IEEE Transactions on Software Engineering Vol. 40 No. 4, April 2014, 338-365.

Analyzing Critical Decision-Based Processes

Christophe Damas, Bernard Lambeau and Axel van Lamsweerde, *Member, IEEE*

***Abstract*— Decision-based processes are composed of tasks whose application may depend on explicit decisions relying on the state of the process environment. In specific domains such as healthcare, decision-based processes are often complex and 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 a variety of errors to be detected early and incrementally on partial models, notably: inadequate decisions resulting from inaccurate or outdated information about the environment state; incomplete decisions; non-deterministic task selections; unreachable tasks along process paths; and violations of non-functional process requirements involving time, resources or costs. The proposed techniques are based on different instantiations of the same generic algorithm that propagates decorations iteratively through the process model. This algorithm in particular allows event-based models to be automatically decorated with state-based invariants.

A formal language supporting both event-based and state-based specifications is introduced as a process modeling language to enable such analyses. This language mimics the informal flowcharts commonly used by process stakeholders. It extends High-Level Message Sequence Charts with guards on task-related and environment-related variables. The language provides constructs for specifying task compositions, task refinements, decision trees, multi-agent communication scenarios, and time and resource constraints.

The proposed techniques are demonstrated on the incremental building and analysis of a complex model of a real protocol for cancer therapy.

***Index Terms*—Process modeling, process analysis, model verification, decision errors, safety-critical workflows, non-functional requirements, domain-specific languages, formal specification.**

I. 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 domain-specific processes such as medical processes [13, 25, 37, 68, 69] where process safety is a key concern [16, 39, 44]. Conversely, such domains raise new challenges on modeling and analysis techniques. For example, cancer therapy processes are composed of safety-critical subprocesses, such as radiotherapy, surgery and chemotherapy processes, to be coordinated over long periods of time, at multiple sites, according to critical decisions often made under incomplete information, and subject to a variety of non-functional requirements. The latter refer to strict timing and dose constraints, resource limitations, cost restrictions, and so forth. Such processes are continuously evolving from progress in research and practice.

Models in this context may be used for a variety of purposes, e.g., for process orchestration, conformance checking, process documentation, or the generation of directives, explanations or other operational information targeted at specific parties [13, 25].

Process models should therefore be as error-free as possible. Building an adequate, complete, and consistent model may be far from easy in such domains. Techniques should therefore help detect and fix severe flaws –in the model being built or in the actual process itself [12, 25].

To enable tool-supported analysis, the target processes should be captured through some adequate formal model. Many languages are available for modeling processes and workflows, e.g., UML Activity Diagrams [62], BPMN [63], Yawl [27, 75] and Little-Jil [78] to cite just a few. When a formal semantics is available, such languages support various analyses such as model checking against event-based properties [28, 54, 79], verification of process termination [73] or of absence of deadlocks [54, 80], or conformance checking between the process model and its execution [73]. Model enactment can also be used for runtime support [78].

The modeling techniques available to date do not allow process decisions to be formally captured in terms of state variables characterizing the process environment (e.g., state variables about the patient under treatment). As a consequence, the properties that can be model-checked are purely event-based; they refer to task applications only. When alternative branches in a task flow are supported, the choice among them is non-deterministic.

The paper focusses on decision-based processes to address this current limitation. In a *decision-based process,* decisions relying on the state of the process environment regulate the subsequent tasks to be specifically performed. For example, a specific sequencing of weekly chemotherapy sessions is the outcome of a medical decision relying on environment state variables such as the patient's blood platelet level.

The possible unobservability of the environment state at which a decision must be made is a challenging issue raised by such processes. State variables approximating the environment state are needed; such variables do not necessarily reflect the exact state of the process environment at the corresponding decision

Manuscript received November 12, 2012. This work was partially supported by the Regional Government of Wallonia (GISELE and PIPAS projects, RW Conv. n° 616425 and 1017087) and the MoVES project (PAI program of the Belgian government).

The authors are with the Department of Computing, ICTEAM Institute, Université catholique de Louvain, e-mail: {christophe.damas, bernard.lambeau, axel.vanlamsweerde}@uclouvain.be.

point. In our example, the exact value of the patient's platelet level might not be the one used in the state where the specific therapy decision is taken. A modeling/reasoning framework must be able to cope with this.

The main contribution of the paper is a coherent set of complementary techniques for early and incremental analysis of critical decision-based process models.

As a prerequisite, we need a formal modeling language to enable those analyses. For ease of communication and validation, the models should be as close as possible to the material commonly used by process stakeholders.

The available documentation of medical guidelines, pathways and protocols [30, 43, 57, 65] together with comparative surveys involving process stakeholders [42] provide evidence that stakeholders in this domain use *informal flowcharts* to document their procedures. Such documentation typically shows:

- sequences of tasks, the latter being sometimes refined into subtasks;
- conditions guarding alternative task flows, possibly nested to form decision trees;
- occasionally, fine-grained interaction sequences among process agents [66];
- flowchart annotations with time constraints, resource restrictions, or underlying goals on state variables about the process environment.

This observation was confirmed by our extensive experience in assembling clinical process fragments supplied by medical staff in multiple hospitals. The unconvinced reader may check *Google Images* with the keywords "care pathway flowchart" for a wide variety of examples of use of flowcharts by process stakeholders in the medical domain.

To support both analyzability and communicability, the language of High-Level Message Sequence Charts [40] is extended with guard constructs and non-functional annotations. The extended language, called Guarded High-Level Message Sequence Charts (g-HMSCs), mimics informal stakeholder flowcharts while having a formal trace semantics in terms of labelled transition systems (LTS) [25].

To enable process decisions based on both task applications and environment states, the language should integrate the event-based and state-based specification paradigms [23, 47, 56]. Specific process paths can then be governed by the truth value of hybrid conditions on states and events. Such integration is achieved in a g-HMSC by letting guards refer to task-related fluents [31, 59] and environment tracking variables. The latter, more precisely defined in Section IV, are intended to track environment quantities that are not continuously observable by the agents involved in the process.

The analysis techniques described in this paper support the following types of checks on g-HMSCs.

- *Guard analysis*. The guards on the various alternatives at any decision point in a task flow must be satisfiable in the state reached at that point (for subsequent tasks to be applicable). They may not overlap in that state (for decisions to be deterministic). They must cover all possible cases in that state (no alternative branch is missing).
- *Detection of inadequate decisions*. A decision can be *inadequate* if it relies on incorrect information about the environment state. This arises from state variables being outdated or inaccurate due to missing tasks or unexpected events. Every decision in a task flow should be adequate.
- *Verification of non-functional requirements on the process*. All timing, resource and cost constraints should be met along all possible paths of the process model.
- *Verification of task preconditions*. All task preconditions should be satisfied along all possible paths of the model.

These various types of checks are performed through different instantiations of the same generic algorithm for propagating decorations through the process model. This algorithm is designed to require as little instantiation effort as possible; for each type of check, the user just needs to instantiate the generic decoration together with the rule for propagating decorations through a single state transition. No correctness proofs of the instantiated algorithms are required; the proof of the generic algorithm is similarly instantiated.

The paper expands on preliminary results described in [25]. The main improvements and extensions include: (a) the handling of environment state variables, (b) new types of checks, namely, the adequacy of decisions and the satisfaction of non-functional requirements, and (c) a uniform treatment of quite different types of check through a single generic model decoration algorithm with fairly simple instantiations. Unlike our model checker described in [25], all checks are performed here at the intermediate level of a guarded LTS, generated from the g-HMSC model, which avoids enumerating all LTS traces covered by the g-HMSC model.

The paper is organized as follows. Section II introduces the running example used for explanation throughout the paper. Section III provides some minimal background on LTS, HMSCs and fluents. Section IV introduces g-HMSCs for process modeling. Section V defines the formal trace semantics of this language. Section VI describes our generic decoration algorithm. Section VII presents various instantiations for different types of check on a decision-based process model. Section VIII discusses tool support for those different analyses. Section IX evaluates the approach with respect to correctness; performance and scalability; applicability; utility; and usability. Section X discusses related work.

II. Motivating Example

The following process for treating acute manic-depressive troubles is used throughout the paper as a simple running example for explanatory purpose.

A patient with symptoms compatible with manic-depressive disorder enters the workflow through an admission consultation. During this consultation, a psychiatrist determines whether the patient is a danger for himself. If so, an acute drug therapy is started. Otherwise, an evaluation consisting of a psychological test and a blood test is performed; the blood test is necessary for determining whether the patient is under influence of drugs. Based on test results, the psychiatrist provides a diagnosis and a treatment recommendation. If necessary, the patient may be put in observation. A long-term medication may also be prescribed. In case of medication, a

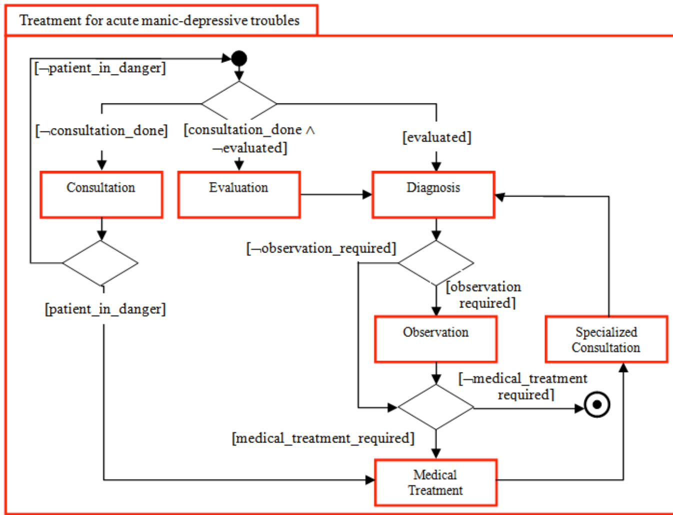

Fig. 1. Process model for treating acute manic-depressive troubles

specialized consultation is planned to assess the treatment effect and the patient's evolution.

Fig. 1 shows a model sketch for the process of treating manic-depressive troubles. Boxes there denote tasks whereas diamonds capture decision points with associated guards on alternative subsequent task flows. The precise semantics of such model is detailed in Section IV.

Interesting questions may already arise from this model sketch, for example,

- What is the minimal and maximal time for treating acute manic-depressive troubles?
- Is it possible for the psychiatrist to make a medical treatment decision that would be based on outdated information about the patient?
- At any process step where a decision node is reached, do the associated guards cover all possible cases? (If not, a process run might be blocked forever with no task prescribed.) Do they overlap? (If so, two patients having right the same symptoms could undergo different treatments.)

The analysis techniques presented hereafter enable precise answers to these questions, among others.

The flow graph in Fig.1 is a refinement of the coarser-grained task "Treatment for acute manic-depressive troubles" mentioned on the top left. Tasks that are not further refined are specified by scenarios showing the sequence of interaction events among agent instances involved in the task, see Fig. 2.

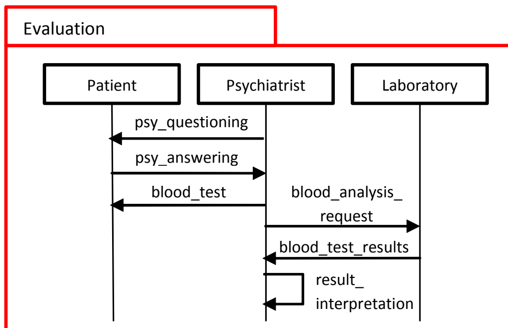

Fig. 2. Scenario for the *Evaluation* task

III. Background

To make the paper self-contained, some basics on message sequence charts, high-level message sequence charts, labelled transition systems, and fluents are first recalled.

A. High-level message sequence charts

Message sequence charts (MSCs) are commonly used for capturing multi-agent scenarios [40]. *Agents* are active process components; they define the process scope. Their instances perform tasks, thereby monitoring some events or state variables and controlling others [48].

An *MSC* is composed of vertical timelines associated with agent instances and horizontal arrows representing interaction events. A timeline label declares a class of corresponding agent instances. An arrow label declares an interaction event among the source and target agent instances; the event is synchronously controlled by the source and monitored by the target. Fig. 2 shows an MSC refining the *Evaluation* task for manic-depressive troubles. The scenario involves three agents, namely, a patient, a psychiatrist, and a laboratory.

A *High-level MSC* (HMSC) is a directed graph where each node is an MSC or a finer-grained HMSC [40]. Edges indicate the acceptable orderings among scenarios. They allow for scenario sequencing and repetition. A complex scenario may thereby be broken into manageable parts that are ordered according to the HMSC specification.

B. Labeled transition systems

A *labeled transition system* (LTS) is an automaton defined by a structure (S, E, δ, s_0), where S is a finite set of states, E is a set of event labels, δ is a labeled transition relation ($\delta \subseteq S \times E \times S$), and s_0 is an initial state [56, 60].

An LTS *trace* is a sequence of events $<e_0,\dots,e_n>$ accepted by the LTS from its initial state ($e_i \in E$).

The semantics of MSCs and HMSCs is defined in terms of LTS and parallel composition [72]. An MSC timeline defines a finite LTS trace capturing the behavior of the corresponding agent instance. The semantics of an entire MSC is similarly defined as a trace of the LTS of the entire system being modelled.

C. Fluents

A fluent FL is an atomic proposition defined by a set $Init_{FL}$ of initiating events, a set $Term_{FL}$ of terminating events, and an initial value $Initially_{FL}$ that can be *true* or *false* [31, 56, 59]. The sets of initiating and terminating events must be disjoint. A fluent definition takes the form:

$$\textbf{fluent}\ FL = < Init_{FL}, Term_{FL} > \textbf{initially}\ Initially_{FL}.$$

For example, a fluent *evaluated* can be introduced to capture whether a patient is evaluated (see Fig. 1):

fluent *evaluated* = *<result_ interpretation, medical_treatment_ok>*
initially *false.*

This fluent definition specifies that a patient is evaluated after a *result_interpretation* event (see Fig. 2) and stops being evaluated after a *medical_treatment_ok* event. The patient is initially not evaluated. The fluent thereby specifies a postcondition of the *Evaluation* task; its negation yields a postcondition of the *Medical Treatment* task.

For any LTS trace and set of fluents, a *state* can be defined after every event in the trace. This state is characterized by the value of every fluent at this point in the trace. In such a *fluent value assignment,* a fluent gets *true* (resp. *false*) if either of the following conditions holds:

- the fluent is initially *true* (resp. *false*) and no terminating (resp. initiating) event has occurred;
- some initiating (resp. terminating) event has occurred with no terminating (resp. initiating) event occurring since then.

Fluents are commonly used for integrating event-based and state-based specifications [56], in particular to enable state-based model-checking of event-based models [31] or to synthesize event-based models annotated with state-based information [22, 23]. In an *event-based specification,* system behaviors are captured in terms of event sequences; in a *state-based specification*, system behaviors are captured in terms of state sequences [47].

