Concurrent Execution Semantics and Sequential Simulation Algorithms for the Metropolis Meta-Model

Felice Balarin[1], Luciano Lavagno[1], Claudio Passerone[2],
Alberto Sangiovanni-Vincentelli[3], Yosinori Watanabe[1], Guang Yang[4]

[1] Cadence Berkeley Labs, 2001 Addison St. 3rd Floor, Berkeley CA 94704, USA,
[2] Politecnico di Torino, C. Duca degli Abruzzi 24, 10129 Torino, ITALY,
[3] University of California, Berkeley CA 94720, USA
[4] University of Massachusetts, Amherst MA 01002, USA

ABSTRACT

This paper presents the simulation techniques that are available in Metropolis, an inter-disciplinary research project that develops a design methodology, supported by a comprehensive design environment and tool set, for embedded systems. System behavior is non-deterministic in general, especially in the beginning of the design process, when several key decision, such as the mapping on an implementation platform, have not yet been made, and thus the traces obtainable by simulation are not unique even under the same input sequence. One may want to visit as many traces as possible for regression tests at the final stage of designs, or may just need one valid trace for a quick validation of the design at an early stage. Our techniques can adapt to these different objectives easily. They are also platform-independent in that simulation using different languages, such as SystemC 2.0, Java, and C++ with a thread library, are possible. This feature is important for co-simulation between designs captured in Metropolis and those that have been already designed in other languages.

1. INTRODUCTION

The ability to capture non-determinism is crucial in embedded system specifications. One source of non-determinism is concurrency resolution. System architecture usually involves several components that operate concurrently, and effective exploitation of the concurrency is a key to realizing an efficient implementation. System behavior is also often modeled with concurrency, but this concurrency does not usually match with the concurrency of the architecture. Therefore, how to transform the concurrency of the behavior to the one in the architecture is an important design problem. The concurrency resolution is one aspect of this problem, where an acceptable sequence of events is obtained from a concurrent set of event sequences. Such a sequence is not unique in general, and often remains non-deterministic until information such as scheduling algorithms, execution speed, or arrival time of external events becomes available [1]. This non-determinism defines the solution space that can be explored for finding implementations, and serves as the basis for making critical design decisions such as scheduling policies. Non-determinism is sometimes unavoidable in specifications. For example, depending upon which inputs arrive first, one may want to take different actions. This non-determinism for example arises in digital filters with dynamic updates of coefficients. Non-determinism is also useful to model abstracted behavior of certain components, such as the environment. Since details of the behavior are not specified, the description becomes simpler while the behavior becomes non-deterministic.

Meta-model is a language with a rigorous execution semantics and specification mechanisms, with which this kind of non-determinism can be specified easily [2]. It is used as the internal means to represent designs in the Metropolis design environment. It is designed so that various computation and communication semantics can be specified using common building blocks, and serves as input for simulation as well as for synthesis and verification methods. Essentially, the meta-model specifies netlists of communicating processes, each executing a sequential program. Each process progresses at its own speed, so at any point of time any number of processes can take their next "step", if they are not blocked executing a special synchronization construct called await. Which subset of processes take a "step" at any given point in time is subject to a non-deterministic choice.

Non-determinism poses unique verification problems. Traditionally, systems are verified by simulating their response to a given set of stimuli. However, with non-determinism, there may be many valid responses, and it is impossible to know which one will be exhibited by the final implementation. It is thus not clear what the value of simulation is in this setting, or even what should be the result of a simulation run. One may be tempted to resort to formal verification methods to reason about all possible responses at once, but current formal verification tools are far from being able to completely verify today's systems. Therefore, simulation still plays, and will continue to play, an important practical role when exploring the design space and verifying the overall system behavior. In this paper, we propose an approach that clarifies the role of simulation in the presence of non-determinism. It is based on

CODES'02, May 6-8, 2002, Estes Park, Colorado, USA.

[1] While under certain assumptions concurrent events at outputs can be uniquely resolved for given inputs regardless of scheduling, it is argued that the assumptions hold in only limited cases in recent applications [4].

