this procedure, a scheduling automaton is gained, which is used either as scheduler for the software implementation, or can be used to speed up the SystemC simulation.

This paper is organized as follows: The model of computation is described in Section II. To utilize symbolic methods, a symbolic representation of SystemC models is needed. Interval diagrams are used as underlying data structure and are explained in Section III. For the proposed model of computation, a novel quasi-static scheduling approach is introduced in Section IV. In Section V, related work is revised and we differentiate our proposal from these. Finally, we show results from applying our symbolic quasi-static scheduling to an actor-oriented SystemC design, a network packet filter, in Section VI.

II. MODEL OF COMPUTATION

Our goal is a quasi-static scheduling method to find schedules for actor-oriented SystemC models. For this purpose, a well defined model of computation (MoC) represented in SystemC is needed. In order to allow as much automation as possible, we require the SystemC model to be transformed into SystemMoC [5], a SystemC library capable of extracting model information for analyses. Other SystemC related approaches of modeling different MoCs exist, e.g., [6], [12]. In contrast to [12], SystemMoC does not demand modifications of SystemC's simulation kernel and is build upon standard SystemC, but avoids context switching to gain simulation speed.

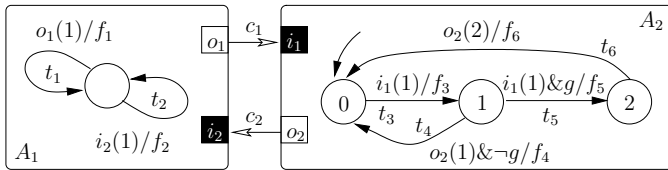


Fig. 1. Two actors A_1 and A_2 communicating via two channels c_1 and c_2 among each other. The communication behavior is defined by finite state machines. On transitions, predicates and actions are annotated.

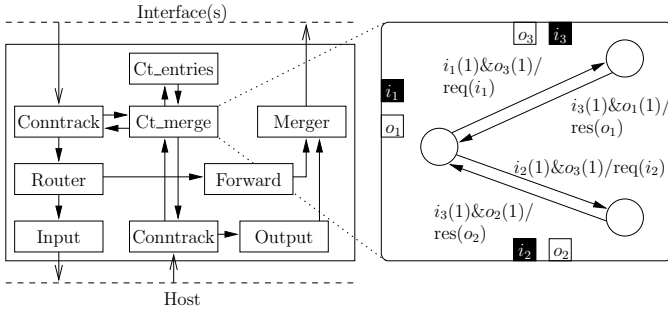


Fig. 2. A SystemC model of a network packet filter. The `Ct_merge` module is zoomed out and its implementation as SystemMoC actor is shown.

In a SystemMoC description, each SystemC module implements an actor which is defined by a finite state machine specifying the communication behavior and methods controlled by this finite state machine. See Fig. 1 for a graphical representation of a simple SystemMoC model. There, two modules A_1 and A_2 , called actors, communicate via two FIFO channels c_1 and c_2 among each other. Their communication behavior is defined by the depicted state machines. On each transition data consumption and production rates according to the associated method f_i are annotated. For example, see the transition t_3 in actor A_2 from state '0' to state '1'. After this transition is taken, one single input data, called token, from the connected channel c_1 is consumed (via port i_1). Methods (like f_3) are executed atomically and data consumption and production is done after computing a method. Constant methods (e.g., g in Fig. 1), called *guards*, are used to test values of internal variables and data in the input channels. Hence, SystemMoC resembles *FunState* (Functions driven by State machines) [18] and also realizes a rule-based model of computation [13]. If the predicates (guards and demands on available tokens/free space) annotated to a transition evaluate to true, this transition is enabled. If more than one transition is enabled, one is chosen non-deterministically for taking and the annotated methods f_i , called *action*, will be performed atomically.