IV. Modeling Decision-Based Process Models

This section introduces Guarded High-Level Message Sequence Charts (*g-HMSC)* as a process modeling language. It first overviews the main language constructs (Section IV.A) before discussing the various types of events involved in a g-HMSC (Section IV.B), the various types of variables being manipulated (Section IV.C), the mechanism for specifying initial conditions on a process (Section IV.D), and the various types of optional annotations for specifying task preconditions and non-functional features refering to time and resource usage (Section IV.E).

A. Process models as g-HMSCs

A *g-HMSC* is a directed graph with three types of node.

- A *task node* captures a process task, that is, a work unit performed by collaboration of agent instances involved in the process.
- A *decision node* captures a process decision. It is characterized by a set of guards. Each *guard* is associated with a specific outgoing branch; it specifies the condition for the tasks along this branch to be performed. Guards are Boolean expressions on process variables (see Section IV.C hereafter).
- *Initial* and *terminal* nodes represent the start and end of the process, respectively.

Nodes in a g-HMSC are connected by two types of arcs.

- An outgoing arc from a task or initial node is called *continuation*. It prescribes how the connected nodes must be sequentially composed.
- An outgoing arc from a decision node is called *guarded transition*. The corresponding guard must be evaluated to true for the arc to be followed.

Tasks may be refined. A non-terminal task is refined into subtasks and decisions forming a finer-grained g-HMSC. A terminal task is an MSC scenario showing sequences of interaction events among agent instances.

A g-HMSC can be represented graphically or textually. The graphical syntax is used here; see [46] for the guarded command language used by the tool. A task is represented by a box, expanded into a finer-grained g-HMSC (non-terminal task) or an MSC (terminal task) –see Fig. 1 and Fig. 2, respectively. A decision is represented by a diamond with a guard labeling each outgoing branch of the decision. In simple cases with two branches only, the guard expression may be moved up inside the decision node with 'yes' or 'no' label being attached to the corresponding branch.

B. Task-related events

As detailed in Section V, the formal semantics of the g-HMSC language is defined in terms of event traces. Two kinds of events are involved.

- *Interaction events* are the MSC events capturing synchronous interactions among agent instances in terminal tasks.
- *Action events* correspond to task applications. Every task *T* has two built-in action events associated with its start and end; they are denoted by T_{start} and T_{end}, respectively.

As seen below in the paper, the built-in *start* and *end* action events serve multiple purposes.

- They specify the task boundaries in the event traces produced by the process. This provides a useful traceability mechanism between refinement trees in the g-HMSC model and event traces in the corresponding lower-level, LTS-based model manipulated by analysis tools. This mechanism is required for roundtrip feedback by such tools.
- The *start* and *end* action events allow process variables and time distances to be more accurately defined through them.
- They yield a default refinement for tasks that are not yet refined, thereby enabling early analyses on partial models.
- They can be used for synchronizing agent instances so as to prevent a task from starting before its predecessor in the graph is not fully completed.

C. Process variables

The variables appearing in guards at decision nodes of a g-HMSC model dictate which paths are to be followed in specific process instances. Such variables get their values from the occurrence of specific events; no explicit assignment is needed which makes g-HMSC models simpler. Process variables may be task-related fluents or environment tracking variables.

1) Task-related fluents

The atomic conditions found in g-HMSC guards may be fluents whose initiating and terminating events are interaction events or action events. For example, the fluent *consultation_done* in Fig.1 is defined as follows:

fluent *consultation_done* =
<{$Consultation_{end}$},
{$Medical\ Treatment_{end}$}> **initially** *false.*

2) Environment tracking variables

A decision may also depend on environment quantities that are not necessarily observable at the corresponding g-HMSC node by the agent instances involved in the process. For example, a medical decision might depend on the patient's blood rate of

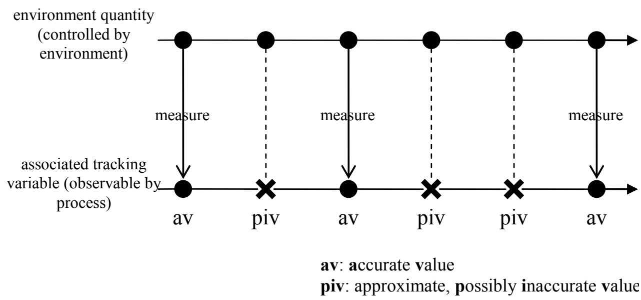

Fig. 3. Behavior of an unobservable environment quantity and its tracking variable

anxiolytic drug; the actual value for this quantity might not be observable at the corresponding decision point.

Tracking variables are intended to approximate environment quantities as accurately as possible through some observable counterpart. When a tracking variable appears in a guard, the decision is based on its current value rather than the actual, unobservable counterpart. In our example, a tracking variable might capture the anxiolytic level found in the patient's record. The decision might then be based on this quantity instead of the actual, unobservable anxiolytic level.

A tracking variable has to be updated periodically through dedicated tasks whose applications yield *measure events* (see Fig. 3). These are events whose effect is to reset the value of the tracking variable to its environment counterpart; both values are then "re-synchronized". For example, *blood_test_results* in Fig. 2 is a measure event for the tracking variable *anxiolytic_tooHigh* as the patient record accurately represents the anxiolytic level right after a blood test.

As suggested in Fig. 3, the value of a tracking variable does not change between two occurrences of a measure event. Between such occurrences, however, some tasks might affect the environment quantity; the corresponding change is then not reflected by the tracking variable. The application of such tasks yields so-called *outdate events*. For example, the anxiolytic level is known to be affected by the administration of an anxiolytic treatment; the latter invalidates former blood test results.

The value of a tracking variable resulting from a measure event might also be known to become outdated after a certain time. For example, the anxiolytic level obtained through a blood test might be considered outdated after 2 days.

A tracking variable is therefore defined by a set of measure events, an initial value and, optionally, a set of outdate events and a duration during which its value remains accurate. The definition takes the following form:

trackVar $tV = \{MeasureEvents\}$ **initially** $Initially_{tV}$
$\{OutdateEvents\}$ **duration** Dur_{tV}.

In our example, we might have:

trackVar *anxiolytic_tooHigh* =
{*blood_test_results*} **initially** *false*
$\{AnxiolyticTreatment_{start}\}$ **duration** *2 days*.

Tracking variables are restricted in this paper to Boolean variables as they are only used for formalizing decisions. They can be seen as predicate abstractions –e.g., the variable *anxiolytic_tooHigh* captures whether "the patient anxiolytic level is above 100 µg/L".

Tracking variables should not be confused with fluents. In contrast with initiating and terminating events, *measure events* do not define which value the tracking variable gets; they only capture that this value is accurately updated to the current value of the corresponding environment quantity. On the other hand, *outdate events* update the environment quantity but not the corresponding tracking variable; they capture that the tracking variable might no longer be accurate.

D. Initial context conditions

A flexible process model should make it possible to capture a process whose instances follow different paths according to different initial conditions. In our running example, different paths should be followed in Fig. 1 dependent on whether or not the patient had a consultation before.

In the definition of the fluents and tracking variables used in a g-HMSC, we may omit the initial value for some variables to express that these may initially be *true* for some process instances and *false* for others. Initial values are thus defined at instance level, not at class level. For example, the definition of the fluent *consultation_done* hereafter specifies that some patients might initially have a consultation already done whereas others might not:

fluent *consultation_done* =
$<\{Consultation_{end}\}, \{Medical\ Treatment_{end}\}>$.

In case of variable definitions with no initial value, we may want to specify at class level that certain combinations of initial values are ruled out in view of domain properties known from a companion goal model [48]. For example, assuming that the initial values of the fluents *consultation_done* and *evaluated* were not specified, we know that the patient may have an evaluation only if a consultation has already been performed; any initial state should meet this domain property.

In such cases, an *initial context condition* may be specified on the g-HMSC in order to constrain the acceptable initial values of process variables. In our example, the initial context condition is:

$$C_0: \quad evaluated \Rightarrow consultation_done.$$

E. Task annotations

A task in a g-HMSC may be annotated with features such as its precondition, its duration, its cost, the resources needed to perform it, and so forth. Such annotations are required only for specific types of analysis; they are thus optional in the process definition.

1) Task preconditions

A task *precondition* is a necessary condition on input variables for the task to be applied. It must hold in any state where the task starts being performed.

Preconditions are Boolean expressions on fluents and tracking variables. For example, the precondition of the task *Medical Treatment* in Fig. 1 is the fluent *evaluated*.

Note that task *postconditions* are indirectly captured by fluent definitions. For example, the postcondition of the task *Consultation* in Fig. 1 is the fluent *consultation_done* since the end of this task is specified as an initiating event for this fluent, see the fluent definition in Section IV.C.

Task	Min	Max
Consultation	1	1
Diagnosis	1	1
Evaluation	2	3
Observation	7	15
Medical Treatment	21	21
Specialized Consultation	1	1

Table 1. Task durations for treating acute manic-depressive troubles (in days)

2) *Task durations*

A *duration interval* may also be specified for any task. It captures the *minimum* and *maximum* time taken by the task, expressed in number of time units. Table 1 illustrates durations of unrefined tasks from Fig. 1.

Such information enables us to check whether time-critical processes meet their timing requirements (see Section VII.D).

More precisely, the duration of an action event T_{start} or T_{end} is characterized by the following function:

$$Duration: E \rightarrow \mathcal{P}(N^{+})$$
$$Duration\ (T_{start}) = \{0\},$$
$$Duration\ (T_{end}) = [min, max],$$

that is, the set of natural numbers between *min* and *max* (included) where *min* and *max* denote the minimum and maximum time taken by the corresponding task, respectively. ($\mathcal{P}$ is the standard powerset notation and N^{+} denotes the set of natural numbers).

The duration of an interaction event is the singleton {0}.

3) *Other non-functional features*

A task may also be annotated with other information relevant to non-functional process requirements such as resources needed to perform it, dose to be delivered, cost, location, and the like. For example, a single radiotherapy task might be annotated with a delivered dose of 1.8 Gray.

Such information enables us to check process models for requirements violations (e.g., radiation overdose or underdose). Counterparts to the preceding *Duration* function are used for this (see Section IX.D).

V. Operational semantics: from g-HMSC models to g-LTS

This section introduces an intermediate formalism between the g-HMSC and LTS formalisms. Roughly, a guarded LTS (g-LTS) is a transition system whose transitions are labeled by events or guards.

The g-LTS formalism provides a convenient milestone on the way from a g-HMSC process model to the corresponding LTS. In particular, it allows the set of traces accepted by the g-HMSC to be precisely defined. This set of traces may in turn be converted into a set of LTS traces. A formal LTS trace semantics allows us to reuse existing frameworks and tools, in particular, model-checking techniques implemented in the LTSA toolset [25, 56].

A g-LTS is a structured form of LTS that reduces state explosion through guard abstractions. It is easier to understand and facilitates code generation. The different types of analyses in this paper are performed on g-LTS representations –one abstraction level above the LTS counterpart analyzed by the model checker discussed in [25].

A. *Guarded LTS*

A *guarded LTS* (g-LTS) is defined by a structure (S, E, VAR, δ, s_0, C_0) where

- S is a finite set of states,
- E is a set of event labels,
- VAR is a set of fluents and tracking variables defined over E,
- δ is a guarded transition relation:
  $\delta \subseteq S \times (E \cup GUARD) \times S$
  where *GUARD* is the set of Boolean formulae over *VAR,*
- s_0 is the initial state,
- C_0 is a Boolean formula over VAR capturing an initial context condition.

Fig. 4 shows a g-LTS derived from the g-HMSC shown in Fig. 1 and Fig. 2. The guards there appear between brackets.

Every transition in a g-LTS is labeled by a guard or by an event.

- A *guard* is a Boolean formula over fluents and tracking variables. It must be evaluated to *true* for the associated transition to be activated.
- A g-LTS *event* is an interaction event from a terminal g-HMSC task (e.g., *result_interpretation* in Fig. 4) or a g-HMSC action event (e.g., $Consultation_{start}$).

The initial context condition C_0 plays the same role as in g-HMSCs.

Any g-HMSC process model can be rewritten as a g-LTS

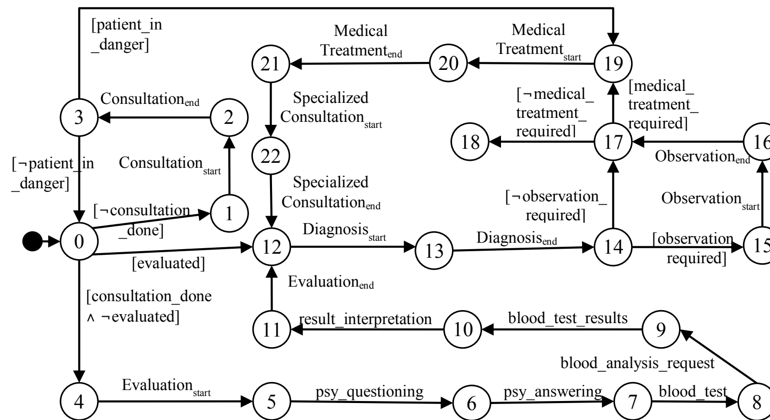

Fig. 4. g-LTS corresponding to the g-hMSC in Fig. 1 and MSC in Fig. 2

having the same set of traces. The rewriting algorithm is detailed in [25, 46]. It extends the algorithm in [72] so as to take a g-HMSC as input and a g-LTS as output. The latter abstracts from agents and captures the set of global behaviors covered by the g-HMSC.

The optional task annotations discussed in Section IV.E are not involved in the resulting operational semantics of a g-HMSC; they are used for dedicated checks (see Section VII).

B. Trace semantics of g-LTS

The semantics of a g-LTS is in turn defined in terms of guard-free event traces. Let G denote the g-LTS (S, E, VAR, δ, s_0, C_0).

A *g-LTS execution* from s_0 is a pair (*Init*, σ) where:

- *Init* is an initial variable assignment mapping every variable to *true* or *false*,
- σ is a sequence of labels l_i, some being events and others being guards ($l_i \in E \cup GUARD$).

A g-LTS execution from s_0 is accepted by g-LTS G iff the following *acceptance conditions* are met for every i:

- $\exists\, s_{i+1} \in S$ such that $(s_i, l_i, s_{i+1}) \in \delta$ *(inclusion)*,
- $Init \models C_0$ *(admissible start)*,
- $VA_i \models l_i$ if $l_i \in GUARD$ *(guard satisfaction)*,

where VA_i is the variable assignment after the i-th event in the g-LTS execution (with $VA_0 = Init$).

The *inclusion* condition states that the sequence of labels is accepted by the automaton. The *admissible start* condition states that the initial variable assignment must meet the initial context condition C_0. The *guard satisfaction* condition ensures that all guards are met along the sequence.

An *event trace* of g-LTS G from initial state s_0 with respect to an initial variable assignment *Init* is a g-LTS execution accepted by G where all labels corresponding to guards have been removed. The set of event traces accepted by G is the union of all such traces, for all initial states *Init* meeting the *admissible start* condition.