(i) formal execution semantics of the meta-model that precisely defines acceptable behaviors, and (ii) a generic simulation algorithm that allows exhibiting one of the acceptable behaviors for a given input stimulus. Which behavior is selected, depends on the minor modification of the algorithm. The choice may be driven by different objectives at various levels of abstraction or at various design stages. In the beginning, one may use simulation only to see some sequence of events which is legal with respect to the execution semantics. This is convenient to quickly validate behavior just specified or to find trivial mistakes. Thus, at this stage, one may opt for the modification of the algorithm that optimizes simulation time. At a later stage of the design, one may want to simulate as many legal sequences as possible in order to evaluate the design more thoroughly, approaching the coverage breadth that would be ensured by formal verification techniques; this is often necessary for regression tests. At this stage, one may opt for maximally randomized modification which can exhibit the most behaviors. Also, an interactive version may be appropriate while defining a scheduling policy to be implemented. In this case, one may want to see all the events that can take place at a particular point during a simulation and then to choose interactively one event to see the effect of such a scheduling policy.

Conceptually, the proposed algorithm alternates two phases, where processes run in one phase and the *manager* runs in the other phase. The manager controls the simulation flow. It keeps track of all the events that can legally take place at a given point of the execution, and can use various functions to determine an order of the events. For example, one valid order may be obtained by using a randomized function, which may be useful to hit corner cases. Alternatively, all these events may be displayed to the user so that he can interactively choose one of them at a time in order to evaluate various scheduling policies. Once a scheduling policy is determined, it can be easily specified in the meta-model using a special object called *scheduler*, which is then taken into account in the simulation algorithm to resolve the non-determinism.

The underlying mechanism also makes the simulation algorithm applicable in different platforms. One type of platforms is a configuration of simulation engines. If one chooses a multi-processor simulation platform in order to utilize the concurrent execution as much as possible, the events can be distributed over the processors so that they run concurrently until the next synchronization point is reached by some processor. If on the other hand a single processor is used for the simulation, the concurrent events are fully serialized and the algorithm tries to minimize the number of alternations between the two phases.

The other type of platforms is languages used for simulation. In our approach, instead of generating machine code directly from meta-model descriptions, we translate the meta-model to an executable language, which is combined with the simulation algorithm also implemented in the same language. The actual simulation is then carried out in terms of the resulting language. This feature is important to co-simulate designs captured in the meta-model together with existing designs that have been already specified in other languages. We have tested this approach in SystemC 2.0 [9, 10], Java[1], and C++ with a thread library.

1.1 Related Work

There are many environments in which systems are described as networks of concurrent processes executing sequential code, including SpecC [5], SystemC [9], and Ptolemy [3]. Many of these, attempt to resolve the concurrency uniquely, largely eliminating non-determinism (at least from this source). This approach has some clear advantage for high-level modeling, as simulation results are required to be identical across different simulator runs and even different simulator implementations. However, the approach has some disadvantages when it comes to implementation. Resolving the concurrency in exactly the same way as in the simulator is often prohibitively expensive and unnecessary in the implementation. So in practice behaviors of the implementations are close to, but rarely identical to behaviors of high-level models. This means that implementations cannot be automatically verified. A significant designer's effort is needed to analyze implementation behaviors or compare them with behaviors of high-level models to check that the differences between the two are still acceptable.

In our approach, in contrast, the acceptable differences are precisely defined by formal execution semantics. It is not hard to extract from this definition simulation monitors which may check automatically that behaviors of the detailed implementation are also possible behaviors of the high-level models.

Approaches in Specmen [11] and Verisoft [6] are similar to ours in that they define precisely a set of non-deterministic choices for the system behavior. However, both of this approaches are burdened with significant overhead. In case of Specmen, the overhead is due to the fact that the set of choices is defined by constraints, and finding a legal choice requires solving a set of constraints, which may be expensive. In contrast, in our approach (and also in Verisoft), the set of legal choices follows immediately from the progress of concurrent process. However, unlike Verisoft, we make no attempts to exhaustively search the space of legal choices, eliminating thus the bookkeeping overhead. Our approach takes the best of both worlds: use of non-determinism to precisely define design space available to implementation, and use of efficient conventional simulation for verification.

The rest of this paper is organized as follows. Section 2 describes key aspects of the meta-model and its underlying execution semantics. Section 3 introduces the simulation algorithm and shows how various objectives can be achieved. In Section 4, we describe issues addressed in implementing the algorithm in the three executable languages above. Finally, Section 5 concludes the paper.