If an application is already available as a SystemC specification, this application can be transformed into a SystemMoC model, if some restrictions apply. E.g., communication is done via SystemC FIFOs and a sole `SC_THREAD` is used. We sketch the transformation for the SystemC module `Ct_merge` from Fig. 2. The figure shows the design of a rule based network packet filter with connection tracking capabilities. Network packets arrive from connected

interfaces or from the host the firewall is running on, and packets are forwarded (or discarded) based on a firewall rule set. The simplified SystemC source code of `Ct_merge` is given next.

```

class Ct_merge: public sc_module {
// ... ports & constructor including SC_THREAD(process);
void process() {
while(1) {
if (o3.num_free() && i1.nb_read(ereq)) {
o3.write(ereq);
i3.read(eres);
o1.write(eres);
}
if (o3.num_free() && i2.nb_read(ereq)) {
o3.write(ereq);
i3.read(eres);
o2.write(eres);
}
}
}
};
    
```

The SystemC module `Ct_merge` dispatches incoming requests for connection tracking entry look ups to the module `Ct_entries` and forwards the corresponding response.

Transformed into an equivalent SystemMoC actor (right side of Fig. 2), the finite state machine controlling the communication behavior checks for available input data and available space on the output channels to store results. Actions can access the values of the SystemMoC FIFOs by help of the bracket operators indicating a read or write offset as shown in the following listing, the implementation of both actions from the SystemMoC actor `Ct_merge`.

```

void req(in_port &in) { o3[0] = in[0]; }
void res(out_port &out) { out[0] = i3[0]; }
    
```

III. SYMBOLIC REPRESENTATION

To reduce the complexity of the data structures and computations, an abstract view of the SystemMoC model is used, which is sufficient for the scheduling task. This abstract view only considers the communication behavior of the model—concrete data values and data transformations performed by actions are neglected. Then, the abstract model state is given by the current fill size of each FIFO channel and the current state of each actor's state machine. For example, the abstract state of the model shown in Fig. 1 is defined by the tuple (q_1, q_2, s) of integers, with q_1/q_2 being the fill size of channel c_1/c_2 , and $s \in S$, $S = \{0, 1, 2\}$, being the state of actor A_2 . The state space is spanned by these integers, i.e., $Q_1 \times Q_2 \times S$. The start state of the example is $(q_1, q_2, s) = (0, 0, 0)$ and taking, e.g., transition t_1 followed by transition t_3 results in the abstract state $(1, 0, 0)$ followed by $(0, 0, 1)$.

Scheduling such models is done by traversing their state space, as we show later. Traversing could be done by enumeration, but due to huge state spaces, this is, in general, prohibitive. We need a way to traverse the state space symbolically to avoid enumeration. Symbolic techniques consider the state transition system implicitly, rather than manipulating state sets directly. In particular, a symbolic representation of SystemMoC models as described above is

needed. This is done by means of characteristic functions of model states and the state transition relation. A state set S is represented by its characteristic function $Z(s)$ with

$$Z(s) = \begin{cases} 1 & \text{if } s \in S \\ 0 & \text{otherwise} \end{cases}.$$

The characteristic function $\delta(s, s')$ of the state transition relation T is defined as

$$\delta(s, s') = \begin{cases} 1 & \text{if } (s, s') \in T \\ 0 & \text{otherwise} \end{cases}.$$

These characteristic functions allow for usage of symbolic functions, called *image* and *preimage*, which are the basis for the symbolic scheduling. We define them similar to [8].

Definition 1 (Image) *The Image(S, T) of a set of states S and a set of transitions T is the set of all states which can be reached from a state $s \in S$ by taking a transition $t \in T$: $\text{Image}(S, T) = \{s' \mid \exists s \text{ with } Z(s) \wedge \delta(s, s')\}$.*

Definition 2 (Preimage) *The PreImage(S, T) of a set of states S and a set of transitions T is the set of all states which can reach a state $s \in S$ by taking a transition $t \in T$: $\text{PreImage}(S, T) = \{s \mid \exists s' \text{ with } Z(s') \wedge \delta(s, s')\}$.*

We use interval diagrams [16] for symbolic encoding, which are similar to Boolean decision diagrams (BDDs) [3]. Instead of Boolean values, a value range is associated to each variable, and on outgoing arcs an interval is annotated. Interval diagrams are well suited for representing transition systems using FIFO communication and detailed description can be found in [16], [17].

