Abstractions for Analyzing Decision-Based Process Models

Christophe Damas[1], Bernard Lambeau[1], François Roucoux[2] and Axel van Lamsweerde[1]
[1]Département d'Ingénierie Informatique
[2] Département de Radiothérapie Expérimentale
Université catholique de Louvain (UCL)
{damas, blambeau, avl}@info.ucl.ac.be, francois.roucoux@uclouvain.be

ABSTRACT

Decision-based process models capture processes where the application of specific tasks and their sequencing depend on explicit decisions based on the state of the environment in which the process operates. Decision-based processes in areas such as healthcare are often critical in terms of timing and resources.

The paper presents a variety of tool-supported techniques for analyzing models of such processes. The analyses allow us to highlight inadequate decisions due to inaccurate or outdated information about the environment; violations of temporal constraints; and inadequate usage of resources. The analyses differently instantiate the same fixpoint algorithm for propagating abstract decorations through the model graph.

The language of Guarded High-Level Message Sequence Charts is used for modeling decision-based processes. To enable our analyses, this language is extended with timing constructs, preconditions, and state variables tracking environment quantities. These extensions are grounded on the formal semantic framework of labeled transition systems and fluents introduced earlier.

The techniques are illustrated on the incremental building and analysis of a model for a real protocol in cancer therapy.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Verification – *formal methods.*
D.2.9 [**Software Engineering**]: Management – *process models.*
D.2.5 [**Software Engineering**]: Debugging – *symbolic execution.*

General Terms

Algorithms, Management, Reliability, Theory, Verification.

Keywords

Process modeling, decision-based workflows, model analysis, time-critical properties, labeled transition systems, high-level message sequence charts, fluents, tracking variables, medical safety.

1. INTRODUCTION

The growing maturity of software engineering technologies makes it possible to export them to other areas in need of more systematic approaches. This is in particular the case for modeling and analyzing critical healthcare processes to improve medical safety [Cla08, Dam09]. Conversely, such areas raise new challenges on modeling/analysis technology. For example, cancer therapy processes are composed of safety-critical subprocesses such as radiotherapy, surgery and chemotherapy, to be coordinated over long periods of time, at multiple sites, with multiple decision points, exceptions [Han06], and a variety of constraints including hard time constraints, critical doses, resource limitations, costs, and so forth. Such processes evolve continuously from progress in medical research and practice.

Models in this context may be used for a variety of purposes, e.g., enactment during treatment, reference documentation, instruction for nurses or patients, derivation of useful information for medical or administrative staff, etc. Such models should therefore be as free as errors as possible. Building an adequate, complete, and consistent model is far from easy in such areas. Techniques should therefore help detect and fix severe flaws – in the model being built or in the real process itself. The analyses should be applied to models that are as close as possible to the material provided by the process stakeholders. The latter, in our experience, reason operationally in terms of admissible task sequences that depend on decisions based on the state of the process environment (patient state, device state, etc.). Such workflows are often annotated with a variety of declarative constraints about timing or resource usage.

This paper focuses on *decision-based processes* where decisions relying on the state of the process environment regulate the nature of subsequent tasks and their composition. For example, a specific sequencing of weekly chemotherapy sessions depends on a medical decision relying on environment state variables such as the patient's blood platelet level. Decision-based processes and their timing aspects are obviously found in other application domains, as the meeting scheduling problem benchmark [Fic97], taken as running example in the paper, will suggest.

A decision can be *inadequate* if it relies on incorrect information about the environment. This is typically due to state variables becoming outdated or inaccurate because of intermediate, possibly unexpected events. For example, the blood platelet level used at decision time might not be the patient's actual one. Beyond inadequacy, the modeled process might allow combinations of decisions and subsequent tasks that violate *task preconditions* or *critical constraints* such as time constraints, dose constraints, cost constraints, and the like.

The paper presents a tool-supported approach for detecting such problems. The language of guarded high-Level Message Sequence Charts (g-hMSC) is used for process modeling. This language provides simple notations, close to the informal sketches provided by process stakeholders, while having a formal semantics in terms of labeled transition systems (LTS) [Dam09]. Our analysis tools rely on this semantics and an extended form of g-hMSC allowing us to capture timing constraints, task preconditions, and variables tracking environment quantities.

The checks for inadequate decisions and violations of a variety of constraints are all based on a fixpoint decoration algorithm. This algorithm computes *abstract decorations* for each state of a transition system. It significantly generalizes the fluent-based symbolic execution algorithm in [Dam05] so as to cover a wide variety of checks. For each type of check, we need to instantiate the abstract decoration and the rule for propagating decorations through a single transition. As we will see, the algorithm works for any decoration type having a bounded lattice structure.

The analysis of a g-hMSC proceeds in several steps. A structured form of LTS, called guarded LTS (g-LTS), is first generated to avoid state explosion [Dam09]. The abstract decoration algorithm is then instantiated in order to decorate each state of the g-LTS with information meaningful to the specific check we want to perform. The propagation of concrete decorations by the instantiated algorithms may then highlight process flaws of the corresponding type.

The paper is organized as follows. Section 2 provides some minimal background on LTS, fluents, and g-hMSCs. Section 3 introduces the required extensions to the g-hMSC language, namely, environment tracking variables, task preconditions, and time constraints. Section 4 describes our abstract decoration algorithm. Section 5 shows how task preconditions can be verified or generated. Section 6 presents our technique for detecting potentially inadequate decisions. Section 7 describes a technique for detecting process time overruns. Section 8 shows our approach in action on a real protocol for cancer therapy, where inadequate decisions and violations of time and dose constraints are found in a preliminary version of the model. Section 9 discusses performance aspects of our approach together with related work.

2. BACKGROUND

Running example. We focus on the following episode of a meeting scheduling process [Fic97]. A meeting initiator issues a meeting request, specifying the expected participants and the date range within which the meeting should take place. The scheduler contacts the participants to acquire their constraints. A date conflict occurs when no date can be found that fits all participant constraints. In such case, the initiator may extend the date range or request some participants to weaken their constraints; a new scheduling cycle is then required. Otherwise, the meeting is planned at a date meeting all constraints.

LTS. A labeled transition system (LTS) is an automaton defined by a structure (Q, Σ, δ, q_0), where Q is a finite set of states, Σ is a set of event labels, $\delta \subseteq Q \times \Sigma \times Q$ is a labeled transition relation, and q_0 is an initial state [Mag06]. A system is behaviorally modeled by a parallel composition of LTS models – one for each system agent. The LTS being composed behave asynchronously but synchronize on shared events.