2. EXECUTION SEMANTICS

In the meta-model a system is specified as a network of *processes*. Each process is a sequential program, and communicates with other processes through *media*. The semantic domain we use to interpret executions of meta-model netlists is a set of sequences of *observable events*. An observable event is the beginning or the ending of an *observable action*, and observable actions are calls of functions implemented in media objects. While the behavior is defined by observable actions only, we also use other actions to help us define the semantics. This extended set of actions include all the statements of the program.

With each action a we associate two *events*, a^+ indicating the start of an execution of a, and a^- indicating the end. For each process P we define the set of events that contains a^+ and a^- for each action a of P, and a special symbol *nop*, indicating that no events are occuring in P. *Event vectors* are vectors that contain an event for each process in the system. They completely characterize activities in the system at any given point in time.

The execution of netlists evolves through a sequence of *states*. A state of the program consists of two parts. The first is the state of the *memory* which consists of assignments to *state variables*. The second part of the state corresponds intuitively to the program counter. We represent this part of the state with states of *action automata* defined over the alphabet of event vectors. We associate an action automaton with each action. The state of an action automaton indicates whether the corresponding statement or expression is being executed or not. By combining all these states, we can precisely determine the location pointed by the program counter. The transition relation of action automata enforces the proper sequence of the beginning and ending of statement executions.

Even though action automata have a few non-standard features, they can be readily understood based on standard automata definition. Therefore, we introduce action automata here only informally, and refer the interested reader to [2] for the full formal definition.

The syntax and semantics of the meta-model is in great part similar to standard sequential programming languages like C++ or Java. Just like these languages the meta-model has the conditional statement

$$\text{if } (expr) \text{ then } stmt_1 \text{ else } stmt_2 \ .$$

We use an action automaton for this statement (shown in Figure 1) to introduce action automata in general. The action automaton in Figure 1 has the initial state that is not drawn (to reduce the clutter). We assume that any transition without present (next) state is coming from (leading to) the initial state. Transitions of action automata are labeled with expression of the form G/E, where *guard* G denotes a set of states, and E is a set of event-vectors. A transition can occur only if the current global state is in G, and the current event vector is in E. For example, the transition labeled:

$$Vexpr \neq 0/stmt_1^+$$

is enabled if the current value of state variable V_{expr} is not zero. ($Vexpr$ is the memory location where the value of expression is stored.) When the transition occurs, $stmt_1$ must begin. Note that we use a symbol in the alphabet of a process to denote the set of event-vectors such that the component of these vectors corresponding to the process is equal to the symbol. For example, $stmt_1^+$ denotes the set of all event-vectors whose component corresponding to the process taking the action is $stmt_1^+$.

If the transition is unguarded, i.e. if G contains all global states, then we write just E instead of G/E (e.g. transition labeled a^+ in Figure 1). Finally, every state in Figure 1 has a self-loop, which we omit to further reduce the clutter, and put the label of the self-loop directly on the state. It should now be clear, that according to Figure 1, after the conditional statement starts executing, the process can take no other action until the conditional expression starts to execute. When this is finished, one of the two statement is executed next, based on the value of $Vexpr$.

One meta-model construct that is very distinct form sequential programming languages is await. The syntax of await is:

$$\text{await}\{(g_1;T_1;S_1)stmt_1 \cdots (g_k;T_k;S_k)stmt_k\} \ .$$

Statements appearing inside an await, like $stmt_1, \ldots, stmt_k$ above, are called *critical sections*. When an await statement is reached, *one* of the critical sections $stmt_i$ is executed, but only if it is *enabled*. Whether $stmt_i$ is enabled depends on the *guard* g_i, *test list* T_i, and *set list* S_i. The guard g_i is an expression that must be satisfied for $stmt_i$ to be enabled. Lists T_i and S_i specify sets of actions denoted by $[T_i]$ and $[S_i]$ respectively. Details on syntax of test and set lists, and how to determine $[T_i]$ and $[S_i]$ from the syntax are not relevant here (they can be found in [2]), except for the fact that $[T_i]$ and $[S_i]$ can contain only critical sections and observable actions. It is useful to imagine that a reservation flag is associated with each action. A flag can be raised either implicitly, by the action being executed, or explicitly, by specifying the action in the set list of some critical section. The statement $stmt_i$ is not enabled as long as any of the flags in $[T_i]$ is raised, i.e. as long as any of the actions in $[T_i]$ is being executed. Starting $stmt_i$ sets all the flags in $[S_i]$, disabling thus all actions that test any of these flags.