An algorithm for LTS generation from a g-LTS can be found in [25, 46]. Unlike the model checker discussed in [25], the analysis techniques in this paper do not require LTS traces to be explicitly produced.

VI. Computing Generic Decorations on Guarded Transition Systems

The formal trace semantics of the g-HMSC process language enables a variety of checks on process models such as, e.g., the verification of guard completeness and disjointness in the current process state at a decision node, the verification of task preconditions, the verification of non-functional process requirements involving time or resources, and so forth. The overall approach proceeds in three steps.

1. A g-LTS is first generated from the g-HMSC model using the technique described in [25].
2. Each state of this g-LTS is automatically decorated with quantities that are meaningful to the specific type of check being considered.
3. The computed quantities are used to perform the corresponding check.

This section focuses on the computation of state decorations in Step 2. This is the core step common to all different types of checks discussed in Section VII. Depending on the specific check considered, the decorations might refer to assertions, such as state invariants, or to quantities such as the time elapsed from the initial state to the current process state, the cost incurred up to the current node, the radiation dose received by a patient so far, and so forth. Rather than different decoration algorithms for different types of decorations and checks, a single uniform treatment is provided. The meaningful decorations are made generic through so-called placeholders; the generic algorithm described in this section works then for specific placeholder instantiations.

- A *placeholder* is a generic variable whose value in a specific g-LTS state characterizes this state whatever path has been followed in the model to reach it from the initial state.
- A *decoration* of a g-LTS state is a mapping from the values of process variables in this state to the corresponding placeholder values.

A decoration of a g-LTS state is intended to keep track of which value assignment of fluents and tracking variables corresponds to which placeholder value in this state. Such mapping is needed to account for guard satisfaction along the paths through which decorations are to be propagated. To visualize this, suppose that a state is decorated with an interval; the meaning is that the placeholder may in this state have any element within this interval as value. In Fig. 5(a), we cannot determine the placeholder values right after guards g or $\neg g$ without knowing the relation between the value of variable g and the valid interval for such value. In Fig 5(b), we have this information; e.g., when variable g has the value *false* in the source state, the placeholder may belong to the interval *[3, 5]* in this state. We can then propagate that information to determine the resulting decoration for the target states.

Our generic decoration algorithm propagates state decorations through the g-LTS model until a *fixpoint* is reached, that is, until no state decoration changes if the propagation is applied once more to every transition [41, 51, 61]. The algorithm accumulates in every state the decorations contributed by every g-LTS execution reaching this state.

To explain this algorithm piecewise, Section VI.A first deals with propagation of placeholder values along a single g-LTS execution regardless of corresponding assignments of fluents and tracking variables. Keeping track of those assignments along a single g-LTS execution is explained separately in Section VI.B. The decoration lattice structure manipulated by the algorithm is defined in Section VI.C. The propagation of decorations is then detailed in Section VI.D. The generic decoration algorithm is finally provided in Section VI.E.

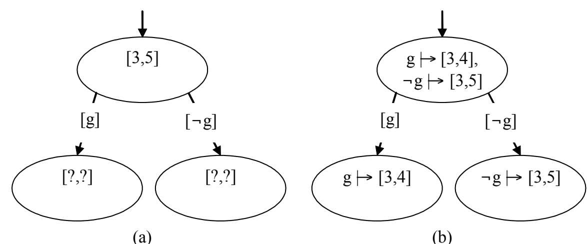

Fig. 5. Propagating decorations through guards

A. Placeholder values along a single g-LTS execution

Let *PlcV* denote the set of possible values of a placeholder. In this set, p_0 denotes the value of the placeholder at the initial process state; the latter corresponds to an empty g-LTS execution.

The *plcPropag* function specifies how a placeholder value is propagated through a single event-labelled transition. This value is assumed to depend only on the incoming event and the value in the preceding state. Placeholder values may thus not depend on subsequent events –e.g., the remaining time before reaching a subsequent task *T* cannot be captured as a placeholder instantiation.

The *plcPropag* function has the following signature:

$$plcPropag\colon\ PlcV \times E \rightarrow \mathcal{P}(PlcV).$$

As the powerset notation in the codomain of this function indicates, a placeholder can have multiple propagation results. This happens when a single value cannot be deterministically determined after the occurrence of an event. For example, the duration of a task might vary depending on the agent instances performing it; the duration is then captured by an interval.

The placeholder values p_i along a single g-LTS execution are recursively determined as follows:

- in the *initial* state, the placeholder value is p_0;
- through a *guard*, the value does not change, that is, $p_i = p_{i-1}$;
- through an *event e*, the value belongs to $plcPropag\,(p_{i-1}, e)$.

The set *PlcV* of placeholder values and the *plcPropag* function are parameters of the generic state decoration algorithm. Any algorithm instantiation for a specific analysis requires instantiating *plcPropag* through a specific propagation rule on a specific set of instantiated placeholder values.

Example: Instantiation for elapsed time. To capture the elapsing time through tasks having some minimum/maximum duration, *PlcV* is instantiated to the set N^+ of natural numbers. The value of the placeholder after an empty g-LTS execution is $p_0 = 0$. The *plcPropag* function is instantiated as follows:

$$plcPropag : N^+ \times E \rightarrow \mathcal{P}(N^+),$$
$$plcPropag\,(p, e) = p \oplus Duration\,(e),$$

where the ⊕-operator "adds" a duration interval to a time point according to the following definition:

$$\oplus\colon\ N^+ \times \mathcal{P}(N^+) \rightarrow \mathcal{P}(N^+),$$
$$t \oplus [\min, \max] = [t + \min, t + \max].$$

Handling unbounded g-LTS executions over infinite domains. As detailed below, the decoration algorithm propagates placeholder values from state to state until the values remain unchanged if the propagation is applied once more to every transition. Such fixpoint might not be reached for process models containing unbounded cycles if the set *PlcV* of possible placeholder values is infinite. In such cases, the algorithm might not terminate.

To avoid this problem, the user is asked in such situations to provide a finite subset of *PlcV*, denoted by *finitePlcV*. If there are g-LTS executions yielding placeholder values out of *finitePlcV*, the algorithm outputs the element *OutOfBounds* for such executions. The set of placeholder values possibly produced by the algorithm, denoted by *boundedPlcV*, is thus:

$$boundedPlcV = finitePlcV \cup \{OutOfBounds\}.$$

The algorithm therefore relies on a bounded version of the *plcPropag* function previously introduced. The *plcPropag** function "wraps" *plcPropag* while restricting output values to *finitePlcV* and using the *OutOfBounds* marker when required:

$$plcPropag^*\colon\ boundedPlcV \times E \rightarrow \mathcal{P}(boundedPlcV)$$

$plcPropag^*(p, e) =$
  if $p = OutOfBounds$ **then return** $\{OutOfBounds\}$
  else $vals := plcPropag\,(p, e)$
    if $vals \subseteq finitePlcV$ **then return** $vals$
    else return $(vals \cap finitePlcV) \cup \{OutOfBounds\}$.

The correctness proof of the generic decoration algorithm, outlined in Section IX.A, requires propagations of elements outside *finitePlcV* to remain outside *finitePlcV*:

for any $p \in PlcV \setminus finitePlcV$ and any $e \in E$:
$$plcPropag\,(p, e) \cap finitePlcV = \emptyset.$$

The impact of this hypothesis is limited in practice. Getting out of *finitePlcV* might occur when the process model contains unbounded cycles. For medical processes, this generally corresponds to a modeling error, detected by our approach as a side effect. When all executions are finite and the bounds of *finitePlcV* are nevertheless reached, a larger *finitePlcV* may be taken to meet the hypothesis.

Propagating sets of placeholder values. In Section VI.D hereafter, placeholder propagation needs be applied to a set of placeholder values (rather than a single value). The *plcPropag** function is extended for this into the *plcSetPropag** function defined as follows:

$$plcSetPropag^*\colon\ \mathcal{P}(boundedPlcV) \times E \rightarrow \mathcal{P}(boundedPlcV)$$
$$plcSetPropag^*(set, e) = \bigcup_{p \in set} plcPropag^*(p, e)$$

B. Values of process variables along a single g-LTS execution

As introduced at the beginning of Section VI, we need to know in each g-LTS state what the values of fluents and tracking variables are; a placeholder value should not be propagated through a guard if the current assignment of those variables does not satisfy the guard (see Fig. 5).

Let *AsgV* denote the set of possible value assignments of fluents and tracking variables. A specific assignment *asg* is a set of elements of form $x \mapsto v$ where *x* denotes a fluent or a tracking variable and *v* denotes the value *true* or *false*. It is often convenient to use the equivalent propositional form instead, namely,

$$asg =_{\text{def}} \textstyle\bigwedge_j x^t_j \wedge \bigwedge_k \neg x^f_k,$$

where x^t_j and x^f_k denote the variables whose assigned value is *true* and *false*, respectively.

The *asgPropag* function specifies how a value assignment for fluents and tracking variables is propagated through a single event-labelled transition:

$$asgPropag\colon\ AsgV \times E \rightarrow \mathcal{P}(AsgV).$$

Here again, the powerset notation in the codomain of this function indicates that a value assignment can have multiple propagation results; for a single assignment $tv \mapsto v$ of a tracking variable *tv*, the propagation through a measure event of *tv* yields the two assignments $tv \mapsto v$ and $tv \mapsto \neg v$.

Unlike the propagation of placeholder values to be instantiated for a specific analysis, the propagation of value assignments always follows the same rule, namely,

$$asgPropag\,(asg, e) = asg\,[/V_e] \wedge \bigwedge_i f_{i,e} \wedge \bigwedge_t \neg f_{t,e},$$

where

- $asg\,[/x] =_{\text{def}}\ asg\,|_{x=true} \vee asg\,|_{x=false}$;
- $B|_{x=v}$ is the Boolean formula B where every occurrence of variable x has been replaced by the value v;
- for a set V of variables $x_1, x_2, \ldots, x_n$, the notation generalizes through composition:
  $$B\,[/V] =_{\text{def}} B\,[/x_1][/x_2]\ldots[/x_n];$$
- V_e is the set of variables affected by event e, that is,
  $$V_e = F_{i,e} \cup F_{t,e} \cup TV_e\,,$$
  where
  - $F_{i,e}$ is the set of fluents $f_{i,e}$ having e among their initiating events;
  - $F_{t,e}$ is the set of fluents $f_{t,e}$ having e among their terminating events;
  - TV_e is the set of tracking variables having e among their measure events.

Example. Consider the following process variables appearing in Fig. 1:

fluent *evaluated* = < {*Evaluation*$_{end}$}, {*Medical Treatment*$_{end}$} >
trackVar *observation_required* = {*Evaluation*$_{end}$}.

Suppose that the current value assignment is:

$$asg\ =\ \neg\ evaluated \wedge \neg\ observation_required.$$

After the event *Evaluation*$_{end}$ this assignment becomes:

$$\begin{aligned} &asgPropag\ (\neg evaluated \wedge \neg observation_required,\ Evaluation_{end}) \\ &= (\neg evaluated \wedge \neg observation_required)\ [/\ \{evaluated,\ observation_required\}] \\ &\quad \wedge\ evaluated \\ &= evaluated. \end{aligned}$$

This means that any patient following the process is now evaluated; one patient might be required to undergo an observation whereas another one not.

The value assignments asg_i along a single g-LTS execution are now recursively determined as follows:

- in the *initial* state, the assignment is the initial assignment *Init*;
- through a *guard*, the assignment remains the same, that is, $asg_i = asg_{i-1}$;
- through an *event* e, the new assignment belongs to $asgPropag\,(asg_{i-1}, e)$;
  - all fluents whose set of initiating (resp. terminating) events contains e are made *true* (resp. *false*);
  - all tracking variables whose set of measure events contains e are made non-deterministically *true* or *false*.

C. The decoration lattice

The decoration of a g-LTS state was informally introduced before as a mapping from the values of fluents and tracking variables in this state to the corresponding placeholder values. More precisely, it is a partial function with the following signature:

$$dec\colon\ AsgV \rightarrow \mathcal{P}(boundedPlcV).$$

(Remember that *AsgV* denotes the set of possible value assignments of fluents and tracking variables.)

Let **dom** *dec* denote the subset of *AsgV* where decoration *dec* becomes a total function; let **img** *dec* denote the subset of corresponding images in $\mathcal{P}(boundedPlcV)$.

State decorations form a lattice ($\leq, \wedge, \vee, \perp, T$) defined as follows:

$$\begin{aligned} &d_1 \leq d_2 \quad \text{if} \quad \mathbf{dom}\, d_1 \subseteq \mathbf{dom}\, d_2 \\ &\qquad\qquad \text{and } \forall asg \in \mathbf{dom}\, d_1\colon\ d_1(asg) \subseteq d_2(asg) \\ &\perp = \varnothing \\ &T = \{\, asg \mapsto BoundedPlcV \mid asg \in AsgV \,\} \\ &d_1 \vee d_2 = \{\, asg \mapsto d_1(asg) \cup d_2(asg) \mid asg \in \mathbf{dom}\, d_1 \cap \mathbf{dom}\, d_2 \,\} \\ &\qquad \cup \{\, asg \mapsto d_1(asg) \mid asg \in \mathbf{dom}\, d_1 \setminus \mathbf{dom}\, d_2 \,\} \\ &\qquad \cup \{\, asg \mapsto d_2(asg) \mid asg \in \mathbf{dom}\, d_2 \setminus \mathbf{dom}\, d_1 \,\} \\ &d_1 \wedge d_2 = \{\, asg \mapsto d_1(asg) \cap d_2(asg) \mid asg \in \mathbf{dom}\, d_1 \cap \mathbf{dom}\, d_2 \,\} \end{aligned}$$

The "$\vee$" supremum operator on this lattice is of particular importance. As shown in the next sections, the decoration algorithm uses this operator to propagate decorations through transitions and accumulate propagation results in each state.

Example: Supremum for elapsed time. Consider a g-LTS with a single fluent *evaluated.* One state might be decorated with

$$\{evaluated \mapsto \{2,3\},\ \neg\ evaluated \mapsto \{4\}\},$$

meaning that any process execution reaching this state takes 2 or 3 time units if the patient is evaluated, and exactly 4 time units if she is not evaluated. Taking the supremum of this decoration with the decoration

$$\{\text{evaluated} \mapsto \{2,5\}\},$$

we get:

$$\{evaluated \mapsto \{2,3,5\},\ \neg\ evaluated \mapsto \{4\}\}.$$

D. Propagating state decorations along a g-LTS execution

Sections VI.A and VI.B separately discussed the propagation of placeholder values and the propagation of value assignments of process variables, respectively. As a state decoration is a mapping from the latter to the former, this section integrates the two mechanisms for decoration propagation.