Fluents. A fluent *Fl* is a Boolean proposition defined by a set $Init_{Fl}$ of initiating events making it true, a set $Term_{Fl}$ of terminating events making it false, and an initial value $Initially_{Fl}$ that can be true or false [Gia03]. The sets of initiating and terminating events must be disjoint. A fluent definition takes the form:

$$\textbf{fluent } Fl = \langle Init_{Fl}, Term_{Fl} \rangle \textbf{ initially } Initially_{Fl}$$

A fluent *Fl* holds at some time if (a) *Fl* holds initially and no terminating event has occurred yet, or (b) some initiating event has occurred and no terminating event has occurred since then.

g-hMSC. A *guarded hMSC* is a directed graph where each node is a MSC, a decision node, or a finer-grained guarded hMSC [Dam09]. A *MSC* is composed of vertical timelines associated with agent instances and horizontal arrows representing interactions among them [Dam05]. A *decision node* states specific conditions for the tasks along outgoing branches to be performed. Each outgoing branch is labeled by a Boolean expression on fluents, called *guard*. A guard must be evaluated to true for the corresponding branch to be followed. In simple cases where there are only two branches, such expression may be moved up inside the decision node with 'yes', "no' labels being attached to the corresponding branch. Instead of initial values for fluents, a g-hMSC is given an *initial condition* that constrains the acceptable initial fluent values. This allows us to model processes where different instances can start in different states; initial values for fluents can then be defined at instance level rather than class level. Fig. 1 shows a g-hMSC for the meeting scheduling process.

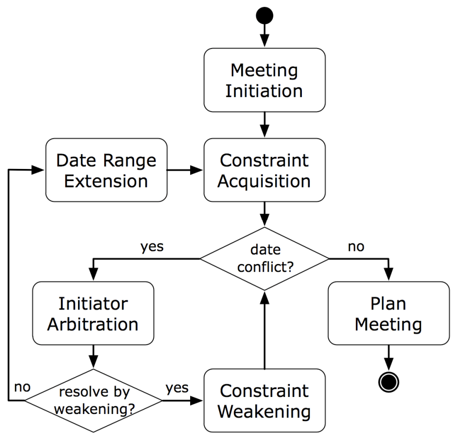

Figure 1. *g-hMSC* for a meeting scheduling process

g-LTS. The semantics of a g-hMSC is defined in terms of guarded LTS, the latter having a trace semantics defined in terms of LTS [Dam09]. A *guarded LTS* (g-LTS) is a transition system where transitions are labeled by events or guards. It is defined by a structure (Q, Σ, Φ, δ, q_0, C_0) where Q is a finite set of states, Σ is a set of event labels, Φ is a set of fluents defined on Σ, $\delta \subseteq Q \mathrm{x} (\Sigma \cup 2^{\Phi}) \mathrm{x} Q$ is a labeled transition relation, q_0 is the initial state, and C_0 is an initial condition (as in g-hMSCs).

For model analysis, a guarded LTS provides a convenient milestone on the way from a g-hMSC to its trace-equivalent LTS, allowing us to compute the set of traces accepted by the g-hMSC. As a structured form of LTS, it avoids state explosion. An algorithm for rewriting a g-hMSC into an equivalent g-LTS is given in [Dam09]. Fig. 2 shows the g-LTS generated from the g-hMSC in Fig. 1. The g-LTS events are MSC ones, e.g., *set_participants*, or special events capturing the *start* and *end* of tasks, e.g., $arbitration_{start}$. Task start/end events are introduced for two reasons:

- to support more accurate fluent definitions where the begin and end of a task can be distinguished,
- to enable g-LTS generation and analysis even when all tasks have not been refined yet into MSCs or finer-grained g-hMSCs (for example, the MSC events for meeting initiation are the only ones to appear in Fig. 2).

The semantics of g-LTS is defined in terms of event traces involving no guards at all – that is, pure LTS. Let *G* denote the g-LTS (Q, Σ, Φ, δ, q_0, C_0). A trace of *G* from q_0 is a pair (*Init*, $<l_0,\ldots>$) where *Init* is an initial fluent value assignment, mapping every fluent in Φ to *true* or *false*, and $<l_0,\ldots>$ is an infinite sequence of labels, some denoting g-LTS events and others denoting guards. Such trace is accepted by *G* from q_0 iff the following acceptance conditions are met for every *i*:

- *Trace inclusion*: $\exists\, q_{i+1} \in Q$ such that $(q_i, l_i, q_{i+1}) \in \delta$
- *Admissible start*: Init $\models C_0$
- *Guard satisfaction*: $S_i \models l_i$ if $l_i \in 2^{\Phi}$,

where S_i is the fluent value assignment after the i-th event in the trace (with $S_0 = Init$).

The trace-equivalent LTS is generated from a g-LTS by composition of a super LTS meeting the trace inclusion condition, an initializer LTS to meet the admissible start condition, and one fluent automaton per fluent to meet the guard satisfaction condition – see [Dam09] for details. Such generation is needed for model-checking against temporal logic properties. The analyses in this paper are performed at g-LTS level, thus avoiding the inefficiency problems inherent to LTS expansion.

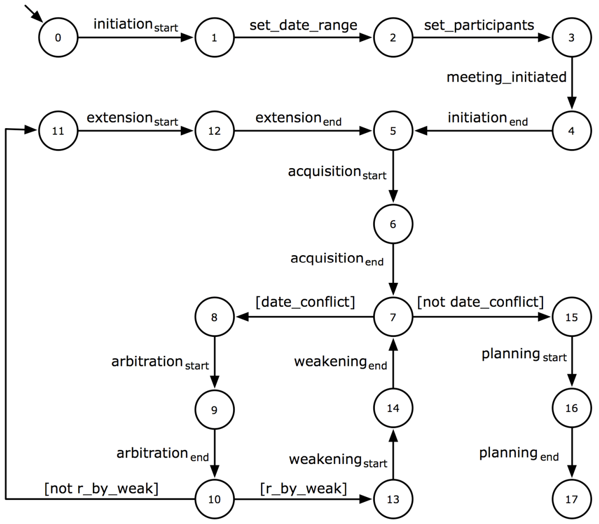

Figure 2. Guarded LTS for meeting scheduling

3. Increasing Model Expressiveness

Our experience in assembling clinical process fragments supplied by medical staff highlighted three modeling constructs not supported by g-hMSCs. This section introduces environment tracking variables for reference in decision nodes, task preconditions, and timing constructs. The g-hMSC trace semantics is adapted accordingly to account for these extensions.

3.1 Environment Tracking Variables

The variables appearing in the decision nodes of a g-hMSC process model dictate which paths are to be followed in specific process instances. Such variables in general refer not only to fluents but also to specific states of the environment in which the process operates.

Tracking variables are state variables tracking environment quantities that are not directly observable by the agents involved in the process. Such quantities are generally changing over time. The values of tracking variables need therefore be updated regularly to accurately reflect the corresponding environment quantity; decisions are based on current values of tracking variables rather than their actual, unobservable counterpart. Examples of tracking variables in medical processes include the patient's temperature or the white blood cell level kept in the patient record and updated when specific clinical tests are performed.