Formally, the semantics of the await statement is given by the action automaton in Figure 2. The key to understanding the semantics of await is understanding the label that marks the start of a typical critical section:

$$True(g_i) \cap \overline{Active([T_i])}/stmt_i^+ \cap \overline{Start([S_i])} \ .$$

The set of global states $True(g_i)$ contains all states s such that executing g_i starting from s may produce a value different from zero, assuming that other processes take no actions during this execution. $Active(A)$ denotes the set of states in which at least one of the actions in A is being executed. Thus, the i-th critical section can start only if g_i may evaluate to a value different from 0 in the current state, and no actions in $[T_i]$ is active. $Start(A)$ denotes the set of event vectors that contain the beginning of at least one action in A. It follows that no action in $[S_i]$ can start executing at the same time as $stmt_i$. Furthermore, the self-loop label in Figure 2 ensure that no actions in $[S_i]$ can start executing until $stmt_i$ finishes its execution.

3. SIMULATION ALGORITHMS

In this section, we present a generic algorithm to simulate a meta-model network. It assumes an underlying multi-threading capability, so that a separate thread can be associated with each process. Media objects are shared between threads. Overall, the simulation consists of two steps. In the first step, we generate simulation code from a meta-model description. This code is in the target language (Java or C++), and includes calls to the simulation manager functions, that we will introduce shortly. The generated code is then compiled, linked with the simulation manager functions, and executed, possibly using an appropriate debugger.

An important parameter of our algorithm is manager's function Yield(set S). Yield takes as argument a set of threads S and selects one or more of them to run. Choosing different selection criteria will lead to different specific simulation algorithms. In addition to selection criteria, the algorithm may also be modified by choosing a set of locations in the code where Yield is called. When presenting our algorithm, we specify only a minimum number of places where Yield must be called. However, at locations where Yield could be called, we specify such a call as a comment. Modifying an algorithm is then as simple as un-commenting these calls.

At the beginning of a design process, it may be appropriate to use a simulator with as few calls to Yield as possible, and to choose a simple Yield that selects a single process. Both of these choices reduce the overhead, resulting in shorter simulation times. At the later design stages, to check as many traces as possible, it makes sense to have as many calls to Yield as possible, and to use a selection criterion that is likely to cover a lot of possibilities. Such a criterion may be based on a random choice, or possibly on a more

$$Care = \{a^+, a^-, stmt_1^+, stmt_1^-, stmt_2^+, stmt_2^-\}$$

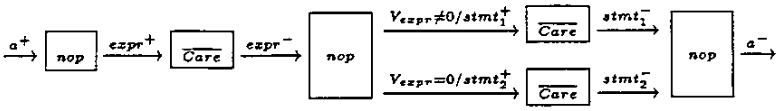
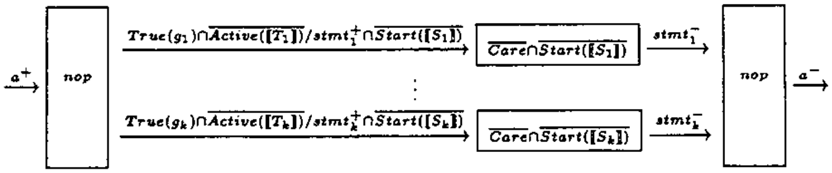


Figure 1: Action automaton $\mathcal{A}[a]$ in case a is if ($expr$) then $stmt_1$ else $stmt_2$.

$$Care = \{a^+, a^-, stmt_1^+, stmt_1^-, \ldots, stmt_k^+, stmt_k^-\}$$



a^+
nop
$True(g_1)\cap\overline{Active([T_1])}/stmt_1^+\cap\overline{Start([S_1])}$
$\overline{Care}\cap\overline{Start([S_1])}$
$stmt_1^-$
$True(g_k)\cap\overline{Active([T_k])}/stmt_k^+\cap\overline{Start([S_k])}$
$\overline{Care}\cap\overline{Start([S_k])}$
$stmt_k^-$
nop
a^-

Figure 2: Automaton $\mathcal{A}[a]$ in case a is await$\{(g_1;T_1;S_1)stmt_1;\cdots(g_k;T_k;S_k)stmt_k;\}$.

intelligent choice that tries to direct simulation to a previously unvisited portion of the state space [7]. Even further on in the design process, when decisions on the implementation platform and its scheduling algorithm have been made, it makes sense to use a version of Yield that models that specific scheduling algorithm. The choice of Yield may be partially driven by the simulation platform as well. For a single-processor platform, Yield that selects one thread may be satisfactory, but choosing Yield that selects multiple threads will make it much easier to take advantage of a multiprocessor platform.