IV. QSS OF SYSTEMOC MODELS

After introducing the model of computation for SystemC designs and their symbolic representation in the previous sections, we now introduce the quasi-static scheduling procedure for those models. The basic task while scheduling a model is searching paths through the model's state space. As described in Section III, a single model state consists of the current fill sizes of all queues and the current state of the finite state automata. Note that this abstraction neglects data values. A path $x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} \dots x_j \xrightarrow{t_j} x_{j+1} \dots \xrightarrow{t_{n-1}} x_n$ through the model's state space is a chain of model states x_i and transitions t_j . Taking transition t_j transforms the model state from state x_j to state x_{j+1} . We need a preliminary definition of the term *conflict*: Two transitions leaving the same state of an actor's finite state machine are in conflict with each other, if they have mutual exclusive guards.

The scheduling basically is performed as follows: As a basis, a shortest path $x_0 \xrightarrow{t_0} \dots x_i \xrightarrow{t_i} \dots x_0$ from the model's start state x_0 back to x_0 is searched—taking whatever transitions into account. If a transition t_{c_1} on this path is in conflict with other transitions t_{c_i} , additional paths are searched to cope with every possible outcome of the conflict at runtime. We call these paths alternative paths for t_{c_1} , as they are alternatives for the conflicting transition t_{c_1} encountered at the first place. At runtime,

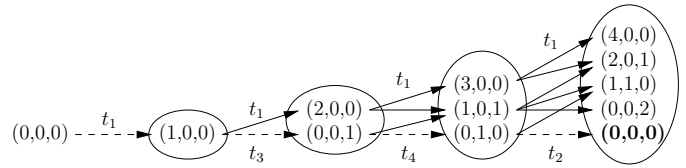


Fig. 3. A sample path search by consecutive image computation. The dotted arrows indicate the found path, the ellipses depict the state sets resulting from image-computations.

one of these paths dependent on the runtime decision is chosen. These alternative paths are shortest paths from the model state the conflict transition t_{c_1} is originating to any known model state, i.e., $x_c \xrightarrow{t_{c_i}} \dots x_k$. A known state x_k is any state on any path found so far. This way, a path is known for every possible runtime decision concerning t_{c_1} . The described conflict handling procedure is recursively applied to every conflict transition encountered on any path. As the result, a clew of paths is obtained, including a path for every possible runtime decision. This clew is a preliminary stage of the scheduler controller automaton, which can be constructed by simplifying the clew [17].

Because queues have limited maximum size and the number of state machine states in actors is limited, too, the models state space is limited. If at some point no alternative path for a conflict transition can be found within these state space boundaries, for the path including the conflict a different (even longer) path is chosen. This backtracking strategy is applied recursively if needed. Therewith, finding a valid quasi-static schedule is guaranteed, if one exists within the state space. Otherwise, the algorithm terminates unsuccessfully.

A. Path Searching

The crucial part of the scheduling algorithm is path searching through the model's state space. This is done symbolically by consecutively computing the image of a single state x_i until a certain model state (e.g., known state) is reached. The image of x_i is the set of all states reachable in a single step. The image of the image of x_i is the set of states reachable from x_i in two steps, and so on. Each set of states reached in n steps is checked for presence of known states. If a known state is reached, at least one path from x_i to this known state must exist. By this symbolic breath-first search, a shortest path from a single state x_i to some known state is found. For the example in Fig. 1, the search for the initial path (from start state $(q_1, q_2, s) = (0, 0, 0)$ back to this state) is shown in Fig. 3. Note that, in contrast to the figure, the exact relation between states, as depicted by arrows, is not known when computing the image of a set of states symbolically. Hence, when a known state is reached, the only thing we know is that at least one path must exist. A backtracking is used to find a concrete path, based on consecutive calculating the preimage of concrete states [17].