Fluents are convenient for capturing conditions that refer to event occurrences. A fluent-based encoding of a tracking variable in the process model would however be awkward; it would require adding special events making the fluents true or false after every occurrence of events resynchronizing the variable with its environment counterpart. The addition of such special events would make process models unreadable and unmanageable. A new kind of modeling variable is therefore introduced.

A *tracking variable* is defined by the set of events changing its value:

trackingVar *tV* = {*UpdateEvents*}

In contrast with fluents, such update events do not define the value of the tracking variable – only the fact that the value may have changed. For example, the following tracking variable is worth considering for meeting scheduling:

trackingVar date_conflict ={$\text{acquisition}_{\text{end}}$, $\text{weakening}_{\text{end}}$}

This variable declaration states that a date conflict can appear or disappear whenever meeting constraints are acquired or weakened. Unlike fluent declarations, it does not bind events to each of the two cases.

The set of tracking variables attached to a process model will be denoted by Θ. Their sort is assumed in this paper to be Boolean. (For tracking variables with numerical values, this restriction requires modelers to introduce mode abstractions à la SCR, e.g., high_fever for patient temperature, low_platelet for platelet level.)

The formal semantics of tracking variables is defined in terms of LTS. To achieve this, the LTS composition introduced at the end of Section 2 for mapping a g-LTS to its trace-equivalent LTS is extended as follows. For each tracking variable *tV*={*UpdateEvents*}, two automata are added in the composition:

(a) A fluent automaton for the fluent *f-tV*=<@*tV*, @¬*tV*>, where @*tV* and @¬*tV* denote *tV*'s update to *true* and *false*, respectively. This automaton is added to meet the *guard satisfaction* condition in case of guards containing tracking variables.

(b) An updater automaton to account for the fact that the fluent *f-tV* may change to either *true* or *false* after each update event associated with the tracking variable. Formally, this automaton is defined as follows:

$Q = \{q_{start}, q_1\}$
$\Sigma = \Sigma_{g\text{-}lts} \cup 2^{\Phi+\Theta} \cup \{@tV, @\neg tV\}$
$\delta = \{(q_{start}, e, q_{start}) \mid e \in ((\Sigma_{g\text{-}lts} \cup 2^{\Phi+\Theta}) \setminus \mathit{UpdateEvents})\}$

$\cup \{(q_{start}, \text{update}, q_1) \mid \text{update} \in \mathit{UpdateEvents})\}$
$\cup \{(q_1, @tV, q_{start}), (q_1, @\neg tV, q_{start})\}$
$q_0 = q_{start}$

Moreover, all references to the tracking variable in guards and in the initial condition C_0 are replaced by references to the corresponding fluent *f-tV*. The latter fluent is also used in the temporal properties we want to model-check [Dam09]. Fig. 3 shows the two automata to be added in the LTS composition for the date_conflict variable involved in meeting scheduling (the "*\" prefix designates the set of all events except those listed).

Note that this encoding of tracking variables through fluents is not visible from the g-hMSC and g-LTS levels; it is kept hidden to process modelers.

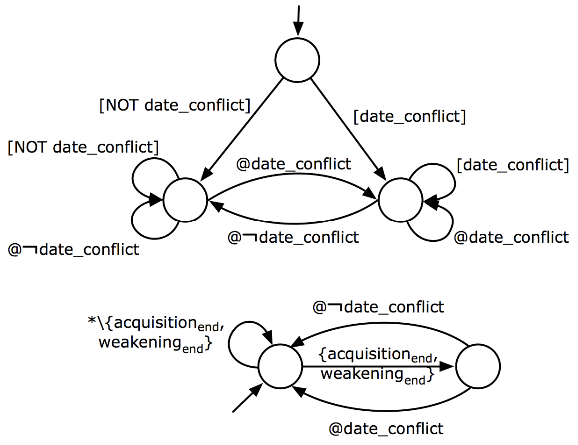

Figure 3. Fluent and updater automata for the tracking variable date_conflict

3.2 Time Constraints

Processes in certain domains can be time-critical. The modeling language should make it possible to annotate tasks with timing information so that we can check whether required time constraints are met by the model. In our modeling framework, a *duration interval* can be associated with any task not refined further in another g-hMSC. It captures the minimum and maximum time taken by a task (in discrete time units). Table 1 illustrates durations of unrefined tasks involved in meeting scheduling.

Table 1. Task durations for meeting scheduling (in days)

Task	Min	Max
Meeting Initiation	1	1
Constraint Acquisition	1	7
Initiator Arbitration	1	2
Plan Meeting	1	1
Constraint Weakening	1	7
Date Range Extension	1	1

The semantics of such temporal annotations is defined in terms of timed automata [Alu94]. The overall process behavior is defined by the parallel composition of:

- the trace-equivalent LTS built from the g-hMSC,
- a timer automaton for each task for which a duration interval is given; the latter enforces the time constraints through a task clock and special events $task_{start}$ and $task_{end}$.

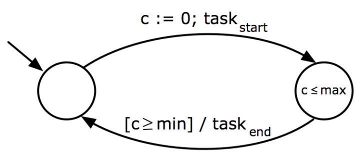

Figure 4. Timer automaton for *Task* with duration [*min*, *max*]

Fig. 4 shows a timer automaton for a task whose duration is [*min*, *max*]. From the initial state, the automaton synchronizes with $task_{start}$ and resets a clock. A clock invariant in the reached state forces leaving it after *max* time. The transition labeled by $task_{end}$ is guarded to prevent ending the task before *min* time has been spent in the state.

3.3 Preconditions on tasks

For some analyses it may be useful to annotate a task with its precondition. The latter must hold in any state where the task starts being performed. In our framework, preconditions are Boolean expressions on fluents and tracking variables. For example, the precondition of task PlanMeeting is ¬ date_conflict.

Postconditions are not introduced as they are already captured implicitly in fluent definitions.

4. Abstract Decorations for Model Analysis

In view of their trace semantics, our process models could be verified using adapted versions of model checkers such as LTSA [Mag06, Dam09] or UPPAAL [Lar97]. This appears computationally expensive as it requires manipulating the trace-equivalent LTS generated from the g-hMSC model. A variety of analyses can fortunately be performed at g-LTS level to avoid such inherent inefficiencies (see Section 9). In particular, [Dam09] showed how guard completeness and disjointness can be checked at decision points thanks to a symbolic execution algorithm that decorates each g-LTS node with a state invariant on fluents.

