Symbolic Scheduling

Abdul-Azeez Olanlokun
Electronic Engineering
Hochschule Hamm-Lippstadt
Lippstadt, Germany
abdul-azeez.olanlokun@stud.hshl.de

***Abstract*—This paper focuses on the developed symbolic scheduling algorithm of an HLS (high-level synthesis) for heterogeneous embedded systems. In the first instance, an internal representation model named FunState is introduced, which allows for the definite representation of nondeterminism and scheduling through the use of functions and state machines. This method of scheduling we introduced can combat mixed data/control flow specifications and takes into consideration various non-determinism processes that emerge in the design of this kind of embedded systems. We should take into account that, Other already implemented components' constraints are honored. By leveraging symbolic scheduling techniques, the scheduling methodology eliminates explicit enumeration of execution routes and guarantees the existence of a deadlock-free and constrained schedule. The resulting schedule is composed of statically scheduled basic blocks that are dynamically invoked during execution.**

***Index Terms*—High Level Synthesis, FunState, Heterogeneous Systems, Symbolic Scheduling**

I. Introduction

According to Webster's dictionary, the verb schedule is defined as "to appoint, allocate, or designate for a specified time." when it comes to the development and design of a digital subsystem, Scheduling is a vital phase that requires optimum carefulness. The goal is to develop a correct implementation that meets design goals such as low execution latency, small silicon area, and low power. The heterogeneity of embedded systems is a primary source of complication in their design. On the one hand, the specification of functional and temporal behavior necessitates using a variety of basic models of computation and communication derived from transformative or reactive domains. Furthermore, we are witnessing an increase in implementation heterogeneity. This does not only apply to functional units, which can be implemented as specialized or programmable hardware, microcontrollers, domain-specific or even general purpose CPUs. Furthermore, these units communicate with one another via various media, such as busses, memory, and networks, as well as a variety of synchronization techniques. The scheduling problem occurs in a wide range of applications, from connectivity to manufacturing ,to high-level digital system synthesis (HLS). Many HLS systems rely on scheduling, which allocates activities to time slots in a synchronous system while taking into account data and control-flow dependencies as well as resource limits. As a result, efficiently tackling this challenge is a direct method to improve the capabilities of these kind of systems. Recently, a framework for dealing with the modeling challenge of complicated embedded systems for scheduling purposes has been established [6, 7]. The model SPI (system property intervals) as stated here is an internal design representation of a design system. It combines the representation of communicative processes with correlated operation modes, non-determinate behavior, various communication mechanisms such as queues and registers, and scheduling constraints.

II. Literature Review

The current study is about a scheduling process that is tailored to this type of internal representation. Different types of non-determinism, as an example:

- specification which are unknown in part(to be settled at the time of design)
- data-dependent control flow (to be settled at run time)
- scheduling policy that is unclear (to be settled at compile time)

and dependencies between design decisions which are for different system components, are common for the design of complex embedded systems. As the number of execution possibilities to be evaluated grows rapidly with ever higher degrees of non-determinism, these qualities need innovative scheduling algorithms. Furthermore, the complexity of computing and communication models raises the risk of system deadlocks or queue overflows (an example can be found in [5]).

The strategy used in this research is based on symbolic techniques that employ a blend of efficient models of state spaces and transition models, as well as symbolic model verification fundamentals, to prevent explicit listing of execution paths. There are some techniques to using symbolic methods to schedule control/data paths for high-level synthesis. BDDs(Binary Decision Diagrams) are used to define scheduling restrictions and solution sets directly or as part of finite state machine (FSM) descriptions.