Let us now introduce some notation used by the algorithm. We use *Proc* to denote the set of processes in the system, *Obs* to denote the set of all observable actions, and *Crit* to denote the set of all critical sections. For each action $a \in Obs \cup Crit$ we use $a.proc \in Proc$ to denote the process executing it. In addition, for each critical section $a \in Crit$ we use $a.test \subseteq Obs \cup Crit$ to denote the set of actions specified by the test list associated with a, and $a.set \subseteq Obs \cup Crit$ to denote the set of actions specified by the set list associated with a. Finally, we use $a.guard()$ to denote the function which, when called, evaluates the guard of a in the current state.

During the simulation, the manager maintains sets $Ready \subseteq Proc$ and $Active \subseteq Obs \cup Crit$ and $Blocked \subseteq Obs \cup Crit$. An action is in *Active* if it is currently being executed. An action is in *Blocked* if the execution flow has reached it, but it cannot be executed yet. There are two kinds of reasons why an action may be block. The first is that it cannot start until some other action finishes, due to set list constraints. The other reason is that a guard of a critical section may not be satisfied. A process is in *Ready* if none of its actions is in *Blocked*, or the simulator cannot be sure that actions of the process that are in *Blocked* are still blocked. This happens when an action in *Blocked* no longer has to wait for any other actions to finish. If that action is observable, than it is certainly no longer blocked. However, if that action is a critical section, we still need to re-evaluate its guard to check if it is blocked.

For the most parts, simulation code generation is a simple transformation requiring only minor changes to account for syntactical differences between the meta-model and the target language. The two exceptions are calls to observable actions, and await statements, which we discuss in the following sections.

```
BegObsAction(action $a$) {
  //Yield(UpdateReady());
  if ($Blocking(a) \neq \emptyset$) {
    $Ready = Ready - \{a.proc\}$;
    $Blocked = Blocked \cup \{a\}$;
    Yield(UpdateReady());
    $Blocked = Blocked - \{a\}$;
  }
  $Active = Active \cup \{a\}$;
  //Yield(UpdateReady());
}

EndAction(action $a$) {
  //Yield(UpdateReady());
  $Active = Active - \{a\}$;
  //Yield(UpdateReady());
}

set UpdateReady() {
  $Ready = Ready \cup \{a.proc | a \in Blocked, Blocking(a) = \emptyset\}$;
  return $Ready$;
}
```

Figure 3: Functions BegObsAction **and** EndAction.

3.1 Executing observable actions

Consider a typical observable action f(...);, and let $a \in Obs$ be its unique descriptor. Such a piece of code is replaced with:

BegObsAction(a);
f(...);
EndAction(a);

where functions BegObsAction and EndAction are as defined in Figure 3. The set $Blocking(a)$ contains all active actions which must finish before a can start. If a is an observable action than $Blocking(a)$ contains all active critical sections whose set list specifies a. If a is a critical section, then $Blocking(a)$ contains all active actions specified by its test list. Formally:

$$Blocking(a) = \begin{cases} \{b \in Active \cap Crit | a \in b.set\} & \text{if } a \in Obs \ , \\ Active \cap a.test & \text{if } a \in Crit \ . \end{cases}$$

In Figure 3, to start an observable action, we first check if it is blocked. If that is the case, we update *Ready* and *Blocked* accordingly, and yield the control. The control can return to this thread only if the process gets back to *Ready* when some other process executes UpdateReady. This property is an invariant of our algorithm. It is certainly satisfied by the code in Figure 3, even if all calls to Yield are un-commented. It is also preserved by other simulation manager's functions that we will introduce later. It follows from this property that $Blocking(a)$ is empty after the execution proceed beyond the call to Yield. Thus, we remove a from *Blocked*, put it in *Active* and continue with the call to the observable action. When that call finishes we remove a from *Active* in EndAction.

In addition to calling Yield when an action is blocked, we can also call it at the entrances and exits of BegObsAction and EndAction. However, we assume that manager's functions are executed atomically, except for calls to Yield.[2] This is not a problem if Yield selects a single thread, because there is always a single thread active between two Yield calls. If Yield selects more than one thread, then care should be taken to avoid hazards while accessing *Ready*, *Active*, and *Blocked*.