This section significantly generalizes this decoration algorithm in order to achieve the following objectives:

- support tracking variables, in addition to fluents, in the generation of state invariants;
- generate other types of decorations, in addition to state invariants, to support reasoning about timing, resources, doses, and so forth;
- abstract from the specific kind of transition system to decorate;
- keep the core algorithm template unchanged while supporting additional modeling abstractions.

Section 4.1 presents an abstract decoration algorithm meeting those objectives. Section 4.2 discusses the instantiation process required to generate concrete decorations for use in a variety of analyses. The next sections will detail concrete instantiations to perform dedicated analyses of critical process properties.

4.1 Generating Abstract Decorations

The algorithm in Fig. 5 computes a decoration for each vertex of a graph. When the algorithm terminates, the decoration of a vertex captures the accumulation of the decorations contributed by all paths leading to the vertex. Such accumulation proceeds by propagation of vertex decorations to their successors until a fixpoint is reached.

```
Input: A labeled directed graph G: (V, E)
       A lattice D of decorations: (≤,∧,∨,⊥,T)
       A Propagate function D x E → D
       An initial vertex v0 ∈ V
       The decoration after an empty path d0 ∈ D
Output: deco : V → D

/* initial decorations */
for each v ∈ V do deco(v) ← ⊥
deco(v0) ← d0
/* fixpoint loop, starting with the initial vertex */
toExpl ←{v0}
while toExpl ≠ ∅ do
   source ← getOne(toExpl)
   toExpl ← toExpl\{source}
   for each edge ∈ outgoing_edges(source) do
      /* propagate source decoration */
      pDeco ← Propagate(deco(source), edge)
      /* compute accumulation */
      target ← target(edge)
      deco'(target) ← pDeco ∨ deco(target)
      /* mark as "to explore" if changed */
      if deco'(target) ≠ deco(target) then
         toExpl ← toExpl ∪ {target}
         deco(target) ← deco'(target)
return deco
```

Figure 5. Abstract decoration algorithm

The semantics of the labeled directed graph taken as input is intentionally kept abstract. It is expected to be provided by the labels of vertices and edges. The use of these labels is fully encapsulated in the *Propagate* function given as input. The latter defines how a vertex decoration must be propagated through a single edge, according to the specific graph semantics, to contribute to the decoration of the target vertex.

Decorations are also kept abstract. They must be elements of a bounded lattice. We denote the partial order on this lattice by $\leq$; its supremum and infimum operators by $\vee$ and $\wedge$, respectively; and its bottom and top elements by $\bot$ and $\top$, respectively. Initially, every state is decorated with $\bot$, except a particular vertex v_0 decorated with an initial decoration d_0 (both taken as input). Vertex decorations are propagated along edges using the *Propagate* function until a fixpoint is reached. The accumulation rule is defined through the *supremum* operator. Lattice boundedness is required to ensure termination; the associativity of the *supremum* operator ensures that the order in which the graph is explored does not matter.

4.2 Concrete decoration-based analyses

The abstract decoration algorithm is intended to be applicable for a variety of analyses of the g-LTS generated from a g-hMSC model. For each type of analysis, the following process is applied:

(a) The g-LTS is mapped to a labeled directed graph. In all cases, the graph vertices are given by the g-LTS states and the edges by the transition relation; an edge label is either an event or a guard. The initial state s_0 of the g-LTS is taken as initial graph vertex v_0. Vertex labels are to be carefully chosen for each particular analysis (see Sections 5-8).

(b) A decoration type is defined so as to meet the desired type of analysis.

(c) The decoration lattice structure is defined; an initial decoration d_0 is chosen.

(d) The *Propagate* function is defined, making it precise how edge and vertex labels are used to transform the decoration of a source state through a single edge.

5. Checking Task Preconditions

As tasks in a g-hMSC process model can now be annotated with preconditions, we would like to check that the preconditions when given cannot be violated through paths in the model, or generate them when they are not given.

A precondition checker is obtained in a fairly straightforward manner through the instantiation process in Section 4.2:

(a) No vertex label is specifically required for this analysis.

(b) The decorations to be generated are Boolean expressions on fluents and tracking variables. They are conditions holding every time the corresponding vertex is visited. This instantiation thus generalizes the algorithm in [Dam09] to handle tracking variables in addition to fluents.

(c) The bounded lattice is the set of propositional formulas, with logical implication as partial order; *false* and *true* as bottom and top elements, respectively; and conjunction and disjunction as infimum and supremum operators, respectively. The initial condition C_0 of the g-LTS is taken as initial decoration d_0.

(d) The *Propagate* function is defined as follows:

```
Propagate(deco, edge):
  if label(edge) is a guard
    return deco ∧ label(edge)
  else /* edge label is an event */
    return deco|label(edge)
```

In this function, $deco|_{event}$ denotes the result of applying the corresponding event to the source vertex decoration, according to the definitions of the fluents and tracking variables involved. For a fluent F, if the applied event is among its initiating (resp. terminating) events, the result is obtained by replacing all occurrences of $\neg F$ (resp. F) in the source decoration by F (resp. $\neg F$). For a tracking variable tV, if the applied event is among its update events, the result is obtained by dropping all occurrences of tV from the source decoration (as tV's value might no longer be accurate). If the applied event appears in multiple definitions of fluents and tracking variables, the process is repeated for each of them.

The precondition PRE_T of a task T is never violated if, for every *source* vertex of a T_{start} event, the following formula is verified with a SAT solver:

$$deco\,(source) \models PRE_T$$

Generating preconditions. When task preconditions are not provided by the modeler, our tool uses the same decorations to generate them. For a task T, it retrieves all source vertices of an edge labeled by T_{start}, and takes the disjunction of their decorations. The preconditions inferred this way may need to be further simplified in view of domain properties.

In our running example, the precondition ¬ date_conflict of PlanMeeting (given in Section 3.3) is checked to be correct. As the precondition of DateRangeExtension was not provided, our tool generated the following precondition:

date_conflict ∧ ¬ resolve_by_weakening

6. Checking the Adequacy of Decisions

As tracking variables capture information about quantities in the process environment, we would like to check that every time they are used, they accurately reflect their actual counterpart in the environment. This is particularly important for tracking variables appearing in decision nodes of a g-hMSC model. Inaccurate values for tracking variables at decision points may result in inadequate decisions, where the outgoing branch taken differs from the one that should be taken with accurate values.

We consider two different sources of inaccuracy resulting in potentially inadequate decisions: some tasks affect the environment quantity without updating the tracking variable accordingly (Section 6.1), and the value of a tracking variable becomes outdated after some time (Section 6.2).

6.1 Task-dependent inadequacies

Consider the tracking variable date_conflict in the process model for meeting scheduling in Fig. 1. A value *true* means that every date in the initiator's date range is excluded by at least one participant. This value is expected to be accurate right after all participant constraints have been acquired. It might become inaccurate when the date range is extended; a conflict-free date might now be found in the extended set of dates.

