Priority in Logical Time Partial Orders with Synchronous Relations

Régis Gascon, Julien Deantoni, Jean-François Le Tallec

Priority in Logical Time Partial Orders with Synchronous Relations

Regis Gascon
INRIA Aoste
Sophia antipolis, France
regis.gascon@inria.fr

Julien Deantoni
Universite Cote d'Azur
I3S/INRIA Kairos
julien.deantoni@univ-cotedazur.fr

Jean-François le Tallec
INRIA Aoste
Sophia antipolis, France
jean-francois.le-tallec@inria.fr

***Abstract*—The Clock Constraint Specification Language (CCSL) offers constructs for expressing chronological and causal relations on events of an embedded system. CCSL simulator, TimeSquare allows one to visualize executions of the specified systems by determining step by step sets of synchronously occurring events. When several different sets of events are possible at a given step, the simulator uses a global simulation policy to choose one. However, this mechanism does not consider any priority between events. Inspired by priority in Petri nets, we show how to formally define a priority system supporting possibly synchronous partial orders of events. Both formal definitions and an efficient implementation are presented.**

I. Introduction

The Clock Constraint Specification Language (CCSL) is a declarative formal language [1] initially introduced in the UML MARTE profile [2]. CCSL allows to manipulate *Logical Time* [3], [4] with both asynchronous and synchronous constraints. It is based on the notion of *logical clocks*, which are an ordered set of instants. CCSL defines constraints over these instants to specify partial orders, possibly with synchronous relations between some instants [5]. Globally, it is used to equip engineering models with an abstract representation of their behavior. More precisely it has been used in several domains to formally specify the behaviors of models or the concurrent and timed aspects of language operational semantics, at different abstraction levels [6]–[12]. TimeSquare [13] is a tool that implements the operational semantics of CCSL [14]. It simulates a CCSL specification and proposes a specific total order (*i.e.,* a trace) of, possibly synchronous, event occurrences that satisfies the CCSL specification. The choice between the acceptable ordering of clock instants is done according to a specific simulation policy (e.g., random choice or maximum synchronous instants). CCSL was used to express the behavior of many kinds of models, being hardware, software or a mixed of them. In such models, the notion of exclusive shared resource (*i.e.,* , that can be accessed by only one user at a time) is often required. In CCSL it is successfully modeled by conflicts between the clocks that use the shared resource. However, the operational semantics does not allow to specify the *priority* between the clocks that use the shared resource. More generally, when different clock instants are exclusive (i.e., they can not be synchronous), it is not possible to express which total order must be preferably chosen during a simulation. The notion of priority between CCSL clocks is missing. There exist multiple approaches to define priority in formal languages specifying partial orders. For instance Petri Nets defines a notion of Priority in [15]. However, Petri Net simulations being asynchronous, we cannot reuse their algorithm based on decision graphs. There exist approaches from synchronous languages to define the notion of priority. For instance SyncCharts [16], [17] (or Esterel [18]) defines the notion of priority between transitions outgoing the same state. In this case all the transitions are exclusive and the priority specification is a total order between these transitions. In this specific case the solving mechanism consists in choosing the transition with the highest priority. This is too specific to be reused in the CCSL priority mechanism since clocks are not always conflicting (but some of their instants can, depending on the specification). Additionally, we prefer, like in Petri net to defines priorities by a partial order to avoid being too restrictive. In this paper define the notion of priority on CCSL clocks and we prove a mechanism based on Decision Graph to prune, at each step of the simulation, non desired solutions according to a priority specification. Additionally, we prove the correctness of an algorithm to efficiently implement the priority solving. Finally we implemented the algorithm in the TimeSquare tool so that various domain specific priority mechanisms can be encoded from the primitive notion of priority introduced in this paper. After a quick reminder on CCSL, the proposed priority mechanisms are formally defined and an algorithm to solve the priority is given. Then integration in TimeSquare is quickly defined and related works are described before to conclude.