[6] presents FunState, a common model that unifies several distinct well-known models of computation, provides iterative refinement and hierarchy, and is suitable for representing many alternative synchronization, communication, and scheduling strategies.We describe an approach to symbolic scheduling

based on this model, employing interval diagram techniques. The study specifically describes the following new findings:

- A FunState modification of the SPI model of computation [7, 9] is described, which permits the explicit description of multiple methods of non-determinism and scheduling using a combination of functional programming and state machines.
- A scheduling approach for heterogeneous embedded systems is devised that takes these various types of non-determinism and restrictions imposed by other already implemented components into account and works with mixed data/control flow specification.
- The static block's length and the number of states in the resultant scheduling automaton are optimized
- A hardware/software implementation of a rapid molecular dynamics simulation engine is used to demonstrate the concept.

III. Scheduling and FunState Representation

Common types of representation for mixed control/data-oriented systems have grown in relevance, particularly in the domains of embedded systems and communication electronics. As a result, the FunState formalism, which combines dataflow features with finite state machine behavior, was created[6]. It improves on the SPI model of computation[7, 9] by incorporating internal states for modeling scheduling policies, for example. In the design process, FunState can be utilized as an internal representation.

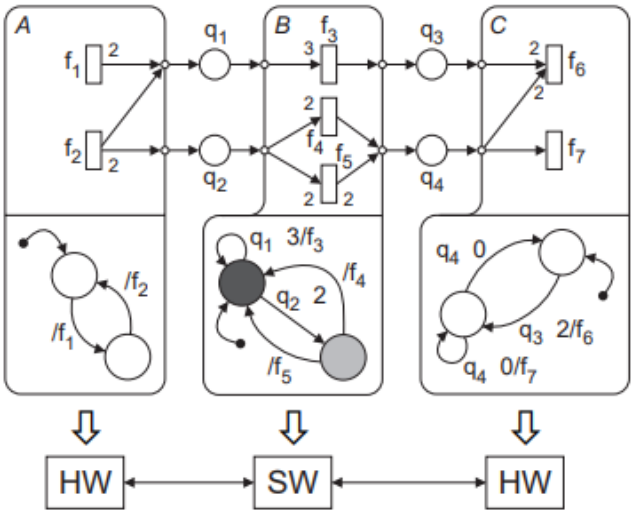

Fig. 1. FunState Model Example[6]

A. Model of Computation in FunState

Only the scheduling features of FunState are discussed in this study. A formal introduction is provided to the reader in [6]. Figure 1 depicts an example of a simple FunState model. It is made up of three portions, each of which has two elements: an upper, data-oriented part that depicts dataflow using functional units (rectangles) and FIFO queues (circles) and a lower, control-oriented part that is defined by a finite state machine.

The data-oriented part's queues hold data items represented by tokens, while the functional units conduct computations on the data. The functions contain consumption and production rates for each linked edge, which are only shown for values other than 1. The execution of each component's functions is controlled by the relevant state machine, which is defined in a statechart-like way.

The state machine labels transitions are combinations of a condition and an action(for example, "$q_1 \geq 3/f_3$"), indicating that the relevant transition, and therefore the action, may be executed only if the condition is met (i.e., if the queue labeled with q_1 has at least three data items). If the preceding transition is used, the function f_3 is performed, consuming three tokens from queue q_1 and producing one token for queue q_3.

In this study, we only employ a basic subset of FunState that is suited for scheduling. While transition predicates might be on the values of data items in general, we only accept predicates on queue contents—the number of tokens in queues. We disregard specified timing characteristics (execution times, timing constraints, etc.). The execution of state machines in its concurrent form from various components is asynchronous and interspersed.