To check that this variable is adequately used, we introduce an *accuracy meta-fluent*. In states where this fluent is true, the value of the associated tracking variable accurately reflects its environment counterpart; in any other state where the fluent is false, the value might be inaccurate. In our example, the meta-fluent definition is thus:

$$\textbf{fluent}\ \text{date\_conflict-}\mathit{Accurate} = <\{\text{acquisition}_{\text{end}}\},\{\text{extension}_{\text{end}}\}>$$

Any tracking variable may have an associated accuracy meta-fluent. The *initiating events* of this fluent are all events that synchronize the tracking variable with its environment counterpart (e.g., constraint $\text{acquisition}_{\text{end}}$ in our example). The terminating events are the ones potentially affecting the environment quantity while not appearing among the update events of the tracking variable (e.g., date range $\text{extension}_{\text{end}}$ in our example). The initial value of accuracy meta-fluents is *false* and therefore omitted in their definition.

Checking a decision node in a g-hMSC model for task-dependent decision inadequacies amounts to checking that the tracking variables appearing in the corresponding guards in the g-LTS model are accurate – that is, their accuracy meta-fluent is *true* in all source states of the guarded transitions.

To do this, our abstract decoration algorithm is instantiated in a way similar to the one described in Section 5. The differences are the following.

- The decorations must now refer to the accuracy meta-fluents in addition to the fluents and tracking variables of the model itself. We simply add the accuracy meta-fluents to the set Φ of fluents to be accumulated by the *Propagate* function.
- As accuracy meta-fluents are initially false, the initial decoration must be $C_0 \wedge \neg\, tV\text{-}Accurate$.

Once such decorations have been computed, we need to verify the following formula for each source state *source* and tracking variable *tV* appearing in a guard on the outgoing transitions:

$$\text{deco}\ (\mathit{source}) \models tV\text{-}\mathit{Accurate}$$

A SAT solver is used for this check. If the formula is not verified, we know that there is a task sequence reaching the decision node in the model such that the value of the tracking variable at this node is inaccurate; the decision on which subsequent path to follow is therefore potentially inadequate.

The decision node date_conflict? in Fig. 1 is checked to be adequate using this technique. The decoration of state 7 in Fig. 2 ensures that date_conflict-*Accurate* is *true* at this point. Indeed, new constraints are immediately re-acquired every time the date range is extended (see Fig. 1). The cancer therapy process fragment modeled in Section 8 will show an example where an inadequate decision is detected by this technique.

An inadequate decision should be resolved in a corrected version of the process model. A simple heuristics consists in adding an initiating event of the corresponding accuracy meta-fluent right before the decision node.

6.2 Time-dependent inadequacies

In the meeting scheduling example so far, there was an implicit assumption that the participant constraints do not change after their acquisition. This is not the case in practice.

It is often the case that tracking variables remain accurate during a certain period of time only. The environment quantities they reflect may change due to events that are unobservable by the process agents. In order to capture such situations, we may specify a duration in the definition of an accuracy meta-fluent:

$$\textbf{fluent}\ tV\text{-}\mathit{Accurate} = <\text{Init}_{\text{Ac}}, \text{Term}_{\text{Ac}}>\ \textbf{duration}\ \mathit{Dur}_{\mathit{Ac}}$$

This fluent holds iff an initiating event has occurred and, since then, no terminating event has occurred and the elapsed time is less than to $\mathit{Dur}_{\mathit{Ac}}$ (in discrete time units).

For checking time-dependent decision inadequacies, each g-LTS state is now decorated with a partial function associated with it:

$$\mathit{countdown}\colon 2^{\Phi\cup\Theta} \rightarrow \mathfrak{R}^{+}.$$

For a specific g-LTS state, the domain of this function is a set of Boolean assignments to fluents and tracking variables that encodes an invariant holding in this state; the disjunction of its elements yields this invariant. For a given assignment in this g-LTS state, the value of *countdown* captures the remaining time to spend in that g-LTS state, coming from the most time-consuming trace leading to it, before this assignment becomes outdated. After that time, the accuracy meta-fluent will necessarily be *false* which means that the associated tracking variable may no longer accurately reflect its environment counterpart. When the accuracy meta-fluent is known to be *false* (e.g., immediately after a terminating event), the value of *countdown* is considered to be *0*. Note that we keep Boolean assignments to fluents and tracking variables in each decoration, in addition to countdown values, in order to rightly propagate countdown values through the guards along model paths.

To check a g-hMSC decision node for time-dependent decision inadequacies, we need to verify the following formula for each source state *source* and tracking variable *tV* appearing in a guard on the outgoing transitions:

$$\forall\ x \in \text{dom}_{\text{countdown}}\ \text{countdown}\ (x) \neq 0,$$

where *countdown* is the decoration of state *source*. If this formula is not verified, we know that there is a task sequence reaching the decision node in the model such that the value of the tracking variable at this node is inaccurate or outdated.

For decoration generation, we follow the process in Section 4.2 to instantiate our abstract decoration algorithm.

Instantiation step (a). Each g-LTS state corresponding to the end of a task (i.e., source state of a task_{end} event) is labeled with the maximal duration of this task. The other states are labeled with "0".

Instantiation steps (b, c). The bounded lattice is defined over the set of *countdown* functions as follows:

```
$f \le g$   if $dom_f \subseteq dom_g$
        and $\forall x \in dom_f,\ g(x) \le f(x)$
$\bot = \varnothing$
$\top = \{(x \mapsto 0) \mid x \in 2^{\Phi\cup\Theta}\}$
$f \curlyvee g =$
      $\{(x \mapsto min(f(x),g(x))) \mid x \in dom_f \cap dom_g\}$
    $\cup \{(x \mapsto f(x)) \mid x \in dom_f \setminus dom_g\}$
    $\cup \{(x \mapsto g(x)) \mid x \in dom_g \setminus dom_f\}$
$f \curlywedge g =$
      $\{(x \mapsto max(f(x),g(x))) \mid x \in dom_f \cap dom_g\}$
```

In this lattice, the partial order is defined by two conditions. The inclusion of function domains corresponds to the logical implication among the encoded invariants; the second condition corresponds to considering countdown values from a worst-case perspective. The bottom element in the lattice is the empty function, corresponding to an unreachable state where $\text{dom}_{\text{countdown}}$ amounts to *false* and no countdown applies. The top element corresponds to a state where all assignments are possible, the disjunction of elements of $\text{dom}_{\text{countdown}}$ being *true*, with the accuracy meta-fluent outdated. The supremum operator yields a *countdown* function whose domain corresponds to a disjunction, with the minimum countdown value on shared assignments being taken according to a worst-case perspective. The infimum operator is its natural counterpart.