Let *DecoV* denote the set of possible values for state decorations, that is, the set of partial functions capturing decorations (see Section VI.C). The *decoPropag* function specifies how a state decoration is propagated through a single transition:

$$decoPropag\colon\ DecoV \times (E \cup GUARD) \rightarrow DecoV.$$

A g-LTS transition is labeled by a guard or by an event.

- Through a *guard* g, the algorithm may only propagate decorations that meet g:
  $$decoPropag\,(dec, g) =\ g \lhd dec,$$
  where "$\lhd$" denotes the domain restriction operator; $g \lhd dec$ restricts the domain of the decoration function *dec* to those assignments meeting guard g.
- Through an *event*, both the domain **dom** *dec* and the image set **img** *dec* of decoration *dec* must be updated. The former is updated according to the *asgPropag* function introduced in Section VI.B for propagating value assignments; the latter is updated according to the *plcPropag* function introduced in Section VI.A for propagating placeholders:

$$decoPropag(dec, e) = \bigvee_{asg \in \mathbf{dom}\, dec} \{ a \mapsto plcSetPropag^*(dec(asg), e) \mid a \in asgPropag(asg, e) \},$$

where *plcSetPropag** is the bounded propagation function over sets of placeholder values introduced at the end of Section VI.A and *asgPropag* is the propagation function over value assignments of process variables introduced in Section VI.B. Given a decoration and an event, *decoPropag* is computed assignment-wise from the decoration domain (see Fig. 6). The results are accumulated through the "∨" lattice supremum operator.

Example for elapsed time. Back to our running example, consider a state decorated with

$$\{evaluated \mapsto \{2,3\},\ \neg\ evaluated \mapsto \{4\}\}.$$

Suppose that we want to propagate this decoration through the guard *[¬ evaluated]*. In this case, the decoration of the subsequent state is obtained through the ◁-operator:

$$decoPropag(\{evaluated \mapsto \{2,3\},\ \neg\ evaluated \mapsto \{4\}\},\ [\neg\ evaluated])$$
$$= \neg\ evaluated \mapsto \{4\}.$$

Suppose now that we want to propagate the same decoration though the event $Evaluation_{end}$. The latter is an initiating event for fluent *evaluated*; its duration is between 2 and 3 time units. The decoration of the subsequent state is therefore:

$$decoPropag(\{evaluated \mapsto \{2,3\},\ \neg\ evaluated \mapsto \{4\}\},\ Evaluation_{end})$$
$$= \{ evaluated \mapsto \{4,5,6\} \vee evaluated \mapsto \{6,7\} \}$$
$$= \{ evaluated \mapsto \{4,5,6,7\} \}$$

E. The generic decoration algorithm

For any state *s*, the decoration computed by the algorithm should be the *most accurate one* in the following sense.

a) *Not-too-general:* any generated placeholder value should be produced by at least one g-LTS execution reaching *s*.
b) *Not-too-specific:* any placeholder value produced by a g-LTS execution reaching state *s* should be generated.

To make the notion of most accurate decoration further precise, let us introduce the *stateDeco* function mapping every g-LTS state to its generated decoration:

$$stateDeco\colon\ S \rightarrow DecoV,$$

where *DecoV* was introduced before as the set of partial functions mapping assignments of process variables to placeholder values. Let us also consider the set of possible placeholder values in the decoration of state *s* regardless of the corresponding value assignments of process variables in that state:

$$Placeholders(s) = \bigcup_{pl \in \mathbf{img}\ stateDeco(s)} pl$$

The two preceding conditions for a generated decoration *stateDeco (s)* to be the *most accurate* one are made more precise as follows.

a) Every element of *Placeholders(s)* must be a placeholder value produced by at least one g-LTS execution reaching *s*. If *OutOfBounds* ∈ *Placeholders(s)*, there must exist at least one g-LTS execution yielding a placeholder value outside *finitePlcV*.
b) For every placeholder value *p* produced by a g-LTS execution reaching *s*, we must have:
   - $p \in Placeholders(s)$ if $p \in finitePlcV$,
   - $OutOfBounds \in Placeholders(s)$ otherwise.

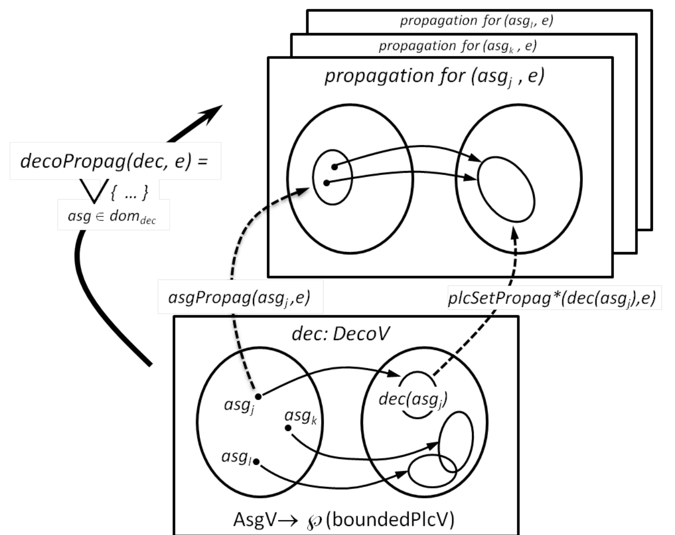

Fig. 6. Propagating decorations using the *decoPropag* function

The specification of the decoration algorithm is the following.

GIVEN

- a g-LTS (S, E, VAR, δ, s_0, C_0),
- a set *PlcV* of placeholder values with finite subset *finitePlcV,*
- a placeholder propagation function
  $$plcPropag\colon PlcV \times E \rightarrow \mathcal{P}(PlcV),$$
- a value p_0 for the placeholder in the initial state ($p_0 \in finitePlcV$),

FIND the most accurate decoration *stateDeco (s)* for every g-LTS state *s*.

The decoration algorithm is given in Fig. 7. The symbols ≤, ∨ and ⊥ there correspond to the partial order, supremum operator and bottom element of the lattice introduced in Section VI.C, respectively. The algorithm propagates decorations from state to state until a fixpoint is reached, that is, until no state decoration changes if the propagation is applied once more to every transition.

The initial decoration maps all value assignments satisfying the initial condition to the initial placeholder value p_0. It is used for the initial g-LTS state s_0. Other states are initially decorated with the bottom element of the decoration lattice.

The algorithm keeps track of the set *ToExpl* of states to which the propagation should be further applied. It terminates when the set *ToExpl* is empty, that is, when no state requires further decoration propagation.

For a given source state in *ToExpl*, its decoration is successively propagated to successor states through the *decoPropag* function introduced in Section VI.D. If the propagation result is not already covered by the decoration of the target state, it is accumulated through the lattice supremum operator. The target state is then added to the states to be further explored.

The correctness proof of this algorithm is outlined in Section IX.A; the full details are found in [21].

stateDeco (s_0) := { (x ↦{p_0}) | x ⊨ C_0 }
forall $s \in S \setminus \{s_0\}$ **do**
  stateDeco (s) := ⊥
ToExpl := {s_0}
while *ToExpl* ≠ ∅ **do**
  source := getOne (*ToExpl*)
  ToExpl := *ToExpl* \ {*source*}
  forall (*source*, *target*, *label*) ∈ δ **do**
    newVal := *decoPropag* (stateDeco (*source*), *label)*
    if not (*newVal* ≤ stateDeco (*target*)) **then**
      stateDeco (*target*):= *newVal* ∨ stateDeco (*target*)
      ToExpl := *ToExpl* ∪ {*target*}
return stateDeco

Fig. 7. Generic algorithm for g-LTS decoration

F. Instantiating the decoration algorithm for specific analyses

The generic decoration algorithm is designed to be applicable to a wide variety of analyses on the g-LTS generated from a g-HMSC process model. For each type of analysis, the same instantiation steps are required.

1. The placeholder domain *PlcV* is instantiated so as to meet the desired type of analysis. If *PlcV* is not finite, a finite subdomain is defined. An initial placeholder value p_0 is then selected.
2. The *plcPropag* function is instantiated by specifying how instantiated placeholder values get transformed through a single event-labelled transition.

These steps were already illustrated in Section VI.A for the time elapsing through tasks having some minimum/maximum duration. The next section illustrates the whole instantiation process on a variety of analyses.

VII. Decoration-Based Analysis of Process Models

This section presents instantiations of the generic decoration algorithm for a variety of analyses on process models. A first instantiation concerns the generation of state invariants (Section VII.A). Section VII.B shows how the guards at a decision node can be checked for disjointness, completeness and satisfiability in the current state where the decision is made. Section VII.C shows how preconditions on tasks can be verified or even generated. Section VII.D describes a technique for verifying time constraints and other resource-related requirements on the process model. Finally, Section VII.E presents a technique for checking whether the decision made at a decision node is adequate, that is, whether the values of the process variables are sufficiently accurate in the current state where the decision is made.

A. Generating state invariants

A *state invariant* is an assertion on a specific state of the process that holds every time this state is visited. The *most accurate* state invariant at a state is the one that accounts for all possible process executions reaching that state and only those. This invariant will be referred to as *MAS invariant* for short.

The annotation of process models with MAS invariants provides multiple benefits.

- The documentation and understandability of the process model is improved.
- The invariants can be used for model validation and anomaly detection.
- The invariants can be exploited by analysis tools for increased efficiency [9, 41].
- Other process analysis techniques may rely on them (see Sections VII.B, VII.C, and VII.E).

1) Placeholder instantiation

To generate MAS invariants over process variables, the generic decoration algorithm is instantiated in a quite simple way. No specific placeholder is required as we only need information about possible values of fluents and tracking variables.

- *PlcV* contains one single element, p_0;
- *plcPropag* is the identity function.

2) Using decorations for state invariant generation

The instantiated decoration algorithm generates the possible value assignments of process variables at each state, see Section VI.B. The MAS invariant *SI (s)* for state *s* is then simply obtained by taking the disjunction of all values obtained for this state:

$$SI(s) = \bigvee_{asg \in \mathbf{dom}\ stateDeco(s)} asg.$$

3) Example

Each node of the g-LTS model in Fig. 4 may thereby be annotated with its MAS invariant. For example, the invariant generated at state 15 is:

evaluated ∧ *consultation_done* ∧ *observation_required.*

Note that the assertion

evaluated ∧ *consultation_done*

is also a state invariant but not the most accurate one.

B. Analyzing guards at decision nodes

The following guard-related checks are worth considering in decision-based process models.

- *Guard completeness:* The guards on the various alternatives at a decision node must cover all possible cases in the state reached at that node; no alternative branch may be missing. Otherwise the process might be blocked forever in that state with no guard evaluated to *true* and, correspondingly, no task being prescribed when the missing guard is true.
- *Guard disjointness:* The guards on the various alternatives at a decision node may not overlap in the state reached at that node; two guards may not be both evaluated to *true* in that state. Non-deterministic decisions where different courses of action are taken by different process instances applied to right the same situation are most often to be precluded. Clinical guidelines, for example, prescribe patients in identical conditions to be treated identically.
- *Guard satisfiability:* The guards on the various alternatives at a decision node must all be satisfiable in the state reached at that node. Otherwise the subsequent tasks prescribed along some branch would be unreachable.

Such checks are close in spirit to those in [36] for SCR tables. Beyond different formalisms there is a notable difference, however. The checks here must take into account the *contextual conditions* holding at the point in the process model where the decision node is reached. These conditions are captured by the MAS invariant at that point.

1) Placeholder instantiation

The instantiation for guard analysis is therefore similar to the one in Section VII.A for generating MAS invariants on fluents and tracking variables.

2) Using decorations for guard analysis

The MAS invariant right before the decision node is first generated. The result of this step is used in various types of satisfiability checks at the next step. Let s_D denote the state reached at decision node D and let $SI(s_D)$ denote the generated MAS invariant at s_D.

(SC1) For *guard completeness* at D, we need to check that:

$$SI(s_D) \models \bigvee_i g_i.$$

In case of incompleteness, our tool returns all value assignments of process variables that are not covered by the guards g_i on outgoing branches from D.

(SC2) For *guard disjointness* at D, we need to check that:

for every pair of guards g_i, g_j from D:

$$g_i \wedge g_j \wedge SI(s_D) \models false.$$

In case of overlap, the tool returns all value assignments of process variables that satisfy several guards at D.

(SC3) For *guard satisfiability* at D, we need to check that:

for every guard g from D: $g \wedge SI(s_D)$ is satisfiable.

In case of unsatisfiability, the tool indicates outgoing transitions that are unreachable.

Checks (SC1)-(SC3) are fairly simple once the contextual MAS invariant $SI(s_D)$ is generated as a first step thanks to the decoration algorithm.

3) Example

Consider the top decision node of the process model in Fig. 1. Let us assume that the initial context condition on this model is ¬ *evaluated.* The guards on the three outgoing transitions are: ¬ *consultation_done, consultation_done* ∧ ¬ *evaluated*, and *evaluated.* The generated MAS invariant right before the decision node is: ¬ *evaluated*. The three guards are verified to be complete and disjoint. However, the tool points out that the third guard is unsatisfiable in view of the generated invariant.

The problem is easily fixed by removing the corresponding guarded transition. The second guard may then be simplified into *consultation_done.*

C. Verifying or generating task preconditions

As discussed in Section IV.E, tasks in a g-HMSC process model may be annotated with their precondition. It is worth checking that task preconditions cannot be violated through some path in the model. Even better, we may want to generate them automatically when they are not given.

1) Placeholder instantiation

Here again, the first step consists of generating the MAS invariant for each state. The instantiation is therefore similar to the one described in Section VII.A.

2) Using decorations to check or generate preconditions

Checking that the precondition PRE_T of task T is never violated amounts to the following satisfiability check performed as a second step:

for every source node s of a T_{start} event: $SI(s) \models PRE_T$,

where $SI(s)$ denotes the MAS invariant generated at this node.

When task preconditions are not provided by the modeler, the tool uses the same decorations to generate them. For a task T, it retrieves all source states of T_{start} events and takes the disjunction of their MAS invariants.

The preconditions thereby inferred may need to be further simplified to remove redundant parts that state known domain properties.

3) Example

Let us assume that the *Medical Treatment* task at the bottom of Fig. 1 has been annotated with *evaluated* as precondition. The precondition check reveals a violation; medical treatment can be prescribed without evaluation when the patient is in danger (see Fig. 1).

Two alternative resolutions might fix the problem: (a) change the precondition or (b) require an evaluation before medical treatment of patients in danger. If (a) is selected, the new precondition may be inspired from the one generated by the tool, namely,

¬ *evaluated* ∧ ¬ *consultation_done* ∧ *patient_in_danger*
∨ *consultation_done* ∧ *evaluated*
∧ *medical_treatment_required*

4) Discussion

Guard analysis and precondition checks might at first glance appear fairly obvious types of checks. Such feeling may arise from the simplicity of the second instantiation step once the MAS invariant has been generated as a first step. Thanks to the latter, the second step must not deal with the intricacies of propagating possible values of process variables along all guard-satisfying paths leading to the considered state; it must only deal with propagations of placeholder values through events, which is much simpler here as *plcPropag* is instantiated to the identity function. Moreover, the first invariant generation step is common to both techniques and can therefore be shared among them.