B. The Problem to be Solved

Take a collection of components that are mapped to separate applicability units and communicate via queues in a distributed, concurrent environment. The components provide both data and control flow. Non-determinism may emerge as a result of insufficient requirements or data dependencies that are only addressed at run time. We address the topic of determining a workable schedule for the components mapped onto one applicability unit while respecting limitations provided by other components in this study. In this case, viable indicates that the schedule is devoid of deadlocks and ensures constrained queue contents. Consider a simple scenario to illustrate this. Assume that component B of Figure 1's example FunState model is a processor that transforms datasets between components A and C. Let A and C be hardware components such as an input or output device, or a gateway to a sensor, an actor, or another processor.Let the corresponding state machines specify A and C's behaviors. Not taking these extra limits into account may result in less efficient or even erroneous schedules. A's state machine specifies that its functions are always executed in the sequence $f_1 f_2 f_1 f_2$.... As a result, it is assured that following each firing of f_1, f_2 is also executed accordingly and vis-a-vis. Figure 1 depicts a state machine for B that specifies potential schedules for B. This requirement should be used to create a viable timetable that takes into account the extra information about other components. All transitions that begin in a dark-shaded state are design possibilities that can be selected throughout the schedule preparation process. A light-shaded state, on the other hand, contains a disagreement over its leaving transition. Because the issue can only be addressed at run time, no design decision is feasible.

Conflicts arise, for example, when judgments must be made based on the value of data or environmental factors. In the FunState model, white states contain either just one outbound transition or all transitions have independent predicates. As a result, the transition pattern of these states is predetermined. Because of this, the state with two outbound transitions is determined in component C.Assume that B and C execute often enough (they are "faster" than the prior component) that no infinite number of tokens are present in q_1 and q_2 or q_3 and q_4, respectively. The practicality and validity of the produced schedule are critical issues in schedule development. The specification $(f_4 \mid f_5)(f_4 \mid f_5)...$ which analyzes a possible schedule of B, where f_4 and f_5 are processed alternately and repeatedly while f_3 is ignored. This plan, however, is not practical since the queue contents of q_1 and q_4 are not constrained. suppose we chose $f_3(f_4 \mid f_5)f_3(f_4 \mid f_5)...$ then f_3 will be executed, followed by f_4 or f_5 etc..., this could lead to a faulty behaviour as this could lead to an attempt of f_6 reading too much tokens coming from q_4 after a while.

As opposed to this, a schedule that is feasible and valid in accordance to specification and component of C, is $(f_4 \mid f_5)f_3(f_4 \mid f_5)....$ This schedule can be profited when implemented, with the fact that f_3 could be executed only when there has been an immediate execution of f_4. Because q_1 always includes enough tokens, the behavior of A implies that no condition is required for the execution of f_3. As a consequence, because less queue contents must be established, the resultant schedule may be executed more efficiently by considering just essential requirements.The aforesaid concerns are addressed using the symbolic scheduling approaches given below. The scheduling is achieved logically by replacing dark-shaded states with white states—making decisions and thereby deleting design possibilities. In this study, we solely look into software scheduling on a single processor. Extensions for hardware scheduling with limited resources or scheduling for many processors are simply achievable.

C. Symbolic Methods and FunState

In terms of formal verification, the approaches given in [10] for symbolic model checking of process networks which are based on interval diagram techniques are directly relevant to FunState since the transition behavior of FunState is extremely similar to that of the examined models of computation.Therefore, modeling a mixed hardware/software system with FunState allows for its formal verification, which includes the entire well-known domain of symbolic model checking, which concerns the identification of flaws in design, implementation, or scheduling. The validity of a schedule may be validated by demonstrating the boundedness of the needed memory and the absence of artificial deadlocks. Symbolic approaches based on interval diagram techniques are utilized not only to evaluate but also to construct scheduling strategies for FunState in the scope of this work.

IV. The Interval Diagram Techniques

Interval diagram approaches, such as interval decision diagrams (IDDs) and interval mapping diagrams (IMDs), have proven to be a viable alternative to BDD techniques for formal verification of, for example, process networks [10] or timed automata. This is because, for these types of computing models, the transition relation has a relatively regular structure that IMDs can easily express.While BDDs must explicitly record all potential state variable value combinations before and after a given transition, IMDs just save the state distance—the difference in state variable values before and after the transition. We simply provide a quick, informal overview of the structure and attributes of IDDs and IMDs, as well as the methods necessary for scheduling, in this study.