The example in Fig. 1 also introduces a class of conflicts we call *multi-rate conflicts*. Due to the guards $\neg g$ and g , the two transitions t_4 and t_5 leaving state '1' of actor A_2

are in conflict. Additionally, t_4 and t_5 have different rates: t_4 demands one token on channel c_1 while t_5 demands one free space in channel c_2 . To cope with such multi-rate conflicts and effectively search for runtime dependent alternative paths, a more advanced path searching algorithm is needed. For multi-rate conflicts alternative paths must cover the conflicting transition *anywhere* on the path, not just at the beginning; it may be impossible to take the conflicting transition because of absence of tokens or insufficient free space for produced tokens. This is the case for the encountered conflict (transition t_4) on the path in Fig. 3. The conflicting transition t_5 is not enabled in model state $(q_1, q_2, s) = (0, 0, 1)$ due to the absence of tokens on channel c_1 . By searching conflict paths including the conflicting transition anywhere on the path, other actors are allowed to produce (or consume) tokens, which may enable the conflicting transition.

For the running example, an alternative path from model state $(0, 0, 1)$ to any known state has to be searched. To resolve the conflict, the path has to cover the conflicting transition t_5 . When searching for paths covering t_5 anywhere on it, a path like $(0, 0, 1) \xrightarrow{t_1} (1, 0, 1) \xrightarrow{t_5} (0, 0, 2) \xrightarrow{t_6} (0, 2, 0) \xrightarrow{t_2} (0, 1, 0)$ is found. (State $(0, 1, 0)$ is known from the first path and therefore a valid end state.) Additionally, transition t_4 must be forbidden on this alternative path search—we search a runtime alternative for its usage at the first place. For this example, the two paths together represent a valid quasi-static schedule and a software scheduler can be derived:

```

while (1) {
  t1 (); t3 ();
  if (g) { t1 (); t5 (); t6 (); t2 (); }
  else { t4 (); }
  t2 ();
}
    
```

This small example shows requirements on the path searching algorithm: While searching alternative paths, some transitions must be disabled. Besides, we must track for all states if some desired transition (or one out of a set of transitions) was used on the path reaching it, only then the path must eventually end in a known state and is an alternative path. Even *unlocking* may be needed to find paths, i.e., if we search an alternative path for usage of t_c , t_c initially is forbidden, but when using one of its conflicting transition on a path, t_c afterwards may be necessary on this path to reach a known state.

To not lose the advantages of symbolic path searching and avoid enumerating possible paths, we modified the image operations in two ways to implement the path searching symbolically: (1) image computation respects a given set of transitions which are disabled while computing the image, and (2) the image functions return, additionally to the set of reachable states, the set of transitions involved in the image computation. Therewith, the advanced path searching, as sketched above, with disabled transitions, unlocking, and splitting/joining state sets, can be implemented efficiently.

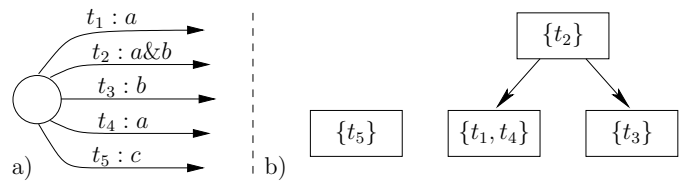


Fig. 4. a) actor state with five leaving transitions. Only the guards—runtime conditions a , b , and c —are depicted. b) the resulting transition graph.

B. Scheduling Algorithm

We now explain the quasi-static scheduling algorithm for actor-oriented models. Here, conflicts that demand runtime decisions are based on the relation of transitions—two or more transitions leaving the same state of an actor's finite state machine can be in conflict. A conflict handling strategy may be: Whenever using a conflicting transition on a path, search paths starting with one of the remaining transitions in conflict, until one path for every transition is found. As shown above, this common conflict handling can not cope with multi-rate conflicts. Furthermore, relations between transitions are ignored, which could be exploited by the scheduling procedure and lead to better results. In the proposed scheduling algorithm, we introduce a novel conflict handling strategy which exploits transition relations and handles multi-rate conflicts.