The examples of detected errors provided as illustrations might appear fairly simple as well. They are however representative of uncovered errors found in real process documentations used daily in clinical environments (see Sections IX.C and IX.D).

D. Verifying non-functional requirements involving time or resources

Safety-critical processes such as those found in the medical domain often involve critical time constraints. The latter should be inferred from the corresponding model or verified on it. For example,

- we may want to know how long a process can take in best-case or worst-case situations;
- we may want to check whether strict timing requirements on a task or process are always met by the model.

General time-related properties can be verified on behavior models by dedicated tools such as temporal model checkers

[53]. The instantiation in this section provides a more lightweight technique for checking or infering specific types of time constraints on process models. As discussed later, the instantiation extends to other resource-related requirements on process models and can easily be combined with other instantiations of the generic algorithm for other types of checks.

1) Placeholder instantiation

The placeholder here captures the discrete time elapsed from the initial state. This instantiation was already introduced as an illustration at the end of Section VI.A.

- *PlcV* is the set N^+ of naturals. The initial elapsed time is $p_0 = 0$.
- The placeholder propagation function is instantiated as follows:

$$plcPropag(p, e) = p \oplus Duration(e),$$
$$\text{with } \oplus: N^+ \times \mathcal{P}(N^+) \rightarrow \mathcal{P}(N^+)$$
$$t \oplus [min, max] = [t + min, t + max]$$

- As *PlcV* here is infinite, a finite subset is taken by restricting *PlcV* to the interval *[0, TMAX]* where *TMAX* is some user-defined upper time bound.

The instantiated decoration algorithm can be used for checking a variety of time-related properties. In particular, inferring or verifying the *minimum* and *maximum time taken by a process or by a composite task* is achieved by looking at the fixpoint decoration generated at the last process/task state (e.g., state 18 in Fig. 4); the least and greatest time points are then taken in the image set of the decoration function in this state.

As discussed in Section VI.A, getting *OutOfBounds* within output decorations means that either the process contains undesired infinite executions or the *TMAX* upper bound is not large enough. In the latter case, a larger value should be estimated based on known maximum durations of tasks and quick checks on task loops.

2) Instantiated algorithm in action

Let us illustrate the first steps of the instantiated algorithm on the simple example shown in Fig. 8.

Step 0: The decorations are initialized to ⊥ for each state except for the initial one whose decoration is the function mapping every value assignment meeting the initial context C_0 to the initial placeholder value:

$$stateDeco(State0) = \{evaluated \mapsto \{0\},\ \neg\ evaluated \mapsto \{0\}\}.$$

Step1: *State0* is in *ToExpl*. Following the algorithm in Fig. 7, its decoration is propagated to all its successor states, that is, *State1* and *State2*. For *State1*, since ¬ *evaluated* is a guard, the decoration of *State0* gets restricted to the specific value assignment ¬ *evaluated*. We obtain:

$$stateDeco(State1) = \{\neg\ evaluated \mapsto \{0\}\}.$$

Similarly, for *State2* we obtain:

$$stateDeco(State2) = \{evaluated \mapsto \{0\}\}.$$

State1 and *State2* are added to *ToExpl* since their decoration has changed.

Step 2: Assume that *State1* is selected in *ToExpl*. Its decoration is propagated to its only successor, *State0*. The *evaluation* event is an initiating one for fluent *evaluated*; its duration is in the interval [2, 3]. Since

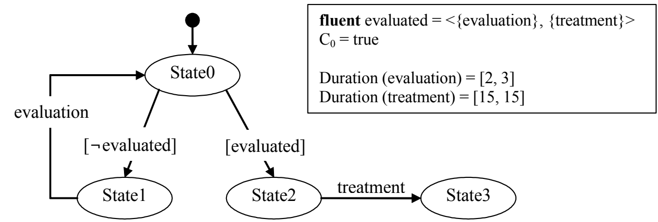

Fig. 8. Executing the algorithm for decorating g-LTS states with time

$$asgPropag(\neg\ evaluated, evaluation) = evaluated,$$
$$plcSetPropag^*(\{0\}, evaluation) = \{2, 3\},$$

we obtain:

$$decoPropag(\{\neg\ evaluated \mapsto \{0\}\}, evaluation) = \{evaluated \mapsto \{2, 3\}\},$$

and thus:

$$newVal = \{evaluated \mapsto \{2, 3\}\}.$$

The new decoration of *State0* is computed from its old decoration and *newVal*:

$$stateDeco(State0) = \{evaluated \mapsto \{0\} \cup \{2, 3\}\} \cup \{\neg\ evaluated \mapsto \{0\}\}$$
$$= \{\neg\ evaluated \mapsto \{0\},\ evaluated \mapsto \{0, 2, 3\}\}.$$

State0 is added to *ToExpl* as its decoration has changed.

Step 3: Assume that *State0* is selected in *ToExpl*. Its decoration is propagated to *State1* and *State2*. For *State1*, *newVal* has the same value as its decoration; no update is required. For *State2*, we obtain:

$$newVal = \{evaluated \mapsto \{0, 2, 3\}\}.$$

The new decoration of *State2* gets the value of *newVal*:

$$stateDeco(State2) = \{evaluated \mapsto \{0, 2, 3\}\}.$$

State2 is added to *ToExpl*.

Step 4: *State2* is the only element remaining in *ToExpl*. Its decoration is propagated to *State3*. The *treatment* event is among the terminating ones of fluent *evaluated*; the duration of *treatment* is in the interval *[2, 3]*. Therefore, *State3* is decorated with:

$$stateDeco(State3) = \{\neg\ evaluated \mapsto \{15, 17, 18\}\}.$$

State3 is added to *ToExpl*. As *State3* was the only element remaining in *ToExpl* and has no successor, the algorithm terminates.

3) Example

For the process in Fig. 1, the minimum overall process time returned by the instantiated algorithm is *3 days* whereas the maximum one is *OutOfBounds*. This means that the treatment might continue forever for patients whose variable *medical_treatment_required* remains indefinitely *true* (see Fig. 1). This property should be validated with medical staff. If process executions are expected to be finite, the model should be changed accordingly to avoid this.

4) Other time-related requirements

The preceding technique may also be used to verify or infer temporal requirements on processes or tasks. The annotation of a refined task by a time constraint might be either a requirement we would like to impose on it or a preliminary estimate to be replaced by a more accurate duration interval inferred from the refinement of the task.

In the latter case, we need to locally apply the instantiated decoration algorithm on the refining subprocess. This requires the initial context condition C_0 to be known for this specific subprocess. The latter condition is directly obtained by use of the other algorithm instantiation described in Section VII.A.

Yet another check variant consists in checking time bounds for specific process paths. To achieve this, the initial values of fluents and tracking variables at the initial state are restricted by strengthening the initial context condition C_0 so as to cover those paths specifically.

5) Resource-related requirements

The instantiation of the generic decoration algorithm discussed in this section can be extended for reasoning about other cumulative properties involving costs, resource consumption, doses in a medical process (e.g., drug doses or radiation doses), and so forth. Non-functional requirements on resource usage are then checked similarly. Section IX shows this on an example of underdose detection.

E. Checking the adequacy of decisions

As discussed in Section IV.C, tracking variables are intended to capture relevant quantities in the environment of the process. It is therefore worth checking that *every time a tracking variable is used in the process model its value accurately reflects the actual corresponding value in the process environment.* This is particularly important for tracking variables appearing in decision nodes; inaccurate values for tracking variables at these nodes may result in *inadequate decisions* where the outgoing branch taken might differ from the one that should have been taken with accurate values.

Inadequate decisions may result from two different sources of inaccuracy.

- *Task-dependent inadequacies*: some tasks affect the environment quantity without updating the tracking variable accordingly.
- *Time-dependent inadequacies*: the value of a tracking variable becomes outdated after some time.

Checking the process model for these two types of inadequacies require different instantiations of the generic decoration algorithm.

1) Task-dependent decision inadequacies

In a *task-dependent inadequacy*, the decision relies on inaccurate information about the environment due to the occurrence of intermediate interfering events.

It is therefore worth checking whether each tracking variable appearing in the guard on an outgoing branch of a decision node is accurate in the source state of the corresponding guarded transition. To achieve this, a so-called *accuracy meta-fluent* is introduced.

- In states where this fluent is *true*, the value of the associated tracking variable accurately reflects its environment counterpart.
- In states where the fluent is *false*, the value might be inaccurate.

Every tracking variable *tv* in the process model has an associated accuracy meta-fluent *tv-acc* defined as follows.

- The set $Init_{tv\text{-}acc}$ of initiating events contains its *measure* events, that is, all the events synchronizing the tracking variable with its environment counterpart (see Section IV.C and Fig. 3).
- The set $Term_{tv\text{-}acc}$ of terminating events contains its *outdate* events, that is, all the events potentially affecting the environment quantity without synchronizing the tracking variable with its environment counterpart.
- The initial value is always *false*.

A decision adequacy check on a specific tracking variable amounts to checking whether its accuracy meta-fluent is *true* at the source node of the corresponding guarded transition. Note that accuracy meta-fluents are not visible at the g-HMSC level. They are automatically derived from the definitions of tracking variables in order to perform adequacy checks at the g-LTS level.

a) Placeholder instantiation

The placeholder here captures the value of the accuracy meta-fluent associated with the target tracking variable *tv*.

- $PlcV = finitePlcV = \{true, false\}$; $p_0 = false$.
- The placeholder propagation function is instantiated as follows:

$$plcPropag\,(p, e) = \begin{cases} \{true\} & \textbf{if } e \in Ini_{tv\text{-}acc}, \\ \{false\} & \textbf{if } e \in Term_{tv\text{-}acc}, \\ \{p\} & \textbf{otherwise}. \end{cases}$$

b) Using decorations for decision adequacy checking

Once the instantiated decorations are thereby computed, the following formula must be verified for every source state *source* and tracking variable *tV* appearing in the guard on an outgoing transition:

$$false \notin Placeholders\,(\text{stateDeco}\,(source)),$$

where *Placeholders (s)* was defined in Section VI.E as the set of possible placeholder values in the decoration of state *s* regardless of the corresponding value assignments of process variables in that state.

If this formula is not satisfied, 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 may therefore be inadequate. The class of patients affected by such inadequate decisions is given by **dom** *stateDeco (source)*.

Inadequate decisions should be resolved in a corrected version of the process model. A simple resolution heuristics consists in *adding a task right before the decision node that includes a measure event for the problematic tracking variable.*

c) Example

Consider the tracking variable *medical_treatment_required* in the process model for treating acute manic-depressive troubles in Fig. 1. A value *true* means that the patient needs to undergo a *Medical Treatment* task. This value is expected to be accurate right after the *Evaluation* task. It becomes inaccurate when the medical treatment is terminated; a new evaluation of the patient is required in order to decide whether the treatment should continue. The tracking variable is therefore defined in

terms of *measure* and *outdate* events as follows (see Section IV.C):

trackVar *medical_treatment_required* =
{$Evaluation_{end}$}
{$Medical\ Treatment_{end}$}.

Checking the decision node *medical_treatment_required* in Fig. 1 reveals that the decision captured there is adequate. The decoration of *state 17* in Fig. 4 tells us that the meta-fluent *medical_treatment_required-acc* is *true* at this point; an evaluation is always performed before making the decision (see Fig. 1).

Section IX shows an example where this check reveals an inadequate decision in the documentation of a real safety-critical process.

2) Time-dependent decision inadequacies

It is often the case that tracking variables remain accurate for a certain period of time only. The environment quantities they reflect may change beyond that period due to potential events that are unobservable by the process agents. In a *time-dependent decision inadequacy*, the decision relies on out-of-date information about the environment.

To check the process model for such situations, the accuracy duration of the considered tracking variable is taken into account in the meta-fluent definition (see Section IV.C):

fluent *tV-acc* = <$Init_{\text{tv-acc}}$, $Term_{\text{tv-acc}}$> **duration** $Dur_{\text{tv-acc}}$.

The fluent then holds iff an initiating event has occurred, no terminating event has occurred since then, and the elapsed time is less than $Dur_{tv\text{-}acc}$ (in discrete time units).

a) Placeholder instantiation

The placeholder here captures the discrete time remaining before the value of the associated tracking variable becomes outdated. Beyond that time, the accuracy meta-fluent must be *false* as the 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 the placeholder is considered to be *0*. Hence the following instantiation:

- $PlcV = finitePlcV = [0, Dur_{tv\text{-}acc}]$; $p_0 = 0$.
- The *plcPropag* propagation function updates the value of the placeholder as follows. If the considered event is among the initiating events of the accuracy meta-fluent, the placeholder is reset; if it is among the terminating events of the fluent, the placeholder is set to 0; otherwise, the placeholder is decremented by the duration of the task associated with the event –without going beyond the lower bound. The propagation function is thus instantiated as follows:

$$plcPropag(p, e) = \begin{cases} \{Dur_{tv\text{-}acc}\} & \textbf{if } e \in Init_{tv\text{-}acc}, \\ \{0\} & \textbf{if } e \in Term_{tv\text{-}acc}, \\ p \ (-)\ Duration(e) & \textbf{otherwise}, \end{cases}$$

where the "(–)" operator removes a duration interval from the remaining time according to the following definition:

$$(-):\ \mathbb{N}^+ \times \mathcal{P}(\mathbb{N}^+) \rightarrow \mathcal{P}(\mathbb{N}^+)$$
$$T\ (-)\ [min, max] = \{x \mid x \in \mathbb{N}^+,\ T - max \le x \le T - min\}$$

b) Using decorations for time-dependent decision adequacy checking

Once the instantiated decorations are thereby computed, the following formula must be verified for each source state *source* and tracking variable *tV* appearing in the guard on an outgoing transition:

$$0 \notin Placeholders(stateDeco(source)).$$

If this formula is not satisfied, 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 outdated; the decision on which subsequent path to follow may therefore be inadequate.

c) Example

In our example of treatment for acute manic-depressive troubles, there was an implicit assumption so far that the decision of treating the patient medically does not change after an evaluation. This is of course not the case in practice. Let us assume that the decision is accurate after an evaluation and remains accurate for 10 days unless a medical treatment has been performed in the meantime (the latter corresponds to an outdate event). The definition of the tracking variable is extended as follows:

trackVar *medical_treatment_required* =
{$Evaluation_{end}$}
{$Medical\ Treatment_{end}$} **duration** *10*

The adequacy checker finds that the decoration of *state 17* in Fig. 4 contains the following mapping:

$$\{\ observation_required \wedge \ldots \mapsto \{0, \ldots\}\}.$$

This means that the clinical decision of requiring a treatment, made during the *Evaluation* task, might be inadequate. The patient observation, when required, may indeed last more than 10 days (see Table 1) which possibly invalidates the previous decision. This might for example result in patients not receiving a treatment in spite of their state aggravating during the observation.