II. CCSL and TimeSquare

a) Overview of CCSL: CCSL is a formal language first introduced in MARTE profile for UML. It offers a set of constructs to specify causal and timed relations on events. The language is based on the notion of logical clock which is an ordered set of instants. Clocks can be mapped on events, which are an ordered set of event occurrences. Instants do not carry values. CCSL relations define constraints on the instants of different clocks. We introduce in this paper only a subset of the CCSL operators, which is sufficient to illustrate our purpose. However, all the upcoming developments are valid for the full language. For a complete and formal definition of

CCSL, see for instance [19] or [14]. CCSL contains a set of clock relations such as:

- the subclock relation $a \boxed{\subset} b$ which means that the instants of clock a always occur simultaneously with an instant of clock b.
- the coincidence relation $a \boxed{=} b$ which means that there is a one-one correspondence between the instants of a and b
- the exclusion relation $a \boxed{\#} b$ which means that instants of a and b never occurs simultaneously.

The language also allows to build new clocks using clock expressions like the classical *union* and *intersection* operators respectively denoted by $a + b$ and $a * b$, the strict sample operators $a \downdownarrows b$ or the preemption $a \lightning b$. A clock expression defines an implicit clock. The instants of the clock corresponding to the union $a + b$ coincide with an instant of a, of b, or with both and every instant of a and b coincides with an instant of $a + b$. The intersection $a * b$ has a set of instants that can be mapped to the instants of a that coincides with some instant of b. The sample expression $a \downdownarrows b$ represents the subclock of b that ticks if a has occurred at least once since the last tick of b. The preemption $a \lightning b$ coincides with all the instants of a until the first occurrence of b and then never occurs again.

A clock expression can be assigned to a clock name using a clock definition of the form $c \boxed{\triangleq} expr$ and this name can be reused in other expressions or relations. A CCSL specification is a set of clock definitions and clock relations seen as a conjunction of constraints. For simplicity, we have introduced here index independent constraints (that do not depend on the number of past occurrences of clocks). CCSL has also index dependent relations and expressions such that precedence or least/upper bound.

b) Simulation with TimeSquare: The tool TimeSquare [13](http://timesquare.inria.fr) provides a solver to simulate CCSL specifications and to represent the result as VCD (Value Change Dump, IEEE Standard 1364). To do so, TimeSquare identifies clocks to Boolean variables. At every simulation step, a clock ticks if its corresponding variable is set to true. As a consequence, the relations between clocks in a simulation step can be expressed by Boolean formulas between the corresponding variables [14]. For instance, coincidence relation corresponds at every simulation step to an equivalence $a \Leftrightarrow b$ and exclusion to a disjunction $\neg a \vee \neg b$. As far as expressions are concerned, the underlying clock $a + b$ is equivalent to the Boolean expression $a \vee b$ and the intersection $a * b$ to $a \wedge b$. Some CCSL constraints have different interpretations depending on the context. For instance, the preemption $a \lightning b$ is equivalent to a if b has never occurred and is always false after that. The strict sampling $a \downdownarrows b$ corresponds to false if a has not occurred since the last occurrence of the underlying clock and is equal to b otherwise. The operational semantics of TimeSquare introduced in [14] follows the intuition above. The semantics is described by a set of Boolean interpretations of the constraints, possibly depending on the context, and rewriting rules to update the context. We do not develop here this whole semantics. What is important for the understanding of the following proposition is that, at each step the set of constraints is translated into a conjunction of Boolean formula. This formula represents the whole set of acceptable valuation for the clocks with respect to the CCSL specification and the context. The simulator chooses one solution of this formula, which defines the state of the clocks at this step (ticking or not). Then the context is updated by applying the rewriting rules. Note that, when the state of the context is bounded, then TimeSquare can compute the whole state space representing all the acceptable solutions of the specification as a graph. When for a given step the Boolean formula has various solutions, the simulator makes a choice according to a global *simulation policy* chosen by the user. This policy can be for instance a random choice between all possible solutions, or a random choice between the set of maximal solutions (ASAP policy). What is missing is a mechanism to guide the choice according to priority rules between the clocks. Such a mechanism has many natural applications in system design like for instance, when events represent operating system task executions or more generally when they represent the access to a shared resource. The remainder of this paper is illustrated by a minimal example.