3.2 Executing critical sections

Consider a typical await statement:

$$\text{await}\{(g_1; T_1; S_1) stmt_1 \cdots (g_k; T_k; S_k) stmt_k\} \ ,$$

and let $c_1, \ldots, c_k \in Crit$ be descriptors of critical sections $stmt_1$, $\ldots$,$stmt_k$. Such an await statement is replaced with the following piece of code:

```
switch(ChooseCritSec({$c_1, \ldots , c_k$})) {
  case $c_1$: $stmt_1$;
          EndAction($c_1$);
          break;
  ⋮
  case $c_k$: $stmt_k$;
          EndAction($c_k$);
          break;
}
```

Our first attempt at specifying function ChooseCritSec is shown in Figure 4. We first check if any of the critical sections is enabled (line 4). If we find an enabled critical section, we activate and return it. Otherwise, we add all critical sections to *Blocked*, remove the current process from *Ready*, and yield the control. When the control returns, we re-check all critical sections, and if we find that one is now enabled we remove all critical sections from *Blocked* (they are no longer candidates for execution). Note that contrary to other calls to Yield, we do not update *Ready* before yielding the control in line 10. The reason is that we want to avoid endless looping where a process is found ready because $Blocking(c)$ is empty for some of its critical sections, but the guard of c is false. To ensure that this does not happen, we update *Ready* in line 1, where we know that some actual progress, which might have cause some state change, have occurred.

```
   action ChooseCritSec(actionSet CritSecs) {
1    UpdateReady();
     // Yield(Ready);
2    while (1) {
3      foreach ($c \in CritSecs$) {
4        if ($Blocking(c) = \emptyset \ \wedge \ c.guard() \neq 0$) {
5          $Active = Active \cup \{c\}$;
6          $Blocked = Blocked - CritSecs$;
           // Yield(UpdateReady());
7          return c;
         }
       }
8      $Ready = Ready - \{p | \forall c \in CritSecs : \ p = c.proc\}$;
9      $Blocked = Blocked \cup CritSecs$;
10     Yield(Ready);
   } }
```

Figure 4: Simplified version of ChooseCritSec.

The algorithm in Figure 4 runs into troubles if guards of critical sections contain calls to functions that include await statement. In general, evaluating a guard may have three outcomes: returning with a value zero, returning with a value different from 0, or blocking trying to execute an observable action or a critical section. More precisely, due to the non-determinism, any combination of these three outcomes might be possible. According to the definition, a global state is in $True(g_i)$ if at least one of the possible outcomes is returning with a value different from 0. Thus, a blocking outcome is in this sense equivalent to returning with a value zero. To address these concerns, we modify ChooseCritSec as shown in Figure 5. The variable *guardNestLevel* tracks the depth of nesting guards. At the top level ($guardNestLevel = 0$) the algorithm in Figure 5 is the same as in Figure 4. At lower levels, instead of blocking and yielding control, ChooseCritSec throws an exception which is then caught at the level above, and treated the same as if the result of $c.guard()$ was zero. The function BegObsAction also needs to be modified, so that at lower levels of nesting it throws an exception instead of blocking. The algorithm in Figure 5 will always find a non-blocking outcome, if one exists. However, it can still effectively under-estimate the set *True* if both zero and non-zero non-blocking outcomes are possible. Fortunately, this is still consistent with the execution semantics. It is not hard to check that the automaton in Figure 2 accepts all sequences generated by a similar automaton in which sets $True(g_i)$ are replaced with their subsets.

4. SIMULATOR IMPLEMENTATIONS

In this section we briefly describe three simulator implementations which can be seen as specializations of the algorithm presented in Section 3. One of the implementations is built on top of multi-threading capabilities of Java, one on top of SystemC, and one on top of Pamela multi-threading library [8].