This problem might be resolved in different ways.

- The patient state might be re-evaluated after the observation through a new *Evaluation* task to be added in the model.
- A new measure event $Observation_{end}$ might be added to the definition of the tracking variable. This would model the fact that the decision of providing a treatment is reconsidered at the end of the observation.
- The accuracy duration might be set to more than the maximum duration of the observation task –that is, over 15 days.

Interactions with medical experts are required to decide which resolution makes more sense.

VIII. TOOL SUPPORT

The various types of analysis detailed in the previous section are all supported by a toolset called GISELE. This section discusses a few design decisions and highlights key points of the implementation.

The main facilities provided by GISELE are the following.

- Editing and visualization of g-HMSC process models and their refinements.

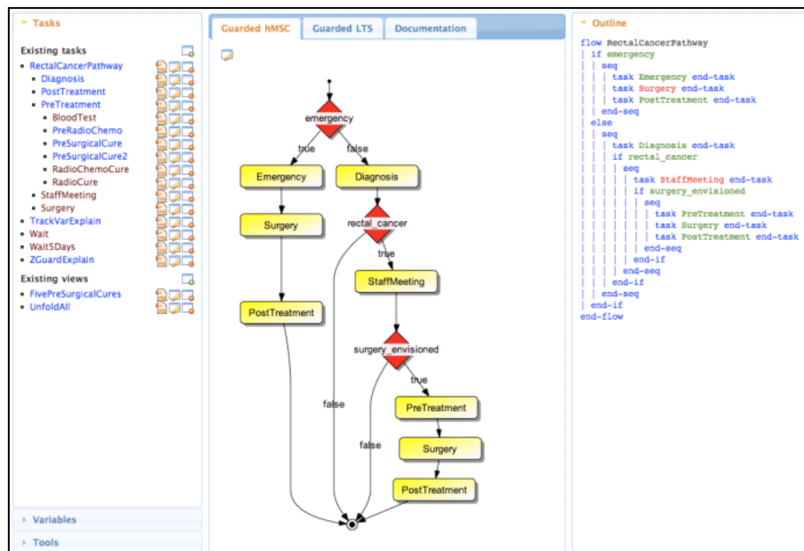

Fig. 9. Using the GISELE tool: top-level g-HMSC model for treating rectal cancer

- Visual means for eliciting process models, documenting them with process stakeholders, unfolding them for specific analyses, composing them, and projecting them on specific subclasses of instances.
- Multiple types of analyses including model checking [25] and the various analyses discussed in Section VII.

Figure 9 shows a tool screenshot. More details about the tool can be found in [46].

A. Design decisions

The toolset was designed for analysts to (a) elicit critical medical processes from interviews with experts and from available documentation of medical protocols and guidelines; (b) analyze them incrementally and on the fly; and (c) produce browsable process documentation. These objectives led to two main design decisions.

- *Textual input:* To enable rapid process capture by an analyst, a simple textual language is provided as input language to the tool (rather than a graphical one). This language amounts to a guarded-command language with constructs for sequential composition, guarded alternatives, iteration, and refinement. A graphical layout is automatically produced in full synchronization with the textual input (see Fig. 9).
- *Real-time proactive mode:* Instead of performing specific checks on demand, the tool lets the analyst navigate through the graphical model via its GUI running inside a web browser. Every time the user looks at a specific task, the tool highlights in red the problematic nodes in the model (if any). These are nodes where a check failed, revealing an inadequate decision, incomplete or overlapping guards, a pre-condition violation, a time constraint violation, and so forth. A corresponding diagnosis is produced (see Fig. 11). Our experience suggests that such incremental, non-obtrusive highlighting of problems provides natural and effective guidance towards continuous improvement during model building. Error detections stimulate discussions between stakeholders and the analyst, thereby contributing to the model elicitation process. Section IX provides some evidence on this.

B. Architecture

The toolset is implemented in Ruby using a MVC-like design pattern. Its user interface is implemented in HTML5, CSS3 and Javascript, therefore running on any recent web browser. The graphical model layout is generated server-side using the well-known *dot* utility, and presented in Scalable Vector Graphics in the browser.

Our analyses make intensive use of dedicated libraries for manipulating automata and binary decision diagrams (BDD). The automaton toolkit is the one implemented for the STAMINA contest [76] co-organized by two co-authors of this paper, see also *http://stamina.chefbe.net/*. The Buddy BDD library is available at *http://buddy.sourceforge.net*. As it is implemented in C, an interface binding for Ruby is used; it is available at *http://people.cs.aau.dk/~adavid/BDD/*.

IX. EVALUATION

This section evaluates the process analysis techniques proposed in this paper according to five criteria hierarchically organized –the idea being broadly that the evaluation criterion at one level is a prerequisite for considering the evaluation criterion at the next upper level. The criteria are the following:

- *Correctness*: are the techniques meeting their specification?
- *Performance and scalability*: are the techniques efficient enough to potentially deal with real-sized problems?
- *Applicability*: are the techniques working in the context of real-world situations?
- *Utility:* Are the techniques solving a real problem?
- *Usability*: Are the techniques accessible to process analysts? Are their results accessible to process stakeholders?

Those criteria are considered successively from bottom to top in Sections IX.A to IX.E, respectively.

A. Correctness

As discussed in Section VI, each analysis technique consists of two steps once a g-LTS is generated from the g-HMSC model:

- the decoration algorithm is instantiated in a specific way to quantities that are meaningful to the target check;
- the instantiated decorations generated by the algorithm are used in a specific way to perform the check.

The correctness of each technique thus depends on (a) the correctness of the generic decoration algorithm, (b) the correct placeholder instantiation for the considered check, and (c) the correct use of instantiated placeholders for this check. While (b) and (c) appear fairly straightforward for the various types of checks discussed in Section VII, (a) is not.

A detailed proof that the generic decoration algorithm computes the most accurate decoration *stateDeco (s)* for every g-LTS state *s* is given in [21]. The main steps of the proof are outlined here.

As discussed in Section VI.E, the most accurate decoration at state *s* must be both "*not too specific*" and "*not too general*" with regard to g-LTS executions reaching state *s*. This may be further characterized in terms of the decoration lattice defined in Section VI.C:

- the computed decoration for *s* is sufficiently high in the lattice, that is, *it covers all placeholder values produced by g-LTS executions reaching s* (Theorem 1);
- the computed decoration for *s* is not too high in the lattice, that is, *it only covers placeholder values produced by at least one g-LTS execution reaching s* (Theorem 2).

The proof of Theorem 1 relies on the following lemma.

> When the algorithm terminates, the decoration of every state *s* is higher in the lattice than the results of propagating the decoration of the predecessor states *s'* through the transition connecting *s'* to *s*.

This property is proved first for any iteration step of the algorithm and any state not in *ToExpl* at this step. Next, the property is proved for any state by noticing that *ToExpl* is empty when the algorithm terminates.

Using the preceding lemma, Theorem 1 is proved by structural induction on the length of an arbitrary g-LTS execution.

Theorem 2 is proved by computational induction on the main loop of the algorithm. The proof shows that all placeholder values added in state decorations are always values produced by at least one g-LTS execution reaching the corresponding state.

The algorithm terminates as the set *ToExpl* is eventually empty. Decorations can only go upwards in the lattice; this lattice is finite and a state can change its decoration a finite number of times only.

B. Performance and Scalability

This section discusses the theoretical complexity and practical performance of our approach.

Before a fixpoint is reached and the decoration algorithm terminates, a state can change its decoration at most H times, where H is the height of the decoration lattice. This height is exponential in the number $|VAR|$ of process variables and the size $|boundedPlcV|$ of the finite set of placeholder values. Let n denote the number of g-LTS states to be decorated; this number is linear in the number of tasks in the corresponding g-HMSC model [25]. The theoretical complexity of the decoration algorithm is thus $O(n \times 2^{|\mathrm{VAR}| \times |boundedPlcV|})$.

The size of the LTS equivalent to the g-LTS manipulated by the decoration algorithm is exponential in the number of fluents and tracking variables. The algorithm, however, does not build this LTS explicitly; it explores the same process state space symbolically.

In addition to symbolic exploration at the g-LTS level, the theoretical state blow-up problem is attenuated in various ways.

- The worst-case situation occurs when all g-LTS states are decorated with the highest lattice element. This generally corresponds to infinite executions resulting from unbounded loops. Process models are normally expected to contain finite executions only.
- As illustrated in Section IX.C hereafter, the g-HMSC refinement mechanism supports *local* and *incremental* checks. The model refinement structure effectively reduces both the size of the decorated g-LTS and the number of process variables to be considered for a specific analysis at a specific level of granularity.
- As mentioned at the end of Section VII.C, the decoration generation cost may be distributed among *multiple* types of checks. A single decoration generation, such as the computation of MAS invariants at every g-LTS state, may be exploited by multiple types of checks –in particular, for invariant generation, guard analysis, and precondition checks.
- Our implementation uses compact representations for the decoration functions *dec*, namely, binary decision diagrams (BDDs) for invariants, guards, and value assignments in ***dom*** *dec*; and intervals for placeholder values in ***img*** *dec*.

Those attenuating factors make our tool work in interactive mode quite effectively in practical situations. The user rarely waits more than a few milliseconds for a check on a typical clinical process model –for example, the model consolidated in Section IX.C hereafter has 25 tasks, 10 decision nodes, 3 fluents and 5 tracking variables; there are 4 levels of refinement and the corresponding g-LTS model has 98 states. It is worth noticing that real-sized medical process models generally have a relatively small set of states (unlike software models). A clinical pathway model for breast cancer treatment, considered as a highly complex process, includes 150 tasks to be coordinated [70].

The longest response time we experienced in our medical projects was around one second. It was observed with models erroneously containing infinite executions. For desirable unbounded loops in a process model, a timeout may be used to produce prompter feedback.

Further performance improvements might be achieved in the future for analyzing larger and/or unstructured models.

- The convergence of the decoration algorithm might be sped up through effective strategies for selecting states in the set *ToExpl* of states to be considered for further propagation (see Fig. 7). Such strategies might

significantly impact on the number of iterations required before a fixpoint is reached. Some propagations appear useless as their resulting decorations will be overridden by subsequent propagations. States having a long-lasting impact on decorations of successor states should be selected first. One effective strategy might consist of implementing *ToExpl* as a priority queue where elements are topologically sorted; a state would always be selected after the states leading to it.

- The algorithm computes the *most accurate* decoration for each state. This best mapping from assignments of process variables to instantiated placeholder values might not always be necessary for some analyses. The use of abstract interpretation [20] in such situations might significantly improve the efficiency of the instantiated algorithms by generating "good" approximations for those mappings instead (see Section X.D).

C. Applicability

This section addresses the next evaluation question after correctness and performance, namely, *are the proposed techniques working in the context of real-world situations?*

To answer this question, we show our techniques in action on the incremental building and consolidation of a clinical pathway model for treating rectal cancer.

This case study was directly inspired from various practical projects in clinical environments (see Section IX.D on *utility* hereafter) while carefully designed to deploy our multiple instantiated techniques working in combination.

The clinical pathway for rectal cancer can be described in high-level terms as follows.

> A patient generally gets in for cancer consultation (usually through a general practitioner). After this first meeting, a cancer diagnosis is established and a spread evaluation is performed. Such an evaluation is aimed at estimating parameters about cancer invasiveness and extension, namely,
>
> - ***T*** (for local Tumor invasion),
> - ***N*** (for lymphatic Node invasion),
> - ***M*** (for distant Metastasis).
>
> If rectal cancer is confirmed, the medical staff envisions some appropriate therapy strategy based on the evaluation. If the patient can undergo surgery, the main curative treatment consists in surgery. This task may be preceded or followed by chemotherapy sessions or a combination of radiotherapy and chemotherapy sessions. When the patient cannot undergo surgery, palliative care may be provided, consisting of chemotherapy sessions only or a combination of radiotherapy and chemotherapy sessions. Patients may also enter the process through an emergency service. In this case, surgery is directly prescribed.

In the various projects we were involved in, high-level descriptions of this kind were typically elicited from existing documentation of medical guidelines and from interviews with medical staff. In both cases, flowchart sketches showing tasks and decision nodes were available. These sketches were literally translated into portions of an initial g-HMSC model using our tool. The translation often stimulated fruitful discussions with medical staff. Issues about the process were raised and discussed, leading to early clarification of the preliminary, informal "model" fragments available on paper.

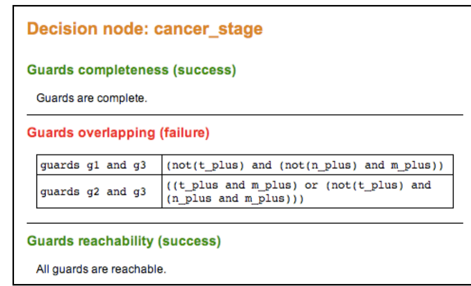

Fig. 11. Pop-up window for problematic decision node: overlapping guards

1) Top-Level g-HMSC for rectal cancer treatment

Fig. 9 shows a top-level g-HMSC model draft based on the description we elicited. In the general case, a treatment consists of a sequence of *diagnosis*, *staff meeting*, *pre-treatment*, *surgery*, and *post-treatment* tasks. The *StaffMeeting* task in Fig. 9 is a critical meeting where all medical agents involved in the process make clinical decisions about the subsequent treatment of the patient.

The tasks appearing in Fig. 9 are to be refined in other g-HMSCs. In the case where the patient comes from an emergency service with occlusion or bleeding symptoms, the patient goes directly to surgery without pre-treatment. If the cancer is not confirmed, the process is completed. If no surgery is envisioned by the medical staff, the process is also completed and the patient follows another, separate palliative care process.

The formalization of guards at decision nodes requires fluents and tracking variables to be identified and defined. Here, *rectal_cancer* and *surgery_envisioned* are tracking variables defined as follows:

trackVar $rectal_cancer = \{Diagnosis_{end}\}$,
trackVar $surgery_envisioned = \{StaffMeeting_{end}\}$.

The first definition expresses that *rectal_cancer* gets an accurate value (*true* or *false*) dependent on whether the *Diagnosis* task has revealed the presence of a cancer or not. Similarly, *surgery_envisioned* gets an accurate value (*true* or *false*) dependent on whether or not the medical staff has decided that surgery is the best plan to fight the patient's cancer.

Task preconditions on process variables might be specified at this point. In particular, the *PreTreatment*, *Surgery,* and *PostTreatment* tasks may be annotated with the following precondition:

$$rectal_cancer \wedge surgery_envisioned.$$

This means that we only proceed to these tasks if the cancer is confirmed and the surgery is envisioned for the patient. The precondition of the *StaffMeeting* task is the fluent *diag_known*, defined as follows:

fluent $diag_known = \langle\{Diagnosis_{end}\}, \{PostTreatment_{end}\}\rangle$.