III. Simulating priorities in CCSL

In the previous section, we reminded that the state of the clocks at each step (ticking or not) is defined by the solution of a Boolean formula. Applying priority boils down to restrict the set of solutions with respect to priority rules. We want to define a notion of priority inspired from classical priorities in transition systems (see e.g. Petri nets [15]). Informally, in a step "a clock cannot tick while another clock of higher priority can tick". This definition is very natural in asynchronous systems where causality is considered as sequentiality; but it is less intuitive in a synchronous (or in a mixed) paradigm. Because Binary Decision Diagram lacks the notion of sequentiality / causality needed to express the priorities, we transformed the problem in order to obtain, for each step, a sequential process where priorities can be applied. A clock state assignment will be valid if it can be obtained incrementally by a sequence respecting the priority rules. However, in the resulting assignment all the clocks that tick does it simultaneously. This notion of causality in a logical instant is used in several other languages like for instance Esterel [18] or VHDL [20]. Considering this, our notion of priority can be informally understood like this: "in a synchronous step, a clock cannot be valuated while a clock of higher priority has not been valuated first".

A. Decision graph for Boolean formulas

We start by defining a sequential process for choosing an assignment satisfying a given Boolean formula. Let ϕ be a Boolean formula and V_ϕ the variables used in ϕ. Given a variable x in V_ϕ we say that

- x is *fixed* in ϕ iff the value of x in all the solutions of ϕ is always the same,

- x is *undetermined* in ϕ iff it is not fixed.

We denote respectively by F_ϕ and U_ϕ the set of fixed and undetermined variables in ϕ.

According to the definitions above, choosing a solution of ϕ means to choose an assignment for the variables in U_ϕ (we have no choice for the variables in F_ϕ). Because the variables in U_ϕ can be logically linked by ϕ, the order in which variables are assigned is important. We want to consider every assignment orders on the variables of U_ϕ that are assigned the value true (corresponding to the clocks ticking). Then, the next step is to reject the assignments for which the order does not respects the notion of priority as defined at the beginning of the section.

The set of assignments satisfying ϕ can be represented by a *decision graph*, *i.e.,* a set of *nodes* and a set of *edges* between a source and a target node. For each edge, we choose a variable x in U_ϕ and set its value. As the application of priorities depend only on the presence of a clock, we consider only the case where the variable is set to true (the other case will be taken into account later in the acceptance condition). The target node is then the cofactor ϕ_x and we can iterate the process until the set of undefined variables is empty. We recall that the cofactor ϕ_x (resp. $\phi_{\overline{x}}$) of ϕ wrt x is the formula obtained by replacing x by 1 (resp. 0) in ϕ. By extension, given a set of variables X we denote by ϕ_X (resp. $\phi_{\overline{X}}$) the formula obtained by replacing every variable in X by 1 (resp. 0) in ϕ.

Formally, we build an acyclic graph $G_\phi = (N, E)$ such that N is a set of Boolean formulas and E a set of edges labeled by variables of ϕ. The sets N and E are inductively defined as the minimal sets such that $\phi \in N$ (ϕ is called the initial node) and for every $\psi \in N$ and every $x \in U_\psi$ we have $\psi \xrightarrow{x} \psi_x$ belongs to E and $\psi_x \in N$. This construction always terminates since the set of undetermined variables strictly decreases in ψ_x.

Note that for every node $\psi \in N$, each outgoing edge corresponds to the assignment of a variable of U_ψ to true. So all the paths starting from the node ϕ corresponds to solutions where at least one variable of U_ψ is true. If we do not want to forget a valid solution we also have to consider the potential solutions where no variable of U_ψ is true. So, we say that a state $\psi \in N$ is *accepting* iff the valuation that assigns 0 to every undetermined variable of ψ (and the right value to the fixed variables) satisfies ψ. Nodes where all variables are fixed are obviously *accepting* according to this definition. The different assignments represented by the decision graph is thus defined as follows.

Definition III.1. Let ϕ be a Boolean formula; v an assignment on V_ϕ and $v(x)$ the valuation of the variable $x \in \phi$. We say that v is recognized by G_ϕ iff there is a path from the initial node to an accepting node ψ in G_ϕ such that

- (C1) every fixed variable of ψ has the right value,
- (C2) for every $x \in U_\psi$ we have $v(x) = 0$,
- (C3) for every variable x labeling an edge of the path we have $v(x) = 1$.

Actually, the conditions **(C1)**, **(C2)** and **(C3)** define the whole assignment. Indeed, we can show that $U_\psi \uplus F_\psi \uplus L$ is a partition of V_ϕ where L is the set of variables labeling the path from ϕ to ψ. If a variable is chosen in the path then it does not belong to ψ which is obtained by substituting the variables of L by 1 in ϕ. So $V_\psi = V_\phi \setminus L$. Then V_ψ can also be partitioned into U_ψ and F_ψ since a variable is either fixed or undetermined.