The initial decoration is defined as follows:

$$d_0 = \{(x \mapsto 0) \mid x \in 2^{\Phi\cup\Theta} \text{ and } x \models C_0\}$$

The domain of this *countdown* function corresponds to all assignments satisfying the initial condition of the g-LTS. They are all mapped to a countdown value '0' as the accuracy meta-fluent is initially false.

Instantiation step (d). Given the accuracy meta-fluent *tV-Accurate* = <Init_{Ac}, Term_{Ac}> **duration** Dur_{Ac}, the *Propagate* function is defined as follows:

```
**Propagate**(countdown, edge):
 label ← label(edge)
 duration ← label(source(edge))
 **if** label is a guard **return** $\{label\} \lhd countdown$
 **else**
   **if** label $\in Init_{Ac}$
     **return** $\{(x|_{label} \mapsto Dur_{Ac}) \mid x \in dom_{countdown}\}$
   **else if** label $\in Term_{Ac}$
     **return** $\{(x|_{label} \mapsto 0) \mid x \in dom_{countdown}\}$
   **else**
     **return**  $\{(x|_{label} \mapsto$
                 min(0, countdown (x) - duration))
                $\mid x \in dom_{countdown}\}$
```

This propagation rule considers different cases. If the label of an edge is a guard, the domain of the *countdown* function is restricted to the specific Boolean assignment to fluents and tracking variables defined by *label* ("$\lhd$" is the domain restriction operator); countdown values are kept unchanged as guard evaluation does not take time. If the label is an event, the following cases are considered:

- if the event is among those initiating the accuracy meta-fluent, the countdown for each assignment is reset;
- if the event is among those terminating the accuracy meta-fluent, all countdown values are set to zero;
- Otherwise, the countdowns are decremented by the duration of the task given by the label of the source state, without going beyond the lower bound.
- In all cases, the definitions of fluents and tracking variables are used by the xl_{event} operator to update assignments as in Section 5.

Back to our meeting scheduling example, let us consider that the participant constraints are accurate when they are acquired and remain accurate for 15 days unless the date range has been extended in the meantime:

fluent date_conflict-*Accurate* = <{$\text{acquisition}_{\text{end}}$},{$\text{extension}_{\text{end}}$}> **duration** 15

Our tool finds that the decoration of state 7 in Fig. 2 contains the following mapping:

((date_conflict = *false*, resolve_by_weakening = *true*) $\mapsto$ 0)

This means that there exists a process path where a date conflict seems resolved after constraint weakening but with participant constraints possibly outdated. This might happen, for example, if the weakening task is performed twice in a row by the initiator, each time taking the maximum duration of 7 days (see Table 1). Some acquired participant constraints are then inadequately considered accurate for more than 15 days while the meeting is being planned. To resolve this problem, the model might be changed so that the meeting could be cancelled under certain conditions, or date range extension could be the only possibility for the second round, or participants could be asked to confirm and/or update their constraints after 15 days, etc.

7. Analyzing Timing Properties

For time-critical processes such as those found in the medical domain, we would like to infer or verify timing properties. In the former case, we may want to know how long a process can take, in best or worst situations for example. In the latter case, we may want to impose timing requirements on the process (or task), and check that these requirements are always met in the model. Complex time constraints generally require the use of dedicated tools such as temporal model checkers [Lar97]. This section describes a more lightweight, decoration-based technique for inferring or checking simple timing properties.

The decorations to be generated are now partial functions:

$$timeWindow\colon 2^{\Phi\cup\Theta} \to \mathcal{P}\mathfrak{R}^{+}.$$

For a specific g-LTS state, this decoration function associates Boolean assignments of fluents and tracking variables to sets of time points. A functional pair in the decoration specifies at what time points the state can be visited if the associated assignment holds.

For decoration generation, we follow the process in Section 4.2 to instantiate our abstract decoration algorithm. First of all, an upper time bound has to be chosen by the tool user, *MAX* say, in order to bound loops and ensure termination.

Instantiation step (a). Each g-LTS state source of a $task_{end}$ event, corresponding to the end of a task, is labeled with a time interval [*min*, *max*], that is, $\{x \mid x \in \Re^+, \min \leq x \leq \max\}$; this interval captures the minimum and maximum duration of the task. The other states are labeled with [0, 0].

Instantiation steps (b, c). The bounded lattice is defined over the set of *timeWindow* functions as follows:

$$
\begin{aligned}
& f \leq g \quad \text{if } dom_f \subseteq dom_g \\
& \qquad \text{and } \forall x \in dom_f,\ f(x) \subseteq g(x) \\
& \bot = \varnothing \\
& \top = \{(x \mapsto [0, MAX]) \mid x \in 2^{\Phi \cup \Theta}\} \\
& f \curlyvee g = \\
& \qquad \{(x \mapsto f(x) \cup g(x)) \mid x \in dom_f \cap dom_g\} \\
& \quad \cup\ \{(x \mapsto f(x)) \mid x \in dom_f \setminus dom_g\} \\
& \quad \cup\ \{(x \mapsto g(x)) \mid x \in dom_g \setminus dom_f\} \\
& f \curlywedge g = \\
& \qquad \{(x \rightarrow f(x) \cap g(x)) \mid x \in dom_f \cap dom_g\}
\end{aligned}
$$

The partial order is naturally defined in terms of inclusion on sets of time points. The supremum and infimum operators are defined accordingly in terms of set union and intersection, respectively. The top element maps all assignments to the full admissible time interval; the corresponding state can be reached at any time.

The initial decoration captures all assignments compatible with the initial condition of the g-LTS, mapping them to an interval reduced to the singleton "0" (no time has passed so far):

$$d_0 = \{(x \mapsto [0, 0]) \mid x \in 2^{\Phi \cup \Theta} \text{ and } x \models C_0\}$$

Instantiation step (d). The *Propagate* function is defined as follows:

```
Propagate(timeWindow, edge)
 label ← label(edge)
 duration ← label(source(edge))
 if label is a guard return {label}◁ timeWindow
 else
   return {(x|_label ↦ timeWindow(x) ~ duration)
                          | x ∈ dom_timeWindow}
```

Guards are handled by domain restriction as in Section 6.2. The "~" operator "adds" a duration interval to a set of points according to the following definition:

$$\sim:\ P\Re^+ \times [\min, \max] \rightarrow P\Re^+$$

$$T \sim [\min, \max] = \{x \mid x \in \Re^+, x \leq MAX, \exists\, y \in T:\ y + \min \leq x \leq y + \max\}$$

Given a set *T* of time points at which a task may start, in the decoration of a source state, this function yields all possible time points at which the task may end while fitting the upper bound set on the algorithm.

The instantiated decoration algorithm can now be used for checking a variety of time-related properties.