As a preliminary step, before scheduling, for each state in each actor's state machine all outgoing transitions including guards are analyzed. As the result, for each state a graph is gained, which represents a relation of the transitions leaving this state. As example, in Fig. 4a), a state with five leaving transitions is shown. This is a part of an actor's state machine, simplified to only include guards (runtime decisions) a , b , and c , as they are relevant for the transition graph. The resulting transition graph for these transitions is depicted in Fig. 4b). Transitions with the same guards are put in the same equivalence class (node), and a logical implication of the (Boolean) guards is depicted by arrows. These transition graphs are the basis for conflict handling. If on a path a transition including a guard is taken, the corresponding transition graph is used to determine the minimum number of additional needed paths to guarantee proper execution at runtime: Only one path for each leaf node must be searched. Therewith, alternatives within conflicting transitions and conflict implications are respected.

We give two examples using Fig. 4. Assume a path $\dots x_i \xrightarrow{t_1} \dots x_k$ is found at first, covering transition t_1 with guard a . According to the transition graph, only two alternative paths with transition t_3 and t_5 have to be searched, starting from state x_i . Therewith, for every possible runtime condition paths exist. Now assume the path $\dots x_i \xrightarrow{t_2} \dots x_k$; t_2 is not included in a leaf node. So, we need to find three additional paths, starting from x_i , and covering t_1 or t_4 , t_3 , and t_5 . By this procedure, we find the minimum required paths to guarantee proper execution at runtime; alternatives and implications are exploited.

Together with the advanced path searching (search paths that cover desired transitions *anywhere* on the path), the algorithm also handles multi-rate conflicts.

The outline of the scheduling algorithm is as follows:

```

function find_schedule(start)
2  known = {start}; // known states
   disabled = {}; // disabled transitions
4  (conf,  $x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} \dots x_{n-1} \xrightarrow{t_{n-1}} x_0$ ) =
   find_path(start, known, disabled,  $T_{all}$ );
6  known = known  $\cup$  { $x_1, \dots, x_{n-1}$ };
   initialize schedule with  $x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} x_0$ ;
8  while conf  $\neq$   $\emptyset$  do
   (x,  $t_{chosen}$ , disabled) = conf.pop();
10  leafs = get_leaf_nodes( $t_{chosen}$ );
   leafs = leafs / disabled;
12  while leafs  $\neq$   $\emptyset$  do
   tmp_dis = disabled;
14  add transitions  $\geq t_{chosen}$  to tmp_dis;
   (tmp_conf,  $x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} x_k$ ) =
16  find_path(x, known, tmp_dis,  $T_{leafs}$ );
   known = known  $\cup$  { $x_1, \dots, x_{n-1}$ };
18  add  $x_0 \xrightarrow{t_0} x_1 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} x_k$  to schedule;
   remove covered leaf from leafs;
20  conf = conf  $\cup$  tmp_conf;
   endwhile
22 endwhile
   return schedule;

```

Sole parameter of `find_schedule` is `start`, the start state of the model. The function returns the found paths representing the constructed schedule. The algorithm is simplified and always finding a schedule without backtracking is assumed. First, a path from the model's start state `start` back to `start` is searched (line 5).¹ `find_path` expects four parameters: (1) the originating state of the path, (2) a set of known states (possible end states), (3) a set of disabled transitions, and (4) a set of transitions from which this path has to cover one. As the result, a set of conflict positions `conf` on the found path and the path itself are returned. Transitions on the path that are in conflict with other transitions need further investigation to assure proper operation at runtime, so these occurrences are stored in `conf`. In line 5, T_{all} denotes all transitions of the model, thus, path searching is not limited. The set of known states is updated by subjoining the new states of the path (line 6), and `schedule`, which should contain all found paths and eventually the found schedule, is initialized (line 7). If there are no runtime decisions to make, i.e., the set `conf` is empty after the first path search, a static schedule is returned.

While `conf` contains elements, additional paths for these conflicting transitions have to be searched (lines 8 to 22). Elements of `conf` are tuples of: (1) the state some (2) conflicting transition is originating from and (3) disabled transitions. Function `get_leaf_nodes` returns the set of sets of transitions from the transition graph (line 10). For each leaf, one alternative path has to be searched

¹It is sufficient for the first path search to find a path back to itself, not only to the start state. With it, some initialization phase of the model could be treated.