We can then establish the correspondence between the set of assignments satisfying a formula and its decision graph.

Proposition III.1. *A Boolean formula ϕ is satisfied by an assignment v on V_ϕ iff it is recognized by the decision graph G_ϕ.*

1) Proof.: By construction, every assignment recognized by G_ϕ satisfies the formula. Let us consider a path from the initial state to an accepting state ψ and L the set of variables labeling the path. The construction implies that $\psi \equiv \phi_L$. By definition, v must assign the right value for the variables that are fixed in ψ (c.f. **(C1)**) $v(x) = 0$ for every $x \in U_\psi$ (c.f. **(C2)**). Since ψ is accepting, we can deduce that v satisfies ψ. Now, if we consider that $\psi \equiv \phi_L$, it must be the case that v satisfies ϕ iff v satisfies $\bigwedge_{x \in L} x \wedge \psi$. Since by condition **(C3)** $v(x) = 1$ for every $x \in L$ this property holds.

Conversely, suppose that an assignment v satisfies ϕ. Let x be a variable of U_ϕ such that $v(x) = 1$. By construction, ϕ_x is a successor of ϕ and we can easily check that v satisfies ϕ_x. We can repeat this operation until we reach a state ψ such that for every $x \in U_\psi$ we have $v(x) = 0$. Since in the operation we only visit formulas that are satisfied by v, v satisfies also ψ. As a consequence ψ is an accepting state because the restriction of v on the variables of ψ assigns 0 to all the undetermined variables. Moreover, it is obvious that v assigns the right value to the variables of F_ψ. Finally, every variable of U_ϕ that is not forced in ψ is equal to 1 iff it labels the path we have built. Indeed, for every variable of $U_\phi \setminus F_\psi$ two cases may arise:

- Either the variable has been chosen in the path and so its value is 1 in v.
- Or the variable has not been chosen and is still undetermined in ψ. Then its value in v is 0 since in ψ every $x \in U_\psi$ is such that $v(x) = 0$ by hypothesis.

So we can conclude. □

As a consequence, choosing a solution of ϕ is equivalent to choose a path in G_ϕ from the node ϕ to some accepting state.

B. Definition of priorities

Now we want to restrict the decision graph so that the remaining assignments respects the priority rules. We define a *priority set* P over a set of variables V to be a strict partial order relation on $V \times V$. Each element of P is called a *priority rule* and is denoted by $x > y$ iff x has priority on y. In this case we also say that x has higher priority than y. By definition P is transitive, irreflexive and asymmetric.

Let P be a priority set on the variables of ϕ. We consider that the set of undetermined variables in the formula associated to each step is the set of clocks that would like to tick. So, applying the priorities remains to prevent at this step to choose

a variable if another variable of higher priority can be chosen instead. For every variable x in V_ϕ, we say that x is a *P-candidate* in ϕ iff it is undetermined in ϕ and no variable of ϕ with higher priority than x in P is undetermined. We denote by $C_{(\phi,P)}$ the set of P-candidates in ϕ.

As a consequence, applying priorities on a decision graph is equivalent to remove the transitions such that the label is not a P-candidate in the formula associated to the state it leaves. We denote this graph by $G_{\phi,P}$ and we define the set of solutions of ϕ respecting the priority rules to be the assignments recognized by $G_{\phi,P}$.

Definition III.2. Let ϕ be a Boolean formula and P a priority set. An assignment v satisfies ϕ according to P iff v is recognized by $G_{\phi,P}$.

Minimum Illustrative Example: We consider the following CCSL specification:

$$a \boxed{\#} b \qquad b \boxed{\#} c$$

This example is encoded by the CCSL semantics into the formula $\phi \equiv (\neg a \vee \neg b) \wedge (\neg b \vee \neg c)$. From this formula, we derived the decision graph of Figure 1. We have added in the labels of nodes the set of undetermined variables of the formulas as well as the set of its P-Candidates. Every state is accepting. If we consider the priority set $P = \{b > a\}$ (*i.e.,* b prevails on a) then the left part of the graph must be removed in order to obtain $G_{\phi,P}$. The reader can notice that we do not force b to be chosen in the solution since both b and c are in $C_{(\phi,P)}$[1].