This precondition expresses that the staff at the meeting should not discuss about a patient whose diagnosis is not known.

At this overall level, early checks may already be performed even though there are a few tasks only and such tasks are coarse-grained as they are not refined yet.

Checking preconditions. The precondition checker based on the decoration algorithm instantiation in Section VII.C tells us

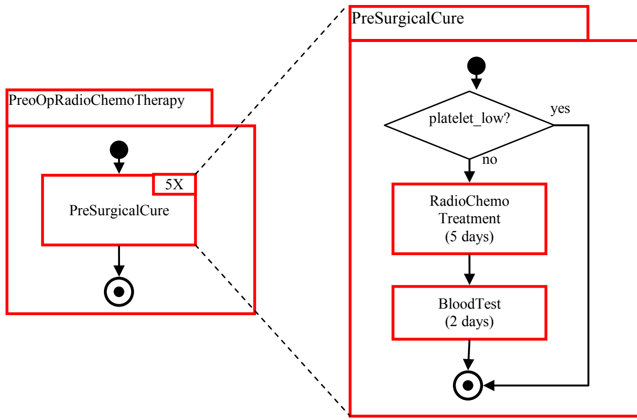

Fig. 12. Refinement of task *PreOpRadioChemoTherapy* (initial attempt)

that the preconditions of tasks *Surgery* and *PostTreatment* can be violated; if the patient comes from an emergency service, the tracking variables *rectal_cancer* and *surgery_envisioned* are not necessarily true.

To fix these problems, alternative resolution strategies may be proposed to the medical staff.

- The preconditions of *Surgery* and *PostTreatment* might be weakened as they might appear too strong.
- Two decision nodes "*rectal_cancer?*" and "*surgery_envsioned?*" might be inserted right after the *Emergency* task, requiring these two conditions to be met before the *Surgery* task.
- A distinction between two different surgeries might be made, namely, *NormalSurgery* and *EmergencySurgery*. The tasks to be performed after *EmergencySurgery* should then be elicited from medical experts.
- The treatment of rectal cancer for a patient coming from an emergency service might be too different from "normal" treatment. Two completely different processes might be envisaged for those two cases.

At this early stage, all tasks are to be refined. The next sections select some interesting ones for further refinement.

2) Refining the PreTreatment task

A refinement of the *PreTreatment* task appearing in Fig. 9 is

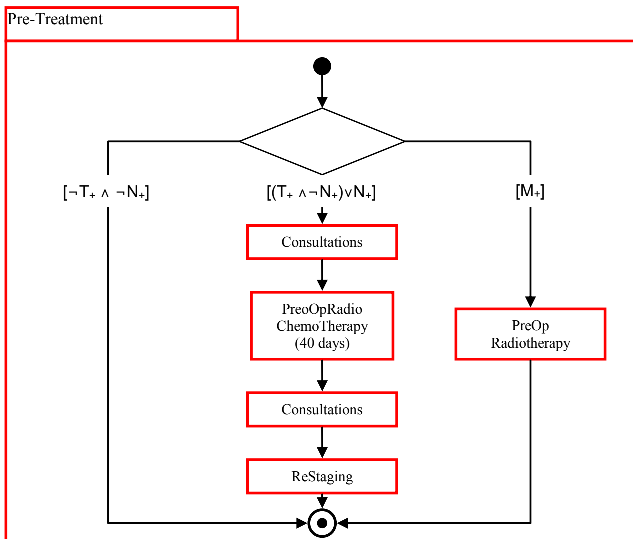

Fig. 10. g-HMSC for the *Pre-Treatment* task

shown in Fig. 10. During the *StaffMeeting* task preceding *Pre-Treatment* in Fig. 9, a specific treatment must be decided for the patient. The decision is captured by the decision node at the top of Fig. 10. The three outgoing branches and their respective guards were taken literally from a process documentation given to us by medical staff. The documented alternatives are the following.

- In case of a small tumor with no invaded lymphatic nodes, no pre-treatment needs to be envisioned.
- In case of a large tumor or in presence of invaded lymphatic nodes, the pre-treatment consists of intertwined radiotherapy and chemotherapy sessions.
- In presence of metastases, the treatment consists of radiotherapy sessions only.

Checking guards for completeness, disjointness, and satisfiability. The instantiation described in Section VII.B is used to analyze the three guards at the decision node in Fig. 10. The guards are verified to be complete and satisfiable. However, the tool opens a pop-up window explaining that they are not disjoint (see Fig. 11); the third guard overlaps the two first ones. As a consequence, the patient might be non-deterministically directed to one treatment or another. For each pair of overlapping guards, the tool gives all value assignments for the relevant process variables that satisfy them both.

The problem is easily fixed in this case by adding the conjunct $\neg M_+$ in the first two guards, as advised by medical staff.

3) Refining the task PreOpRadioChemoTherapy

The task *PreOpRadioChemoTherapy* in Fig. 10 consists of intertwined presurgical radio- and chemotherapy cures. It is refined into another g-HMSC.

The task more precisely consists of five radio-chemotherapy cures. According to available medical guidelines, the protocol requires the total radiation dose to be *exactly* 45 Grays (Gy), with 1.8 Gy per day, administered 5 days a week. For specific reasons, the duration of this task may not exceed 40 days.

Fig. 12 shows a first refinement attempt where protocol excerpts were literally translated from the medical guidelines, 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, the notation (*D* days) is used for the duration interval [*D* days, *D* days] required to perform the corresponding task. A task box with "*n*X" inside unfolds in a sequence of *n* occurrences of this task.

Fig. 12 also introduces the tracking variable *platelet_low* defined as follows:

$$\textbf{trackVar}\ platelet_low\ =\ \{BloodTest_{end}\}\ \{RadioChemoTreatment_{end}\}.$$

This definition expresses that the platelet level is accurately known after a blood test and remains accurate until the end of a subsequent *RadioChemTreatment* task. The corresponding accuracy meta-fluent is therefore:

$$\textbf{fluent}\ platelet_low\text{-}acc = \ <\{BloodTest_{end}\},\ \{RadioChemoTreatment_{end}\}>\ \textbf{initially}\ false.$$

The tool detects two problems at this stage.

Checking decision adequacy. The instantiation described in Section VII.E reveals a task-dependent decision inadequacy. The treatment decision is based on the tracking variable *platelet_low* whose value may not be accurate at the decision point –e.g., in case of first cure. The accuracy meta-fluent *platelet_low-acc* may indeed be *false* when the decision node is evaluated.
The problem may be fixed by moving the *BloodTest* task so that it appears right before the decision point (see Fig. 13a).

Checking dose requirements. A second, more subtle problem remains after rechecking the revised model in Fig. 13a. The problem is detected by a variant for dose constraints of the instantiation described in Section VII.D for time constraints. The instantiated checker finds that the total dose administered in five cures is less than the required 45 Gy in cases where the *RadioChemoTreatment* task is bypassed at least one time because of low platelet level.
This problem may be fixed if treatment cancelation is replaced by a waiting period of 5 days to allow for normalization of the platelet level. The revised model after the latter fix is shown in Fig. 13b.

Checking time requirements. If we now recheck the revised model in Fig. 13b, the tool highlights another kind of problem. The instantiation described in Section VII.D for time constraints detects that, when the treatment is delayed too often, the 40-day duration requirement can be violated.
A model-based discussion with process stakeholders suggests that a delay between radiotherapy cures is not advisable. To resolve the new problem, the radiotherapy and chemotherapy treatments might be uncoupled. Following medical advice, the possibility of delaying treatment only on first occurrence of a platelet fall is introduced, see Fig. 14. The variable *first_occurence* introduced there is a fluent defined as follows:

fluent *first_occurence* = $< \{Wait_{end}\}, \{\ \} >$ **initially** *false.*

Re-checking time requirements. When rechecked again, the revised model still violates the 40-day duration requirement; the longest possible process execution may take 42 days. The latter duration is reached for all process runs where the *Wait* task is applied for blood platelet normalization.
A discussion with oncologists suggests that the 40-day duration requirement might be a bit too unrealistic; a decision is made to weaken the requirement to 42 days.
With this weakened requirement, the model revised after three iterations meets the *time*, *dose,* and *adequacy* requirements expressed in the medical protocol. This revised version may raise further discussion about the possibility of dose reduction rather than a mere skip of chemotherapy.

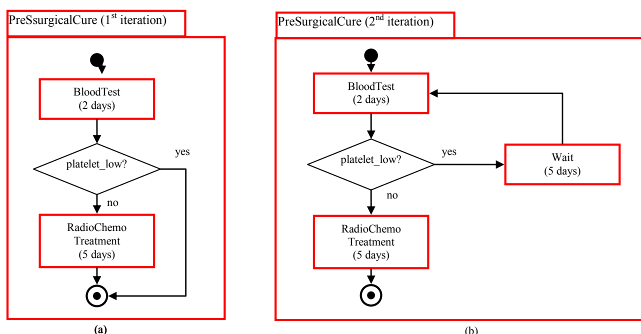

Fig. 13. Check-driven refinement of task *PreSurgicalCure* (first and second iteration)

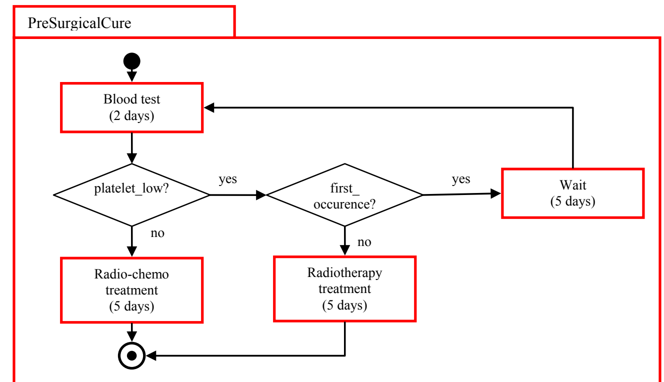

Fig. 14. Refinement of the *PreSurgicalCure* task (third iteration)

D. Utility

After correctness, performance, and applicability, this section addresses the next evaluation question, namely, *are the proposed techniques solving a real, practical problem?*
The incremental model building and analysis process in the previous section was directly inspired from our experience in multiple projects targeted at real, complex process models for cancer treatment, in particular:

- the clinical pathway for treating rectal cancer at the UC Louvain university hospital in Brussels;
- the clinical pathway for treating breast cancer at St Elisabeth hospital in Namur (Belgium);
- the workflow of the radiotherapy department at the UC Louvain university hospital in Brussels;
- the workflow of the radiotherapy department at Bordet hospital (University of Brussels).

Two different kinds of medical processes were covered [24]:

- *clinical pathways* are multi-disciplinary processes for the medical treatment of a specific class of patients presenting the same pathology;
- *department workflows* capture the various decisions and tasks across multiple pathologies in a specific hospital department.

In those various projects, each of the problems illustrated in the previous section was detected multiple times using the instantiated checkers described in Section VII.
In all projects, members of the medical staff were acting as process stakeholders. In general, they appeared highly motivated in view of the growing, widespread concern for better support towards higher-quality treatments [16, 37, 39, 44, 70]. In particular, the role of high-quality process models is being increasingly recognized for documenting, replicating, orchestrating and improving complex therapies [25, 30, 37, 43, 65, 70]. Such driving models should obviously be error-free. The multiple errors found with our formal techniques were specification and modeling errors originating from the informal documentations and flowcharts being currently used by medical staff. These errors in practice called for on-the-fly fixes at process runtime by generally overloaded medical staff.
In addition, the feedback from the tool often generated discussions with stakeholders beyond the model itself, suggesting improvements of the medical process on the field.

E. Usability

This section addresses the last evaluation question after correctness, performance, applicability, and utility, namely, *are the proposed techniques accessible to process analysts? Are their results accessible to process stakeholders?*

Let us address the second question first. The g-HMSC language was specifically designed to support analyzability by tools while remaining as close as possible to the material used by process stakeholders. The available documentation of medical guidelines, pathways and protocols [30, 43, 65] and comparative surveys involving process stakeholders [42] provide evidence that stakeholders in this domain commonly use informal flowcharts to document their procedures. Our own experience in assembling clinical process fragments supplied by medical staff confirmed this. See also *Google Images* with the keywords "care pathway flowchart" for a wide variety of examples of use of flowcharts by medical process stakeholders. Commercial tools in clinical environments are often based on flowcharts too –see, e.g., [57].

In our experience, the g-HMSC modeling language provides a good balance between (a) simplicity and communicability to medical stakeholders and (b) formality for analyzability of process specifications involving critical decisions. Restrictions were made to keep the language simple enough –at the price of limiting its expressive power, e.g., by ruling out parameterization or complex concurrency schemes and by replacing explicit assignments of arbitrary process variables by event-controlled Boolean variables.

Beyond language accessibility, the level of tool feedback in case of errors is important too for effective stakeholder involvement in process elicitation and model debugging. The roundtrip feedback of the GISELE toolset was felt essential in practice. Errors are highlighted in dedicated colors in the g-HMSC model (see upper right part of Fig. 9), and their explanation is provided at this level too (see Fig. 10 and Fig. 11) –rather than at the g-LTS level where the checks are made.

As for the first question of accessibility to process analysts, our decoration algorithm was specifically designed to be as easy to instantiate to dedicated analyses as possible. For each type of placeholder, the analyst only needs to instantiate the rule for propagating placeholders through a single event. The instantiator does not need to reason about all possible guard-satisfying paths leading to a decision nor define a lattice structure specific to the instantiation.

However, the role of process analyst in all aforementioned projects was played by ourselves. The main reason was that this role corresponds to a new, emerging type of staff profile within large clinical organizations.

X. Related work

This section compares our results with related work along four directions. Section X.A reviews process modeling languages together with techniques available for analyzing them. Section X.B focuses on process modeling languages that are specific to the medical domain. Section X.C relates our techniques with those available for the analysis of software behavior models. Finally, Section X.D compares our analysis techniques with program analysis techniques such as data flow analysis and abstract interpretation.

A. Process modeling and analysis

UML activity diagrams [62] and the Business Process Modeling Notation (BPMN) [63] are standard graphical representations for business process models. Like g-HMSCs, they have a flowchart flavor. Their specification in [62, 63], however, does not provide a precise semantics. Process descriptions based on those specifications are therefore ambiguous and cannot be verified for behavioral correctness [79]. To address this problem, various formal semantics were proposed to enable model analysis.

a) A semantics for a subset of UML 1.5 activity diagrams, inspired from the STATEMATE semantics of Statecharts [34], allows structural consistency properties to be checked on activity diagrams [28] using the NuSMV model checker [14].

b) Two semantics for a subset of BPMN are defined in the CSP process algebra [79]: an untimed one and a relative timed one. These semantics allow behavioral properties of BPMN models to be verified through model checking. In addition, interacting BPMN processes can be checked by verifying that their composition is deadlock-free [80].

c) A formal semantics for UML activity diagrams was proposed in terms of the REO coordination language [45]. It enables model-checking of activity diagrams against properties described in the μ calculus.