(lines 12 to 21). These alternative paths respect some disabled transitions (`tmp_dis`) and have to cover one of `leafs`'s transitions (T_{leafs}). If a path is found, the set of known states is updated, the found path is added to `schedule`, the covered leaf is removed from `leafs`, and maybe newly encountered conflicting transitions are added to `conf` (lines 17 to 20). If `conf` is empty, for all conflicts alternative paths have been searched and the clew of paths is returned, which is the basis for the quasi-static schedule. By further reducing this clew of paths, an automaton is the result, which represents the quasi-static schedule for the model and is the basis of a software scheduler.

V. RELATED WORK

In recent years, techniques related to quasi-static scheduling using dataflow specifications [2], restricted Petri nets [4], [7], [14], [15], or FunState [19] were presented. Key difference between these approaches and our proposal for scheduling is the underlying abstract model. Our model can be extracted automatically from an actor-oriented SystemC design and has less restrictions. E.g., whenever an output transition in free-choice Petri nets [14] or equal conflict nets [15] is enabled, all output transitions of the originating place are enabled. A similar approach is taken by Strehl et al. [19] by defining conflict states in FunState models.

The symbolic QSS approach presented in [19] seems to be the most promising one for actor-oriented SystemC designs and we used it as a basis for our proposed scheduling procedure. But there are two major drawbacks with conflict states as used in the FunState approach: (1) each transition leaving a conflict state results in a path search while scheduling (and therewith in at least one transition in the scheduler automaton), and (2) all transitions must have exactly the same demands on input tokens. By the first point, conflicting alternatives (i.e., two transitions leaving a conflict state with the *same* runtime condition in conflict with other transitions) are not respected as alternatives, and conflict implications are ignored. The second point implies that no multi-rate conflicts are allowed. Our approach has neither drawback. We extended it in several ways and adapted it for actor-oriented designs, which enables scheduling of not yet considered model properties (e.g., multi-rate conflicts). Additionally, we eliminated a manual classification step, and search only for the minimum required alternative paths.

As a remedy to construct a monolithic scheduler specification automaton, we discarded the state classification based approach, define the term *conflict* based on relations of transitions, and introduced a novel conflict handling strategy by using transition graphs. This also enables more natural design of state machines without the need of outsourcing conflicts and alternatives in separate states (if this is possible at all). Summarizing, we gain several improvements regarding the FunState method: (1) our methodology starts with SystemC models instead of FunState, which is intended for internal representation of sys-

tems only [18], (2) construction of a monolithic automaton with classified states, which represents the full dynamic schedule, is avoided, (3) conflicts with different rates are considered and can be scheduled in particular cases, (4) transitions instead of states determine conflicts, and (5) by introducing transition graphs, we search only for the minimum required runtime dependent alternative paths by respecting conflict alternatives and conflict implications, which results in lean schedules.

VI. EXAMPLE

So far, the proposed symbolic quasi-static scheduling procedure was introduced by small examples. Next, we apply our scheduling to a real world SystemC example, the connection tracking network packet filter with routing capabilities, shown in Fig. 2. The packet filter consists of nine actors (therein 35 transitions) and 16 channels. Filtering is done in actors *Input*, *Output*, and *Forward*. Based on a rule set, a decision is made to accept or to drop a packet. Connection tracking is done by the *Conntrack* actors.

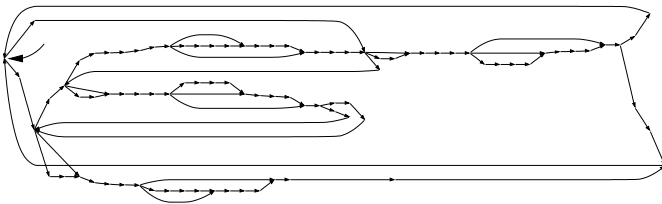


Fig. 5. The found clew of paths for the packet filter (Fig. 2). Only each fork represents a runtime decision, between them, chains of static scheduled transitions are depicted.