A. Interval Decision Diagrams, IDDs

IDDs are an extension of BDDs and MDDs, which allow diagram variables to be integers and child nodes to be associated with intervals rather than single values. Figure 2a depicts an example of IDDs. It denotes the Boolean function:
$s(u, v, w) = (u \leq 3) \wedge (v \geq 6) \vee (u \geq 4) \wedge (w \leq 7)$
with $u, v, w \in [0, \infty)$.

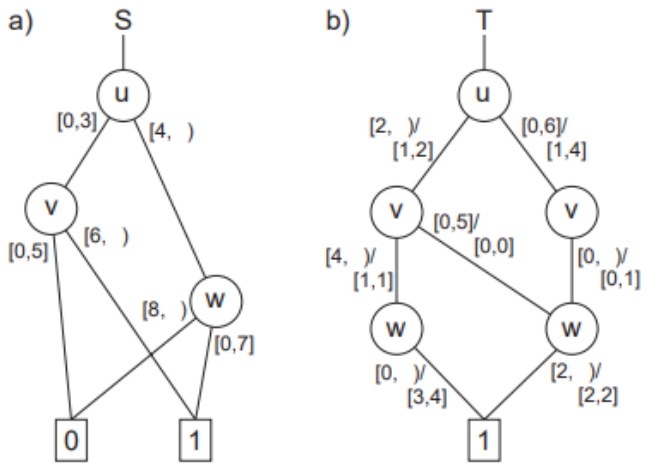

Fig. 2. Interval Decision Diagram and Interval Mapping Diagram[11].

IDDs, which are equivalent to BDDs, have a lowered and ordered form, providing a definitive illustration of a class of Boolean functions—which is essential for accurate fixpoint computations, which are frequently required for validation, as well as for the symbolic scheduling techniques discussed here. During scheduling, IDDs are used to represent state sets.

B. Interval Mapping Diagrams, IMDs

IMDs indicate legitimate state transitions, such as function execution based on boolean on queue contents. IMDs, like IDDs, are represented by graphs. Their boundaries are indicated by functions that relate intervals to intervals. There is just one terminal node in the graph. Figure 2b) depicts a typical IMD. In terms of transition relations, IMDs function as follows. Each edge is identified with a condition, the

logic interval on its source node variable as well as the kind and quantity of change the action operator and the action interval the variable is to go through. Each path illustrates a hypothetical state transition that may be carried out if all of the edges along the channel are activated.

C. The Image Computation

A crucial process for symbolic scheduling approaches is *image computation*, which is similar to formal verification such as symbolic model checking. The set of all states that may be reached after exactly one legal transition from a state in set S is represented by the image $Im(S,T)$ of a set S of system states with regard to transition relation T.An efficient algorithm for performing forward or backward image computation using an IDD S for the state set and an IMD T for the transition connection is described in [10], which result in an IDD $S^{'}$ representing the collection of image state.

V. The Symbolic Scheduling

Symbolic approaches for control-dependent scheduling, for example, [12], have been proved to be useful ways for doing control/data path scheduling. They frequently outperform both ILP and heuristic approaches while producing precise results. Moreover, all feasible solutions to a particular scheduling issue are computed concurrently, allowing for the application of additional restrictions to obtain optimum schedules. In this study, we provide a symbolic way to scheduling FunState models of systems. By utilizing these symbolic approaches, the methodology centered on interval diagram techniques bypasses the explicit cataloging of execution pathways.

A. Scheduling Conflict-Dependent

Quasi-static and related scheduling methods as mentioned earlier, Attempt to combine the benefits of static and dynamic scheduling approach.To do this, data or environment dependent control is resolved at run time, but activities that must be accomplished as a result of a run-time decision are statically scheduled.The goal is to make the majority of scheduling decisions at compile time, leaving only choices at run time that, for example, depend on the quality of data. This second type of run time decision *conflict*, as well as the accompanying scheduling methods, is referred to as *conflict-dependent.* The previous design decisions made at compile time are referred to as alternatives. Because we neglect explicit timing characteristics in the scope of this study, the resultant schedule comprises of sequences of function execution, comparable to scheduling of, for example, marked graphs. The presented FunState model initially includes a schedule specification automaton, which extends the FSM portion to simulate all potential schedule behaviors. This FunState model exhibits entirely dynamic scheduling behavior and is utilized to carry out the symbolic scheduling process described below. This approach yields the schedule controller automaton, which limits scheduling behavior to being just conflict-dependent. This automaton may be used to replace the specification automaton of the initial FunState model, for example, for analysis and verification. Lastly, the controller automaton may be converted into computer code in order to be implemented.