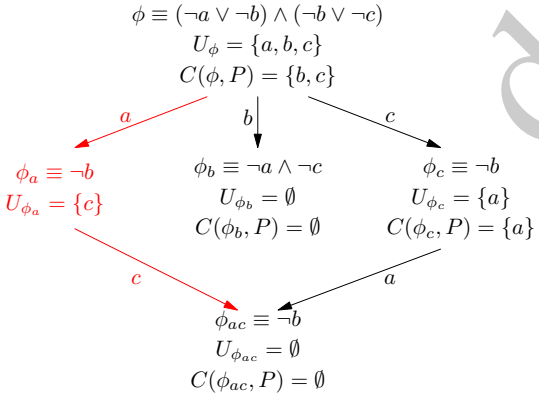

Fig. 1. Decision Graph

C. Algorithm

In practice, it is not convenient to compute and handle the graph $G_{\phi,P}$. We propose to express the restrictions imposed by the priority set on ϕ by another Boolean formula $\Psi(\phi, P)$. The goal is that the set of assignments satisfying $\phi \wedge \Psi(\phi, P)$ will be equal to the set of assignment satisfying ϕ according to P. The gain is also that we will manipulate only Boolean formulas (actually BDDs) in TimeSquare.

[1]In other words, we are not in the SyncChart case where priority specification is a total order including all undetermined clocks and then the resolution mechanism from SyncChart is not appropriate

Informally, $\Psi(\phi, P)$ is defined recursively and follows steps similar to the construction of $G_{\phi,P}$. However, there is a difference in the halting condition. In Figure 1, one can notice that the successors of the node ϕ_c are not needed since no priority rule can be applied (*i.e.,* $U_{\phi_c} = C_{(\phi_c,P)}$). As a consequence, we are sure that all the solution of ϕ_c will be valid wrt the priority set and we can stop the computation. The set of solutions of ϕ associated to such a node is thus the set of solutions of the conjunction of the variables labeling the path from the initial node and the remaining cofactor. Since the cofactor itself is computed by setting the variables of ϕ on the path from the initial node to true, we can actually replace the cofactor by ϕ in the conjunction. We take this remark into account in order to avoid useless computations and combines partial assignments with the initial formula in order to get the set of solutions that respects P.

Formally, the function $\Psi(\phi, P)$ is defined by:

$$\Psi(\phi, P) = 1 \qquad \text{if } U_\phi = C_{(\phi,P)}$$

$$\Psi(\phi, P) = \bigvee_{x \in C_{(\phi,P)}} (x \wedge \Psi(\phi_x, P)) \vee \bigwedge_{x \in U_\phi} \neg x \wedge \phi \qquad \text{otherwise.}$$

The first line states that there is no constraint to impose on ϕ if there is no priority rule to respect. In the second line, the left part of the disjunction corresponds to all the variables that we can choose according to P. We compute recursively Ψ on the cofactor ϕ_x and P. Similarly to the construction of $G_{\phi,P}$, this part only considers the cases where at least an element of $C_{(\phi,P)}$ is true. So we have to add also $\bigwedge_{x \in U_\phi} \neg x \wedge \phi$ to the disjunction, which corresponds to the case where every element of U_ϕ is false (c.f. accepting states in $G_{\phi,P}$). We now prove the correctness of this construction.

Lemma III.1. *For every formula ϕ and priority set P, the set of solutions of $\Psi(\phi, P) \wedge \phi$ is exactly the set of assignments recognized by $G_{\phi,P}$.*

Proof: Suppose that an assignment is recognized by $G_{\phi,P}$. As $G_{\phi,P}$ is a restriction of G_ϕ this assignment must satisfy ϕ by Proposition III.1. We just have to show that v satisfies $\Psi(\phi, P)$.

Consider a path recognizing v. If there is no state ψ in this path such that $U_\psi = C_{(\psi,P)}$ then we consider the final state ψ_f of the path. If v is accepted then ψ_f is an accepting state and v satisfies $\bigwedge_{x \in U_{\psi_f}} \neg x \wedge \psi_f$. Since we are in the case where $U_{\psi_f} \neq C_{(\psi_f,P)}$, we can deduce that v satisfies $\Psi(\psi_f, P)$ (see the last part of the conjunction). If there are states ψ in the path such that $U_\psi = C_{(\psi,P)}$ then we set ψ_f to be the first state of the path verifying this property. Obviously, v satisfies $\Psi(\psi_f, P)$ which is equal to 1 in this case.