We automatically extract the finite state machine for each actor as well as the network graph (interconnection of the actors) from the SystemC program. From this information, the symbolic representations of the state transition relation and the models start state are gained. With a maximum channel depth of four, the models state space consists of about seven million possible states. Scheduling this model takes seconds, including the symbolic encoding of the model. The resulting clew of paths representing the schedule is shown in Fig. 5. It consists of only ten runtime decisions for which 17 alternative paths had to be searched. This is a significant reduction of runtime decisions, as can be seen by the static chains of transitions.

VII. CONCLUSIONS AND FURTHER WORK

In this paper, we introduced a symbolic quasi-static scheduling approach, which is directly and automatically applicable to actor-oriented SystemC designs. By identifying transitions as the origin of conflicts a significant improvement to prior approaches was achieved. The key contributions of this paper are (1) the conflict handling mechanism based on transition graphs, which is an efficient instrument to determine the minimum required alternative paths for conflicts, and (2) handling multi-rate conflicts,

which are not considered in prior approaches. By using a real world SystemC designs, we could show the applicability of the approach. The resulting scheduler reduced the number of runtime decisions significantly and, hence, the scheduling overhead for its software implementation.

In further work, we will use the found schedules not only for generating software schedulers, but also to reduce processing time of the SystemC simulation.

REFERENCES

- [1] S. S. Battacharyya, E. A. Lee, and P. K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.
- [2] B. Bhattacharya and S. S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. In *Proc. of RSP*, pages 84–89, 2000.
- [3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
- [4] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe. Quasi-static scheduling of independent tasks for reactive systems. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 24(10):1492–1514, 2005.
- [5] J. Falk, C. Haubelt, and J. Teich. Efficient representation and simulation of model-based designs in SystemC. In *Proc. of FDL*, pages 129–134, Darmstadt, Germany, Sept. 2006.
- [6] F. Herrera and E. Villar. A framework for embedded system specification under different models of computation in SystemC. In *Proc. of DAC*, pages 911–914, New York, NY, USA, 2006. ACM.
- [7] P.-A. Hsiung and F.-S. Su. Synthesis of real-time embedded software by timed quasi-static scheduling. In *Proc. of VLSID*, pages 579–584, 2003.
- [8] A. J. Hu and D. L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In *Proc. of CAV*, pages 3–14, London, UK, 1993. Springer-Verlag.
- [9] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.
- [10] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.
- [11] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *Proc. of ASILOMAR*, pages 204–210, Washington, DC, USA, 1995. IEEE Computer Society.
- [12] H. D. Patel and S. K. Shukla. *SystemC Kernel Extensions For Heterogenous System Modeling: A Framework for Multi-MoC Modeling & Simulation*. Kluwer Academic Publishers, Norwell, MA, USA, 2004.
- [13] H. D. Patel, S. K. Shukla, E. Mednick, and R. S. Nikhil. A Rule-Based Model of Computation for SystemC: Integrating SystemC and Bluespec for Co-Design. In *Proc. of MEMOCODE*, pages 39–48, 2006.
- [14] M. Sgroi and L. Lavagno. Synthesis of embedded software using free-choice petri nets. In *Proc. of DAC*, pages 805–810, 1999.
- [15] M. Sgroi, L. Lavagno, Y. Watanabe, and A. L. Sangiovanni-Vincentelli. Quasi-static scheduling of embedded software using equal conflict nets. In *Proc. of ICATPN*, pages 208–227, 1999.
- [16] K. Strehl. Interval diagrams: Increasing efficiency of symbolic real-time verification. In *Proc. of RTCSA*, pages 488–491, Hong Kong, 13–15, 1999.
- [17] K. Strehl. *Symbolic Methods Applied to Formal Verification and Synthesis in Embedded Systems Design*. PhD thesis, Swiss Federal Institute of Technology Zurich, Feb. 2000.
- [18] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState—an internal design representation for code-sign. *IEEE Trans. VLSI Syst.*, 9(4):524–544, 2001.
- [19] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich. Scheduling Hardware/Software Systems Using Symbolic Techniques. In *Proc. of CODES*, pages 173–177, Rome, Italy, 3–5, 1999.