In particular, inferring or verifying the minimum and maximum time taken by a process is achieved by looking at the fixpoint decoration generated at the last process state (e.g., state 17 in Fig. 2), taking the least and greatest time points in the codomain of the *timeWindow* decoration.

In the meeting scheduling example, the minimal process time is 3 days while the maximal one is *MAX* days. Reaching this upper bound means that date conflicts might be left unresolved so that the process needs to continue. We might then increase this upper bound and see what happens.

We may also infer or verify time constraints on tasks. For example, let us assume that the ConstraintsAcquisition task in Fig. 1 is further refined through another g-hMSC. The associated duration interval [1,7] in Table 1 either becomes a requirement we would like to verify or a preliminary estimate to be replaced by a more accurate duration interval inferred from the refinement. In the latter case we locally apply the instantiated decoration generation algorithm on the refining sub-process. This requires knowing the initial condition C_0 of the sub-process itself. This condition is directly obtained by use of the other algorithm instantiation described in Section 5.

Yet another check variant consists in checking process time bounds for specific process paths. To do this, we restrict the initial values of fluents and tracking variables at the initial state by strengthening the initial condition C_0 in d_0.

The instantiation of the abstract decoration algorithm discussed in this section can be extended in a fairly straightforward way for calculating other cumulative properties such as costs or doses in medical processes (drug doses, radiation doses, etc.) The next section will show an example of underdose detection.

8. Application to Cancer Therapy Processes

This section presents a step-by-step construction of a process model fragment for the treatment of rectal cancer. Our input was a documented protocol defining multiple options and decisions to be taken, the corresponding diagnosis and treatment procedures, with strict constraints on therapy timing, medication dosage and accuracy of medical test results. Parts of this protocol were summarized in flowchart form by the process stakeholders. The latter were medical staff and oncologists at the UCL university hospital in Brussels.

The rectal cancer therapy intertwines radiotherapy and chemotherapy to combine the noxious effects of radiations and chemical agents (capecitabine, oxaliplatin). Such combination introduces strong timing constraints between the two modes of treatment. Throughout the overall treatment, multiple patient parameters are monitored. Depending on their evolution, the radiation or drug doses are adapted or the treatment is temporarily or permanently interrupted.

For lack of space, we only consider the presurgical treatment phase, with the level of blood platelets as single patient parameter. This phase consists of five radio-chemotherapy cures. The protocol requires the total radiation dose to be 45 Gy, with 1.8 Gy per day, administered 5 days a week. Capecitabin has to be given every working day; oxaliplatin once a week. Due to surgery constraints, the duration of this pre-treatment phase may not exceed 40 days.

Fig. 6 shows a first modeling shot where protocol excerpts were literally translated, in particular "The decision to treat a patient is related to the platelet level; ....; a blood sample is taken after each cure; ....". In this g-hMSC, [M] is used for [M, M]; a task box with a "*n*X" inside unfolds in a sequence of *n* occurrences of this task.

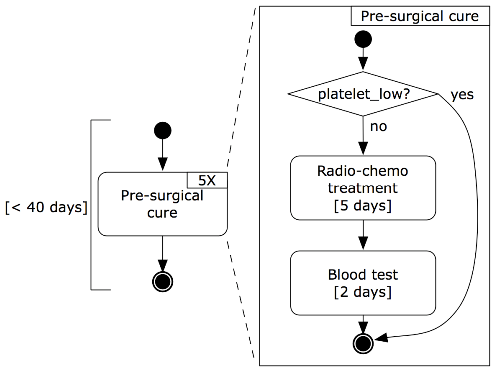

Figure 6. Presurgical treatment phase

Our tool points two problems at this step. First, there is a task-dependent inadequacy; the treatment decision is based on the tracking variable platelet_low whose value may not be accurately available at the decision point (e.g., in case of first cure), see Section 6.1. The problem is resolved by moving the Blood test task so that it appears right before the decision point.

The second, more subtle problem detected by our tool remains after rechecking the fixed model; if the Radio-chemo treatment task is bypassed at least one time, the total dose administered in five cures is less than 45 Gy. In this case, the decoration is the function *dose*: $2^{\Phi \cup \Theta} \rightarrow P\Re^{+}$. The propagation rule captures dose addition along a transition. To resolve the problem, we should replace treatment cancelation by a waiting period of 5 days to allow for normalization of the platelet level. The new model after fix is shown in Fig. 7.

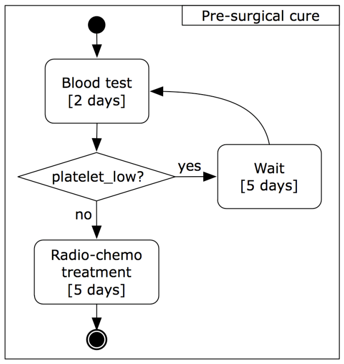

Figure 7. Second modeling iteration

When we recheck the model again, the tool now highlights another type of problem; when the treatment is too often delayed, the 40-day constraint is violated, see Section 7. A model-based discussion with process stakeholders also reveals that a delay in radiotherapy is not recommended. To resolve these two problems, we may uncouple radiotherapy and chemotherapy treatments. Following medical advice, we introduce a possibility of delaying the treatment only on the first occurrence of a platelet fall, see Fig. 8.

When rechecked again, the fixed model turns now to comply with the time, dose and accuracy constraints expressed in the protocol. This modified version raises a new discussion with oncologists on the possibility of dose reduction rather than complete skip of chemotherapy.

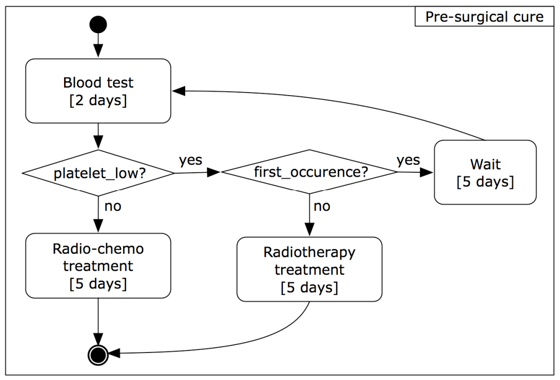

Figure 8. Adequate model

This short excerpt illustrates the stepwise model construction process intertwined with a variety of tool-supported checks. Note that the checks illustrated here take place in the *early* modeling stages. Throughout the model elaboration process, the properties holding on coarser-grained versions of the g-hMSC model are preserved on the finer-grained ones.