Now suppose that a state ψ in the subpath between the initial state and ψ_f is such that v satisfies $\Psi(\psi, P)$. Consider its predecessor ψ' in this subpath and x the P-candidate in ψ labeling the edge from ψ' to ψ. By definition we have $\psi'_x \equiv \psi$ and $v(x) = 1$ because x labels the path recognizing v. Using the induction hypothesis we can deduce that v satisfies

$x \wedge \Psi(\psi'_x, P)$. According to our choice of ψ_f, we know that $U_{\psi'} \neq C_{(\psi',P)}$ so $\Psi(\psi', P)$ is equal to

$$\bigvee_{x \in C_{(\psi',P)}} (x \wedge \Psi(\psi'_x, P)) \vee \bigwedge_{x \in U_{\psi'}} \neg x \wedge \phi.$$

Since $x \wedge \Psi(\psi'_x, P)$ is a disjunct of this formula (x is $C_{(\psi',P)}$), the assignment v satisfies $\Psi(\psi', P)$. By induction, we can then conclude that the state ϕ which is the first node of the path is such that v satisfies $\Psi(\phi, P)$.

Conversely, suppose that an assignment v satisfies $\Psi(\phi, P) \wedge \phi$. We first suppose that $U_\phi \neq C_{(\phi,P)}$ and that there exists $x \in C_{(\phi,P)}$ such that $v(x) = 1$. In $G_{\phi,P}$, ϕ_x is a successor of ϕ and v satisfies ϕ_x. Moreover, by hypothesis v satisfies also $\Psi(\phi, P)$. Since $U_\phi \neq C_{(\phi,P)}$ and $v(x) = 1$ this implies that v must satisfies $x \wedge \Psi(\phi_x, P)$. We can repeat this operation until we reach a state ψ such that $U_\psi = C_{(\psi,P)}$ or for every $x \in C_{(\psi,P)}$ we have $v(x) = 0$.

Suppose that we stop the process in a state ψ where $U_\psi = C_{(\psi,P)}$.We can easily check that for every cofactor ψ_x we will also have $U_{\psi_x} = C_{(\psi_x,P)}$. By induction, this means that the subgraph starting from ψ is actually equivalent to the graph without priorities G_ψ. In the path we have built, we have proved that at each step v satisfies the state. So v satisfies ψ and by Proposition III.1 the restriction of v on V_ψ is accepted by G_ψ. We consider a path accepting the restriction of v in the subgraph corresponding to G_ψ and we append it to the path we have built in the first part. We can prove that this path is accepting for v. The only variables that are in V_ϕ and not in ψ are those that have been chosen in the first part since we have computed successive cofactors. As a consequence:

- **(C1)** and **(C2)** are deduced from the fact that the path from ψ is accepting for the restriction of v on V_ψ
- The value assigned by v to the variables labeling the first part of the path is 1 by construction. For the second part of the path, it is also the case because it is recognized by G_ψ. So condition **(C3)** also holds.

Thus in this case v is recognized by $G_{\phi,P}$.

Now suppose that we stop the process in a step where $v(x) = 0$ for every $x \in C_{(\psi,P)}$ and $U_\psi \neq C_{(\psi,P)}$. At each step, we preserve the fact that v satisfies the successor and the application of Ψ on this successor. So v satisfies $\psi \wedge \Psi(\psi, P)$. According to the definition of $\Psi(\psi, P)$ when $U_\psi \neq C_{(\psi,P)}$ and the fact that $v(x) = 0$ for every $x \in C_{(\psi,P)}$, the assignment v must satisfy $\bigwedge_{x \in U_\psi} \neg x \wedge \psi$. Moreover v satisfies also ψ so it assigns the right values to fixed variables. So we can easily deduce that the state is accepting and that v is recognized by $G_{\phi,P}$. □

By definition, the following corollary then holds.

Corollary III.1. *Given a formula ϕ and priority set P, the set of assignments satisfying ϕ according to P is equal to the set of solutions of $\Psi(\phi, P) \wedge \phi$.*

Note that despite we stop the computation when no priority is needed any more, the computation of $\Psi(\phi, P)$ still contains redundant steps. Indeed, when variables are independent wrt the priority set we may compute all the possible interleavings between these variables but the results will be identical. We are currently investigating ways to improve the efficiency of the algorithm by avoiding this kind of useless computations.