Comparatively, the queue q_1 in Fig 3a) is a multi-reader queue that may include tokens that only one of the queue's readers f_2 and f_3 consumes but the other does not (depending on the token data, for example). Conflicts based on environmental variables may emerge in addition to such data-dependent conflicts.

B. Alternatives and Conflicts

A conflict, in our perspective, is a non-determinism in the specification that cannot be addressed as a design choice but must be taken into consideration during the scheduling. As a result, the multi-reader queue q_4 in Fig 1 does not indicate a conflict since both of the subsequent functions may read all of q_4's tokens regardless of their value or probable external conditions.

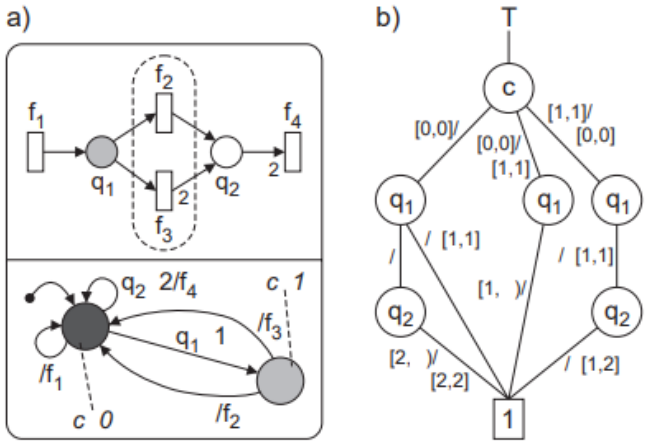

Fig. 3. FunState model of conflict and transition relation IMD[11].

There are three types in the state of FSM Section of the FunState models, light-shaded states which are termed *conflict*States, dark-shaded states which are called *alternative* states, and white states which are called *determinate* states. While a state's determinate attribute is taken immediately from its transition predicates, non-determinate states must be explicitly separated into conflict states and alternative states since both are semantical qualities. All transitions out of an alternative state are design decisions that may be made throughout the schedule creation process. In contrary, all transitions out of a conflict state indicate judgments that may not be made at compile time but retain their non-determinism until at run time. Static are determinate states that have only one outbound transition, as they can only be quitted in one possible way. Determinate states with more than one transition, are called alternative states, and conflict states are called dynamic because they show a dynamic execution behavior with many trails depending on, for example, queue contents or data.

C. Performing The Symbolic Scheduling

The disclosed scheduling process's goal is to sequentialize concurrently supplied functions while maintaining all conflict alternatives. The generated schedule must be free of deadlocks and constrained.

Figure 4 depicts the FunState model's regular state transition graph from Figure 3. It depicts all valid FunState model state transitions in terms of the whole state space, which includes the queue contents of the dataflow component and the discrete system states of the FSM section. Both alternative states of the FSM component are displayed at each coordinate pair of (q_1, q_2).

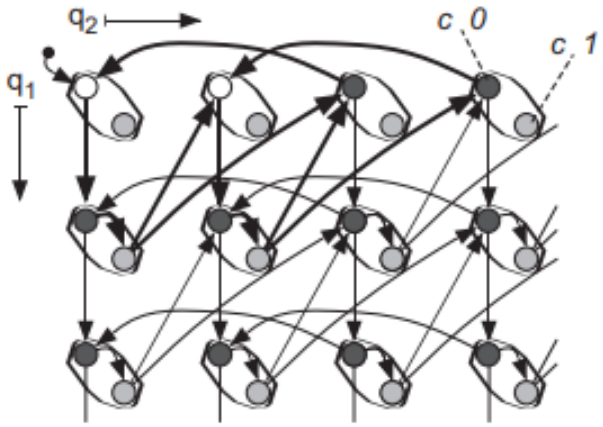

Fig. 4. A Regular state transition graph with schedule[11].