9. Discussion

The abstract decoration algorithm symbolically explores the entire process state space, captured in a structured and compact way through a g-LTS. The potential state explosion problem is tackled in different ways:

- The decoration generation cost may be amortized among different checks. For example, checking all task preconditions requires executing the algorithm only once. A straightforward optimization for decision adequacy checking might check the accuracy of multiple tracking variables within a single execution as well.
- The g-hMSC refinement mechanism naturally supports local and incremental checks. We may thereby perform checks on small g-LTS to reduce the state space size.
- The generated decorations are subject to exponential blow-up in worst-cases. Our tool uses binary decision diagrams (BDDs) to keep them compact as much as possible – in particular, for domains of partial functions. Sets of time points are represented in compact form through sets of intervals. For further optimization, dedicated data structures ensuring canonical forms for decorations need to be implemented.

Our approach builds on good-old-times principles of abstract interpretation [Cou77]. It differs from model checking in two significant ways.

- Properties are checked locally in specific states; they do not necessarily involve temporal histories. Our experience suggests that quite a few "interesting" properties fit this class, as the different examples in the paper aimed at showing. For temporal properties on histories, model checkers can still be used thanks to the trace semantics of our g-hMSC models.
- The way our checks are currently performed does not naturally return violation traces. The tool should be enhanced to support this.

Earlier efforts were made to analyze temporal aspects of a process model. In [Dem06], workflow models are encoded in timed automata [Alu94] and then model-checked. [Ede00] proposes a technique for checking the satisfiability of time constraints on directed graphs with cycles. In both cases, no high-level process language is available to modelers; decisions are not supported. In [Bau06], medical guidelines are written in ASBRU and then rewritten in SMV for model checking. False negatives may however appear due too coarse time abstractions.

Our detection of inadequate decisions somewhat corresponds to the identification of obstacles to accuracy goals [Lam00]; the (implicit) accuracy goal here is that tracking variables should always be accurate at some decision point. In the same vein, [Che06] describes a heuristic technique for deriving fault trees from processes specified in the Little-JIL language, suggesting that tasks may be wrongly executed, communications may fail, etc. In this paper, potentially inadequate decisions are detected even if the process performs correctly along some paths. In [Whi00], conflicts are detected between pre/post-conditions and scenarios prior to statechart generation. This is somewhat similar to our precondition violation detection, but applicable to finite event sequences only.

10. Conclusion

As models for critical decision-based processes may be used for a variety of purposes, their construction and analysis should be supported by systematic methods and tools. The abstract decoration algorithm in the paper can be instantiated to any decoration type having a bounded lattice structure, including assertions on fluents, sets of functions, of time points, etc. Such instantiations enable local, incremental checks of task preconditions, accuracy and freshness of environment tracking variables, adequacy of decisions, and property satisfaction based on cumulative quantities referring to timing, doses, resource consumption or costs. Our approach applies to models close to the informal sketches provided by process stakeholders; our tool-supported techniques work on a structured form of LTS, generated from such models, that keeps state enumeration implicit.

Our techniques are currently used for building error-free models of safety-critical healthcare processes. A variety of errors in the model sketches provided to us were detected and fixed. Although preliminary, this experience appears promising as we apply the techniques to clinical pathway models for other cancer types.

Our approach raises a number of issues to be worked out. Showing counter-example traces is sometimes important to understand causes of the problem found in the process model. Domain properties should be introduced in the model for simplifying derived information such as preconditions, in order to make them more compact and more understandable. The operational process model should be enriched with other declarative information, such as the goals underlying tasks, for complementary analyses [Lam09]. The generation of visual abstractions for process stakeholders, such as patient timelines, appears important too.

Acknowledgments. This work was partially supported by the Regional Government of Wallonia (GISELE project, RW Conv. 616425) and the MoVES project (PAI program of the Belgian government). Warmest thanks are due to Drs. Yves Humblet and Pierre Scaillet, and to the medical staff at the Chemotherapy and Radiotherapy units of Saint-Luc hospital for their time and experience sharing.

11. References

[Alu94] R. Alur and D.L. Dill, “A theory of timed automata”, *Theoretical Computer Science* 126:183-235, 1994

[Bau06] S. Bäumler, M. Balser, A. Dunets, W. Reif, J.Schmitt:, “Verification of Medical Guidelines by Model Checking - A Case Study”, *SPIN 2006*: 219-233

[Che06] B. Chen, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, “Automatic Fault Tree Derivation from Little-JIL Process Definitions”, *Proc. SPW 2006: Software Process Workshop*, Shanghai, LNCS 3966, pp. 150-158, May 2006.

[Cla08] L. A. Clarke, G. A. Avrunin, and L.J. Osterweil, “Using software engineering technology to improve the quality of medical processes”, *Companion Proc. 30th Intl. Conf. Software Engineering*, Leipzig, May 2008, 889-898.

[Cou77] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”, *Proc. POPL’1977: 4th ACM Symp. on Principles of Programming Languages*, 1977, 238-252.

[Dam05] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde, “Generating annotated behavior models from end-user scenarios”, *IEEE Trans. on Software Engineering*, Vol. 31 No.12, Dec. 2005, 1056-1073.

[Dam09] C. Damas, B. Lambeau, F. Roucoux, A. van Lamsweerde, "Analyzing critical process models through behavior model synthesis," *Proc. ICSE’09: 31st Int. Conference on Software Engineering*, Vancouver, 2009

[Dem06] E. De Maria, A. Montanari, M. Zantoni, “An automaton-based approach to the verification of timed workflow schemas”, *TIME 2006*: 87-94

[Ede00] J. Eder, E. Panagos, M. Rabinovich, “Time Constraints in Workflow Systems”, CAiSE 1999: 286-300.

[Fic97] S. Fickas, A. Finkelstein, M. Feather, A. van Lamsweerde, “Requirements & Specification Exemplars”, *Automated Software Engineering*, Vol. 4 No. 4, 1997

[Gia03] D. Giannakopoulou and J. Magee, “Fluent Model Checking for Event-Based Systems”, *Proc. ESEC/FSE 2003*, Helsinki, 2003.

[Han06] M. Han, T. Thiery, X. Song, “Managing Exceptions in Medical Workflow Systems”, *Proc. ICSE’06: 28th Intl. Conf. on Software Engineering*, Shanghai, May. 2006.

[Lam00] A. van Lamsweerde and E. Letier, “Handling Obstacles in Goal-Oriented Requirements Engineering”, *IEEE Trans. Softw. Eng.*, Vol. 26 No. 10, October 2000, 978-1005.

[Lam09] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[Lar97] K. G. Larsen, P. Pettersson and W. Yi, “Uppaal in a nutshell”, *Int. Journal on Software Tools for Technology Transfer*, Vol. 1 No. 1-2, Oct. 1997, 134-152.

[Mag06] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*. Second Edition, Wiley, 2006.

[Whi00] J. Whittle and J. Schumann, “Generating Statechart Designs from Scenarios”, *Proc. ICSE’2000: 22nd Intl. Conf. on Software Engineering*, Limerick, 2000, 314-323