IV. Integration to TimeSquare

The decision graph defined in section III-A to define properly the notion of priority makes explicit all the schedules (*i.e.,* the solutions of a given simulation step) of ϕ with respect to P as paths of the graph. The algorithm proposed in section III-C keeps them implicit as the valuations satisfying $\Psi(\phi, P) \wedge \phi$ and stops when $U_\phi = C(\phi, P)$, so that some clocks are left undetermined. Originally in TimeSquare, the choice of undetermined clocks was done according to a specific *simulation policy*. We want to keep this mechanism the same when priorities are defined. We consequently implemented a software module in TimeSquare that implements the algorithm of section III-C. This module is inserted between the construction of ϕ and its use by the simulation policy. More precisely, since TimesSquare is based on Binary Decision Diagrams (BDD), it takes as input the BDD representing all the solutions of the CCSL specification at a given simulation step and provides in output the BDD restricted according to the priority specification. Note that, by construction the new BDD is a restriction of the previous one.

In our first integration, the priorities are specified by using a textual language. Consequently, the priorities are static along the simulation. The priority language has been defined by using XText(http://eclipse.org/xtext) so that we took benefits from edition facilities (completion, on the fly syntax checking, error report, etc). All the algorithms presented in this paper are already implemented and integrated in the last version of TimeSquare.

User Friendliness: While the definition of priority and the associated algorithm are sound, it can be tedious for a user to define priorities correctly due to the synchronous nature of CCSL. For instance, if we consider example from section III-B extended with a third clock d, which coincides with a:

$$d \boxed{=} a \qquad a \boxed{\#} b \qquad b \boxed{\#} c \qquad \text{b} > \text{a}$$

In this case, at the root of the decision graph, $d \in C(\phi, P)$ and then it can be chosen, implying that a is present while b is not. This behavior is correct but from a user point of view, experiments in our team shown that it is counter intuitive and what users had in mind was a propagation of priority through synchronous relations. For this reason we proposed to programmatically complete the priority specification by propagating the priorities through synchronous relations. We implemented such propagation based on the clock graph already provided by TimeSquare (see [13] section 3.1 for details about clock graph). In this example the inferred priority would be "b > d".

Beyond Static Priority: TimeSquare is an extensible framework where simulation policy and some *backends* can be easily plugged to fit specific domain of use (a backend is a module that can register and react to clock tickings). Additionally,

there is no need in the actual implementation for priorities to be static, they can be changed between each simulation steps without any change in the algorithm. These two features open the way to the definition of existing domain specific dynamic priority mechanisms like EDF (Earliest Deadline First [21]) or more exotic ones. For these reasons, we added to the current CCSL solver an interface that enables setting the priority data structure between each simulation step.

V. Related Work

Our notion of priority is inspired from priority in Petri nets. Petri nets priorities represent a partial order defined locally between transitions. Additionally, a transition not subject to a priority has a priority less than the transition with highest priority. Also, a transition cannot be fired if another transition of higher priority is enabled (*i.e.,* all the input places of this transition have a token). Consider for instance the Petri net in Fig. 2. Transition b has priority on a (represented by the dashed arrow). So a cannot be fired since b is enabled. If we forget the right part of the Petri net, the only solution is to fire b. However, it is not forbidden to fire c. In that case a can be fired *after* c since b is not enabled any more. This does not violate the priority since a cannot be blamed of preventing b in presence of c. Thus the valid marking are those obtained after firing no transition, firing b only, firing c only or firing c and, later, a. The only invalid terminal configuration is the one where a is fired first since it clearly violates the priority. Fig. 2

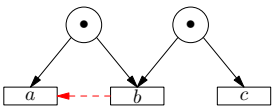

Fig. 2. Petri net with priority

could be represented by the formula $\phi \equiv (\neg a \vee \neg b) \wedge (\neg b \vee \neg c)$ where b has priority on a which we note $b > a$. This is then the same example as in section III-B.

The notion of priority was already studied in synchronous languages like SyncCharts [16] or Esterel [18]. Their definition of priority were different since it represented a total order between *enabled* entities (either transitions in SyncCharts or clocks in Esterel) so that solving priorities was done by choosing the one with the highest priority. However, we were also inspired by the notion of topological sort done in synchronous languages to solve the resolution of schedules in synchronous steps. Somehow, our use of the decision graph or the equivalent algorithm is a way to realize a topological sort dedicated to the resolution of priorities.