The regular state transition graph is explored symbolically without explicitly creating it using interval diagram techniques. This is accomplished using repeated image computations, as described in the preceding Section. The transition relation is represented by an interval mapping diagram, as shown in Fig 3b), whereas interval decision diagrams are utilized to hold interim state sets. [10] demonstrates the effectiveness of these techniques.

The scheduling technique is illustrated in its most basic form in the accompanying graph. To begin, a symbolic breadth-first search is done to determine the shortest paths from the beginning state to itself or any other state already visited throughout the search. The following scheduling approach is based on one of these (potentially several) shortest paths that represents or contains a cycle. Because no conflict choice may be made during the schedule design, all states of the specified path that relate to conflict states must be investigated later. As a result, starting with the successor states of the conflict states, a breadth-first search is done until reaching any state not yet visited. As previously stated, additional conflict states visited during this search are also treated.

When each successor state of each visited conflict state has been evaluated, the schedule is complete. As a result, each conflict alternative throughout run time is guaranteed to be handled by giving a static schedule until the next conflict to be resolved. Fig 4 depicts the resulting schedule with bold arcs.

If no schedule is obtained when traveling one of the conflict pathways, the scheduling function is repeated using the next shortest path. If all shortest paths are exhausted without yielding a complete schedule, longer paths are chosen. The search space can be limited by imposing a bounding box on the state space. As a result, the algorithm's termination is ensured. Furthermore, if a deadlock-free and constrained schedule exists, the preceding technique will discover it.

D. The Algorithms

The algorithm of $determineShortestPath(A, B)$ is presented in Table I, where the set of states A and B, do not have to be disjoint. As a result, the shortest path $x_0, x_1 ...., x_n$ is one of several possible options.

Section IV.C introduces the image operator $Im(S, T)$ and its inverse $PreIm(S, T)$. Choosing one state from a group of states in Table 1 is ideally accomplished by picking non-conflict states. This is a heuristic criteria for reducing the number of conflicts to examine, and thus the size of the search space and resultant schedule.

The list of P of element x_i is represented by this syntax $< x_0, x_1, ..., x_n >$ which starts with x_0. Common list manipulation functions, including $P.size(), P.head(), P.tail()$, or $P.elements()$ are used.

$determineShortestPath(A, B)$:
$S_0 = A; n = 0;$
do
$S_(n + 1) = Im(S_n, T);$
$n = n + 1;$
until $S_n \cap B \neq \emptyset;$
choose a state $x_n \in S_n \cap B;$
$for\ i = n - 1$ *downto* 0
choose a state $x_i \in S_i \cap PreIm(\{x_{i+1}\}, T);$
return list$< x_0, x_1, ..., x_n >;$

TABLE I
ALGORITHM FOR SHORTEST PATH SEARCH.[10]

initial system state

$determineStateSchedule(a, B)$:
list $P = determineShortestPath(\{a\}, B);$
$B = B \cup P.elements()$:
add P as subgraph to graph G;
while $P.size() > 1$
$x = P.head(); P = P.tail();$
if $isConflictState(x)$
for each $y \in Im(\{x\}, T)$
add $< x.y >$ as subgraph to G;
if $y \notin B$
add subgraph $determineStateSchedule(y, B)$ *to* $G;$
return G;

TABLE II
ALGORITHM TO DETERMINE THE STATE TRANSITION GRAPH OF THE SCHEDULE.[10]

Table II shows an outline of the iterative algorithm $determineStateSchedule(a, B)$, where a is a state and B is a collection of states that may contain a. Because the function

call is via reference, the value of argument B is changed. The output is a directed graph of states reflecting the schedule's state transition graph, as defined below. The recursion is started by using $determineStateSchedule(x_0\{x_0\})$, where x_0 is the current state. For clarity, the building of the resultant graph by adding subgraphs is not discussed in depth in Table II. iff x is a conflict state, then the function value of $isConflictState(x)$ is true.