The syntax of the meta-model closely resembles that of Java, and therefore the translation is often almost straightforward. The Java based simulator automatically translates a meta-model specification into a set of Java classes and interfaces, which can be compiled and run together with a simulation library containing the manager. When a process reaches an interface function call or an await statement, it makes a request to the manager, which is implemented in a separate thread, and then waits for an acknowledge to proceed. The manager collects all requests, chooses a non-conflicting subset and sends an acknowledge to each of the corresponding threads. This

[2] We will describe later how one implementation relaxes somewhat this requirement.

```
    action ChooseCritSec(actionSet $CritSecs$) {
1     if ($guardNestLevel = 0$) UpdateReady();
      // if(guardNestLevel = 0) Yield(Ready);
2     while (1) {
3       $guardNestLevel + +$;
4       foreach ($c \in CritSecs$) {
5         try {
6           if ($Blocking(c) = \emptyset \ \wedge \ c.guard() \neq 0$) {
7             $Active = Active \cup \{c\}$;
8             $Blocked = Blocked - CritSecs$;
9             $guardNestLevel - -$;
              // if (guardNestLevel = 0) Yield(UpdateReady());
10            return c;
            }
11        } catch (blocked) ;
        }
12      $guardNestLevel - -$;
13      if ($guardNestLevel = 0$) {
14        $Ready = Ready - \{p | \forall c \in CritSecs : \ p = c.proc\}$;
15        $Blocked = Blocked \cup CritSecs$;
16        Yield($Ready$);
        } else {
17        throw blocked;
    } } }
```

Figure 5: Complete version of ChooseCritSec.

implementation generalizes the algorithm in Section 3, by allowing multiple actions to be activated simultaneously. To achieve this, the simulation manager synchronizes all process trying to activate an action (in line 6 in Figure 3 or line 7 in Figure 5). Then the manager checks if there are any *set list conflicts* among the chosen action. (Two actions a and b chosen to be activated are in a set list conflict if $a \in b.set$, implying that they cannot begin at the same time.) If conflicts exists, they are resolved non-deterministically.

The SystemC based simulator broadly follows the outline of the Java based simulator, but because the underlying execution engine is more efficient, the simulation times are significantly reduced.

We have developed yet another simulator implementation, using C++ with Pamela run-time library [8] as an underlying platform. This implementation, closely follows the algorithm from section 3, with the minimum number of calls to Yield. Contrary to the other two approaches, there is no separate thread for the simulation manager, and Yield selects only one process. This makes this implementations less appropriate for multi-processor execution, but it minimizes the number of context switches and eliminates the need to check for set list conflicts when activating an action. It is therefore more efficient on single-processor platforms, and suitable for quick validation of designs.

Initial experiments with the three implementations support the expectation that our approach has minimal overhead, in the sense that simulation times for models generated by the three simulators are comparable to simulation time when systems are modeled directly in the target environment (Java, SystemC, or C++).

5. CONCLUSIONS

In this paper, we have proposed simulation techniques employed in Metropolis. They are designed in a generic way, so that different simulation objectives arising in various design stages can be served easily. They can be also targeted to either a multi-processor simulation platform or to a single-processor simulator. Further, simulation can be conducted in different languages, so as to realize co-simulation designs captured in Metropolis and those already designed in other languages. The approach has been implemented and tested in SystemC 2.0, Java, and C++ with a thread library respectively.

6. REFERENCES

[1] K. Arnold and J. Gosling. *The Java programming language*. Addison Wesley, 1996.

[2] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Marco Sgroi, and Yosinori Watanabe. Modeling and designing heterogeneous systems. Technical Report 2002/01, Cadence Berkeley Laboratories, January 2002.

[3] J. Buck, S. Ha, E.A. Lee, and D.G. Masserschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, special issue on Simulation Software Development, January 1990.

[4] E.A. de Kock, G. Essink, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application Modeling for Signal Processing Systems. In *37th ACM/IEEE Design Automation Conference*, June 2000.

[5] D. Gajski, R. Zhu, J. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Press, 2000.

[6] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris*, pages 174–186, January 1997.

[7] C. Norris Ip. Simulation coverage enhancement using test stimulus transformation. *IEEE/ACM International Conference on Computer Aided Design*, pages 127–133, November 2000.

[8] M. Nijweide. The Pamela compiler. Technical Report 1-68340-28(1996)08, Delft University of Technology, August 1996.

[9] Open SystemC Initiative. *Functional Specification for SystemC 2.0*, September 2001. avaliable at *www.systemc.org*.

[10] Open SystemC Initiative. *SystemC Version 2.0 Beta-2 User's Guide*, 2001. avaliable at *www.systemc.org*.

[11] Spec-based verification. http://www.versity.com/resources/whitepaper/technicaLpaper.html.