VI. Conclusion

This paper formalizes the notion of priority in possibly synchronous partial orders. Applied to CCSL, it defines an algorithm to efficiently solve the priority during the simulation. Such mechanisms are implemented in TimeSquare. Further experimentation can be found on http://timesquare.inria.fr. As a main future work, we plan to implement well known priority based scheduling algorithm as TimeSquare backends and to reify important constructions into a dedicated language to be integrated in TimeSquare.

References

[1] J. Deantoni, C. André, and R. Gascon, "CCSL denotational semantics," Research Report RR-8628, Inria, Nov. 2014.

[2] U. MARTE, "Uml profile for marte: modeling and analysis of real-time embedded systems," 2015.

[3] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, 1991.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[5] F. Mallet, "Clock constraint specification language: specifying clock constraints with uml/marte," *Innovations in Systems and Software Engineering*, vol. 4, pp. 309–314, Oct 2008.

[6] M.-A. Peraldi-Frati, A. Goknil, J. Deantoni, and J. Nordlander, "A timing model for specifying multi clock automotive systems. The Timing Augmented Description Language V2," in *ICECCS 2012 - 17th International Conference on Engineering of Complex Computer Systems*, (Paris, France), pp. 230–239, IEEE, July 2012.

[7] C. Glitia, J. Deantoni, F. Mallet, J.-V. Millo, P. Boulet, and A. Gamatié, "Progressive and explicit refinement of scheduling for multidimensional data-flow applications using uml marte," *Design Automation for Embedded Systems*, vol. 16, no. 2, pp. 137–169, 2012.

[8] A. Goknil, J. Deantoni, M.-A. Peraldi-Frati, and F. Mallet, "Tool Support for the Analysis of TADL2 Timing Constraints using TimeSquare," in *ICECCS'2013 - 18th International Conference on Engineering of Complex Computer Systems*, (Singapore, Singapore), July 2013.

[9] T. Koch, J. Holtmann, and J. Deantoni, "Generating EAST-ADL Event Chains from Scenario-Based Requirements Specifications," in *European Conference on Software Architecture* (P. Avgeriou and U. Zdun, eds.), vol. 8627 of *Lecture Notes in Computer Science*, (Vienna, Austria), pp. 146–153, Springer International Publishing, Aug. 2014.

[10] M.-A. Peraldi-Frati and J. Deantoni, "Scheduling Multi Clock Real Time Systems: From Requirements to Implementation," in *International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*, no. 14th in IEEE international Symposium on Object/Component/service Oriented Real-Time Distributed Computing, (Newport Beach, United States), p. 50; 57, IEEE computer society, Mar. 2011. NewPort Beach.

[11] K. Garcés, J. Deantoni, and F. Mallet, "A Model-Based Approach for Reconciliation of Polychronous Execution Traces," in *SEAA 2011 - 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, (Oulu, Finland), IEEE, Aug. 2011.

[12] F. Mallet, C. André, and J. Deantoni, "Executing AADL models with UML/Marte," in *Int. Conf. Engineering of Complex Computer Systems - ICECCS'09*, (Potsdam, Germany), pp. pp. 371–376, June 2009.

[13] J. Deantoni and F. Mallet, "TimeSquare: Treat your Models with Logical Time," in *TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012* (S. N. Carlo A. Furia, ed.), vol. 7304 of *Lecture Notes in Computer Science - LNCS*, (Prague, Czech Republic), pp. 34–41, Czech Technical University in Prague, in co-operation with ETH Zurich, Springer, May 2012.

[14] C. André, "Syntax and semantics of the clock constraint specification language," Tech. Rep. 6925, INRIA, 2009.

[15] B. Berthomieu, F. Peres, and F. Vernadat, "Bridging the gap between timed automata and bounded time petri nets," in *FORMATS*, vol. 4202 of *LNCS*, pp. 82–97, Springer, 2006.

[16] C. André, "SyncCharts: A visual representation of reactive behaviors," *Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis*, 1995.

[17] C. André, "Computing SyncCharts Reactions.," *Electr. Notes Theor. Comput. Sci.*, vol. 88, pp. 3–19, 2004.

[18] F. Boussinot and R. De Simone, "The esterel language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, 1991.

[19] J. Deantoni, C. André, and R. Gascon, "CCSL denotational semantics," Research Report RR-8628, Inria, Nov. 2014.

[20] IEEE, "Vhdl language reference manual," *Std 1076-1987*, 1988.

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.