$determineStateSchedule(a, B)$ algorithm is only roughly sketched. Table II does not include all of the necessary extensions. For example, there is typically no cycle that contains the starting state x_0. Hence, starting with x_0, the regular state transition graph must be explored until a cycle that serves as the foundation for a valid schedule is found. This may be considered a startup phase at the start of the resultant schedule. The $determineStateSchedule(a, B)$ algorithm must be adjusted such that the first shortest path search is replaced by looking for a path from the earlier state to any state reached during this search.

E. The Schedule Controller Generation

As illustrated in Figure 4, the resultant schedule is made up of paths from the regular state transition graph. The controller automaton is generated using the matching subgraph in Fig 5a). As a result of the scheduling process, all alternative states have been replaced by determinate states, resulting in decisions and the elimination of design options. The predicate p identifies the conflict node's run-time decision.

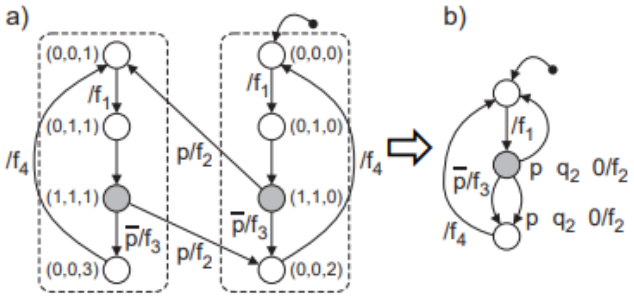

Fig. 5. A State transition graph of schedule for (c, q_1, q_2) and resulting controller automato[11].

This state transition graph might be reduced to decrease implementation effort. Clearly, this process can be motivated by a variety of goals, such as decreasing the number of states in the scheduling automaton or maintaining static node sequences.

For instance, under the constraint that sequences of static nodes are not partitioned, a strategy is presented that reduces the number of states. As a result, the number of dynamic decisions (at run time) in any execution path is not enhanced.

The optimization process is founded on well-known state reduction methods and employs the equivalence relation below:

- The equivalence of two static states is known iff for any input they have similar outputs and the subsequent next states are comparable.
- The equivalence of two dynamic states are considered iif they are of the same kind (conflict, alternative, or determinate) and correspond to the same node in the non-scheduled state machine, for example, they have the same state name but distinct queue contents associated with them.

The above definitions may be used to accomplish iterative state set splitting until only equivalence classes remain. In the case of dynamic states, uncertainty about the following states is handled by adding predicates to the outgoing edges. The result of this method is shown in Figure 5 b). It may be readily converted into pseudo c computer code, as illustrated in Table III.

$a : f_1;$
if p then
$f_2;$
else $f_3;$
$f_4;$
goto $a;$

TABLE III
A CONTROLLER PROGRAM PSEUDO CODE.[11]

F. An Example of Molecular Dynamics Simulation

For a molecular dynamics simulation system, the proposed method was used to conduct conflictdependent scheduling. The reduced core algorithm has been mapped onto a host workstation (Host) that is attached to a special purpose hardware accelerator that serves as a coprocessor (CoPro), as illustrated in Fig 6. The circles holding a square in the diagram indicate data storage registers. As a result, no extra dependence constraints are introduced. For the sake of space, the transition labels $l_1, ..., l_4$ are shown individually.

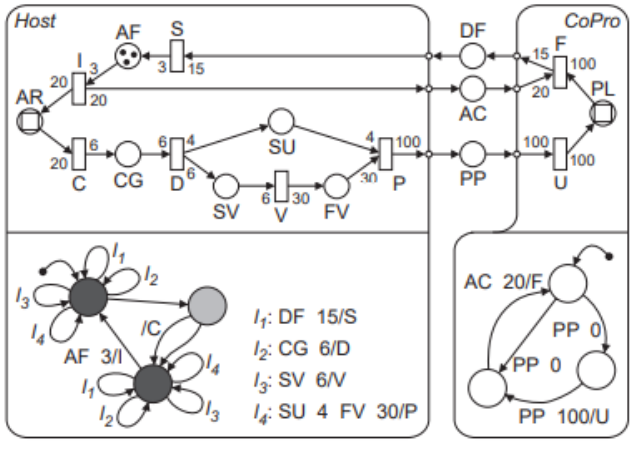

Fig. 6. A Molecular dynamics model with specification automaton.[11].

The simulation is mostly made up of repeated computations in the feedback loop dispersed over both processors, in which

atom forces (AF) are computed (F), summed together (S), and integrated (I) to determine new atom coordinates (AC, AR). Charge groups (CG) are slowly moving sub-molecules whose core coordinates are updated after a configurable number of iterations (C). The pair list (PL) is then computed, which is a new list of neighbors (D, V, P, U).

The following fact indicates a conflict, which is described using a conflict state, because the moment to start this pair list computation is uncertain until run time. The main problem with the schedule is that no cycle in the associated state transition graph that does not have the conflict state in it. The premise that the transition executing *I* cannot be reached without first reaching the conflict state ensures this. Fig 7 shows the schedule controller automaton, which is the output of the symbolic scheduling procedure. It takes the place of the FSM component in Fig 6's *Host* component. It is made up of two static cycles with a conflict state that switches between them. The schedule adheres to CoPro's specifications. Even the CoPro schedule is determined by the content of queue PP because it is dynamic(not static).

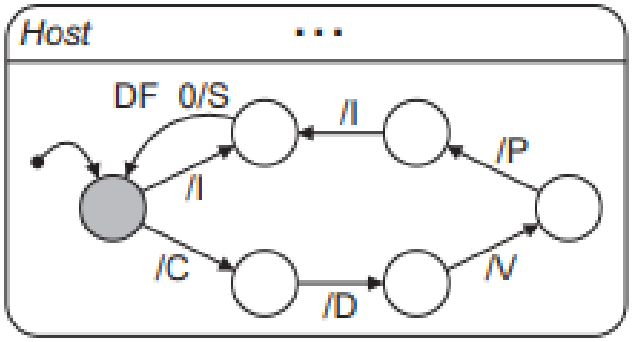

Fig. 7. The Resulting controller automaton[11].

VI. Conclusion

This paper has briefly presented the approach for symbolic scheduling of heterogeneous embedded systems for high-level synthesis, a mixture of hardware/software systems. The approach used was based on FunState Model of computation, which permits the explicit description of multiple methods of non-determinism and scheduling using a combination of functional programming and state machines and the associated scheduling constraints. We have also made used of the Interval diagram techniques to show and model the states transitions as shown in[10] . The final resultant of the symbolic scheduling process shows the generation of schedule controller automaton.

VII. Affidavit

I, Abdul-Azeez Olanlokun, thus certify that I wrote the current paper and work entirely by myself, using only the mentioned sources and aids. Other references to the statement and scope are noted by complete information of the publications concerned; words or sections of sentences cited literally are marked as such. The paper and work in a similar or identical format have never been submitted to an examination board or published. This paper had not yet been utilized in another test or as a course performance, even in part.

References

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

[2] 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 ComputerAided Design (ICCAD-94), pages 175–181, 1994

[3]] 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.

[4] 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.

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

[6] Lothar Thiele, J ¨urgen 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.

[7] 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.

[8] 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.

[9] 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 (ICCAD98), San Jose, California, November 8–12, 1998.

[10] 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.

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

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