UML activity diagrams and BPMN are more expressive than g-HMSC. They support parallelism among sub-processes, dataflows among process agents, and transactions. On the other hand, they do not support important features enabling analyses of g-HMSC models, namely,

- *decisions based on process variables* –decisions in activity diagrams or BPMN are based on non-deterministic choice;
- *state-based properties over process variables*, in addition to event-based properties. In the work (a) and (b) hereabove, the supported properties may only refer to events associated with task performance. In (c), the properties refer to low-level concepts from the REO model (rather than concepts from the activity diagram).

Business Entity Lifecycles [17], in particular the *Guard-Stage-Milestone* (GSM) notation [38], combine data and process views for modeling business operations. In GSM, *stages* capture the lifecycle of business entities through abstract states; these states become active under specific *guard* conditions and until business *milestones* are achieved or invalidated. Guards and milestones are Boolean expressions that may refer to data attributes and event occurrences. External events are sent by the process environment whereas internal events may refer to state transitions of milestones and stages. Stages can be hierarchically refined down to low-level tasks such as service invocations. GSM supports state-based model checking of CTL properties on milestones [6, 32]. The GSM and g-HMSC approaches are similar in that both integrate event-based and state-based specifications; formal guards may refer to state variables. There are important differences, however.

- GSM models do not capture task flows explicitly. Such flows are to be inferred from responses to events in terms of guard enabling, state activation and milestone achievement. This might be more expressive. However, stakeholder involvement in the model building process appears questionable as natural task flows are implicit.
- GSM supports complex data modeling through an information model and assignments of variables. In contrast, g-HMSC restricts data modeling to fluents and tracking variables, defined in terms of events only. Lower expressiveness is the price to pay for language simplicity and separation of concerns. Tracking variables allow modellers to reason about the critical boundary between the process state and the environment state; they have no equivalent in GSM.

Fluents were originally introduced in the Event Calculus [59]. The variant used here is directly taken from [22, 31, 56]. Our fluents are slightly less expressive than those in the Event Calculus as they may not refer to other fluents. Tracking variables provide a simple means for reasoning about an *open system* where components in the process environment and their features may be unknown to the process or changing. Although fairly restricted, such variables support our target of analyzing decision adequacy. More expressive g-HMSC variables is subject to future work, see Section XI.

YAWL (Yet Another Workflow Language) is a process modeling language based on Petri nets. It was originally introduced for describing workflow patterns [74, 75]. Its expressive power is definitely higher than g-HMSCs. In particular, the modelled workflows may involve multiple instances with complex synchronization schemes, dataflows, and resource allocations. Such models, however, appear much more complex and lower-level. Guards in decision nodes are not formalized. YAWL models support different kinds of analysis based on their underlying Petri-net semantics; standard Petri-net tools are used for, e.g., verifying process termination [73] or conformance of process executions with the model [71]. As far as we know, none of the analyses discussed in Section VII are supported –even though some of them could be provided by adapting Petri-net techniques (in particular, for checking time constraints).

Little-JIL is another process language in which processes are modelled as task trees [78]. The language provides structuring constructs for composing tasks such as sequencing, parallel composition and non-deterministic choice. An exception mechanism inspired from programming languages is also available for specifying exception handling [55]. This appears quite useful in the medical domain as exceptions turn to be frequent there [33]. Little-JIL tasks may also be annotated with informal pre- and post-conditions. Decision nodes with guards are not supported. Little-JIL has a formal semantics in terms of finite state machines. Various types of analysis are therefore supported. A model checker may verify event-based properties on a process model and the absence of deadlocks [54]. Unlike our state-based properties on fluents and tracking variables, the properties there refer to events associated with task performance. A heuristic technique is also available in the Little-JIL toolset for producing fault trees from the process model in order to highlight tasks that might be wrongly executed, communications that might fail, etc. [11]. This technique somewhat corresponds to the one available for building obstacle trees against *Achieve* goals associated with g-HMSC tasks [48, 50].

Other process modeling languages were specifically designed for process enactment rather than formal analysis, notably, SPADE [3]; see [29] for a thorough review.

Dedicated temporal analysis techniques were also developed for specific languages [5, 26]. Typically, a state machine model is first derived from the input model and then checked against temporal properties. In [26], workflow models are encoded into timed automata [2] and then model-checked. No high-level language is available to process modelers that could be validated by stakeholders. Process decisions are not supported. In [5], medical guidelines are written in ASBRU [58] and then rewritten in SMV for model checking. False negatives may however appear due too coarse time abstractions. Our tool is also based on a preliminary transformation of the input model into a lower-level, machine-processable form; unlike [5, 26], the tool feedback is provided on the high-level input formalism rather than the lower-level one. Similarly to the other process formalisms mentioned before, state/event-based guards in decision nodes are not supported in [5, 26].

B. *Modeling and Analyzing Medical Processes*

Various languages dedicated to the medical domain have been proposed, notably, ASBRU [58] and Proforma [30]. These languages are generally inspired from knowledge representation techniques in artificial intelligence. They provide constructs for modeling plans composed of tasks. Few analyses are available on them. When available, such analyses are fairly syntactic or not process-specific. A thorough review of these languages may be found in [43].

LEMMA [4] appears closer to our efforts. It is a graphical language for modeling medical processes with a formal semantics in terms of Petri nets. A LEMMA model is a graph whose nodes may capture clinical tests and symptom selectors. Such nodes appear similar to our decision nodes. However, clinical tests and symptom selectors guide patients through different paths according to their current state without reference to specific process variables. This obviously reduces the expressiveness of decisions and of what can be analyzed. In particular, references to previous decisions is not possible (unlike in g-HMSC models).

Some of the process modelling languages reviewed before were also applied to the medical domain. Little-Jil, in particular, has been used to model and analyze transfusion therapies [37] and chemotherapy processes [13].

C. *Analyzing Software Behavior Models*

Our analysis techniques are also related to those available for analyzing software behavior models.

Invariant generation. The instantiation of our generic decoration algorithm in Section VII.A generalizes our previous algorithm for state invariant generation [22] by computing *most accurate* state invariants. It might be seen as a fluent-based counterpart of the algorithm for generating condition lists along scenario timelines in [52], as used in [77] for statecharts synthesis. Our algorithm, however, propagates

decorations along multiple paths to a state (rather than a single timeline); moreover, the propagation is driven by fluent definitions (rather than pre- and post-conditions).

The algorithm for generating mode invariants for SCR state machines in [41] computes one invariant per SCR mode. Like ours, it is a fixpoint one. This algorithm cannot be used for generating invariants on g-LTS states as SCR state machines are quite different. In particular, they are state-based over monitored and controlled variables, whereas g-LTS are event-based.

Consistency/completeness checking. Our checks on guards are close in spirit to those supported in [36] for SCR tables. Beyond the use of different formalisms, there is a notable difference, however. The checks here must account for the local contextual condition holding at the point in the process model where the decision node is reached. This condition is the state invariant at that point, generated by our decoration algorithm.

The technique described in [64] checks whether an RSML specification is consistent, that is, whether two outgoing transitions from the same state, triggered by the same event, have mutually exclusive guarding conditions. This technique is somewhat similar to our technique for detecting overlapping guards; it also first generates an invariant on the source of the outgoing transitions. The guards and invariants are however quite different; in RSML, they are conjunctions of predicates, each capturing whether a given state in the parallel state machine is active or not.

Model-checking. Some of the analyses in Section VII might in principle be performed through model checking [7, 14, 15, 61, 67]. As fluents and tracking variables may themselves be represented by state machines, we might model-check the parallel composition of the system extended with the state machines for each variable [25, 46]. The main benefit in comparison with decoration-based analysis would be the counterexample trace generated when the target property is not satisfied.

In particular, checking that precondition P on task T is never violated might be achieved by verifying the property

$$\Box\ (Occurs\ (T_{start}) \rightarrow P).$$

Checking decision adequacy might be achieved by verifying a property of form

$$\Box\ (Cond \rightarrow tV\text{-}acc),$$

where *Cond* is a condition to be determined so as to be *true* right at the point where the corresponding decision node on tracking variable *tV* is evaluated, and *false* elsewhere; and where *tV-acc* is the accuracy meta-fluent associated with *tV*.

Checks for guard completeness and disjointness are not straightforward through model checking; the target properties appear difficult to translate in terms of temporal logic formulas. In case of guard overlap, our decoration-based tool returns all value assignments meeting multiple guards. Such information would not be provided by the model checker. Moreover, our time-related analyses cannot be translated to fluent-based LTL properties supported by our model checker. Last but not least, the decoration-based analyses are performed at a more abstract level that the LTS level, resulting in easier roundtrip feedback to the input g-HMSC model and more efficient analysis due to symbolic exploration at the g-LTS level.

Some analyses could be translated to model-checking problems for NuSMV [14] (e.g., decision adequacy checks) or Uppaal [53] (e.g., time constraint checks). The integration of multiple model checkers relying on different semantics within a coherent and consistent framework appears, however, more difficult than the simple, uniform decoration-based approach presented here. Moreover, most checks appear fairly straightforward once decorations are generated; the decoration algorithm handles the intricacies of propagating possible values of process variables along all guard-satisfying paths leading to the considered state. The cost of generating decorations may thereby be distributed among multiple checks; a single decoration generation may be used for multiple analyses. In this uniform and reusable framework, new dedicated analyses may be added in a quite easy and semantically consistent way, as shown by the radiation dose check in Section VII.C.

Timing analysis is commonly recognized to be complex and computationally expensive in state-of-the-art verification technology [7]. Our instantiated techniques involving time are less exposed to such problems thanks to the restrictions naturally arising from our context –namely, the exclusiveness of tasks ensured by the guards and the representation of time through discrete sets of time points.

D. Program Analysis

Our techniques can also be related to those used for compile-time verification of program properties. Data flow analysis [35, 61] and abstract interpretation [18, 19, 20, 61] are complementary approaches for program analysis. Data flow analysis computes relevant information about the possible set of values of program variables at every node of the program's control flow graph. Such information is generally obtained by setting up data flow equations at each node of the graph. These equations are solved by repeated calculations of outputs from inputs until a fixpoint is reached. To guarantee termination, constraints are imposed on value domains –typically, a lattice structure or partial order with finite height. To improve efficiency while preserving correctness, data flow analyses are in general combined with abstract interpretation –see, e.g., [8, 61]. Abstract interpretation symbolically executes the program at some level of abstraction where irrelevant details about the semantics and the specification are ignored; *abstract* values are used instead of *concrete* ones.

Like data flow analysis and abstract interpretation, our analysis techniques compute, for every g-LTS node, relevant information about the possible values of process variables. In the presence of process loops, the computation of meaningful quantities similarly proceeds by "climbing" a lattice until a fixpoint is reached. Such fixpoint calculation is also found in other invariant generation techniques [41, 51].

Unlike abstract interpretation techniques, however, our techniques compute *exact* values for placeholders, not approximated ones. Approximations may be avoided for several reasons. First, our process language is higher-level than a programming language, with much simpler constructs such as Boolean variables and guarded branching over them,

event-based transitions, sequential tasks, and time representation through discrete time points. Moreover, some constructs are already built-in abstractions; in particular, Boolean tracking variables capture predicate abstractions over environment quantities.

Abstract interpretation techniques might however be worth considering in future work.

- They might be needed for enriching the process language with more expressive and complex constructs such as, e.g., guards with comparison operators over integer variables.
- The *most accurate* mapping might not always be required for certain analyses. Full accuracy might be sacrified to efficiency. As an example, let us consider a process for which we would like to know whether all executions last less than a specific duration *d*. Computing the exact mapping between value assignments of process variables and sets of time points, as in Section VII.D, might result in poor performance on very large processes with many variables. A similar algorithm might then be used without taking assignments of fluents and tracking variables into account; guarded branches would be selected non-deterministically. With such over-generalization, the target property would be verified for all patients in case the maximum time computed by the algorithm is less than *d*. However, getting a greater value might yield a false positive as the trace reaching the maximum time might not be a valid one (see the guard satisfaction condition on the g-LTS semantics in Section V.B).

Unlike our techniques, abstract interpretation requires dedicated proofs for every specific abstraction. In particular, the abstract semantics must each time be proved correct with respect to the concrete one.

XI. Conclusion

Model-driven engineering requires high-quality models of artefacts or processes in mission-critical domains. The models must be correct for their safe use and readable by stakeholders for their elicitation and validation. Model elaboration may be complex and error-prone.

The paper presented a variety of tool-supported analysis techniques for building more adequate, complete, and consistent process models in which explicit decisions regulate task flows. Such decisions generally depend on the process state and on the state of the environment in which the process operates.

The analysis techniques described in the paper may be applied incrementally and locally to partial models at various levels of refinement. The models may thereby be elaborated, verified, corrected, and refined through successive iterations. The approach thereby reduces the difficulties and cost of late fixing of errors disseminated through a large, complex model.

More specifically, the paper makes the following contributions.

- The formalism for modeling decision-based processes is close to the informal sketches provided by process stakeholders while introducing process variables for higher precision of decision nodes. The language has a formal operational semantics expressed in a lower-level formalism to enable various types of automated analysis. The high-level user language combines event-based and state-based specifications to extend the class of properties that can be checked.
- A generic algorithm computes state decorations by propagations through the lower-level model. Instantiations of this algorithm yield various types of analyses. The decorations map value assignments for the process variables to generic placeholders so as to account for the various guard-satisfying paths leading to the corresponding state.
- A variety of complementary techniques allow decision-based process models to be analyzed formally. The techniques may be used for generating state invariants, checking preconditititions, analyzing guards at decision nodes, checking the adequacy of decisions, and verifying non-functional process requirements involving time and resources. These techniques are obtained through different instantiations of the same generic decoration algorithm. The instantiations require minimal effort; the analyst just needs to define the set of placeholder values and provide the function specifying how instantiated placeholder values are to be propagated through a single state transition.
- A roundtrip tool implements those techniques to proactively and non-obtrusively highlight the problems detected during model elaboration.

The paper provides an evaluation of the techniques in terms of correctness, performance, applicability on a complex cancer treatment process, utility in the medical domain, and usability. Compared with process notations such as BPMN [63] or YAWL [75], the g-HMSC language appears intuitive and simple enough to involve process stakeholders in the model elicitation/validation loop:

- the analyses are performed on models that are close to the material provided by them;
- the flaws are highlighted by the tool on this high-level model (rather than on the lower-level one it manipulates); they can therefore be more easily fixed with stakeholders –at least in our experience with medical staff involved in complex cancer therapies.

The techniques presented in the paper raise various issues for further work.

On the *language* side, the price to pay for simplicity is limited expressiveness.

- Our focus on decision-based processes led us to leave aside language constructs for concurrency among sub-processes and exception handling [33, 55]. While not necessarily needed for medical processes such as clinical pathways, such constructs are required for multi-instance processes, their enactment and orchestration. The integration of a suitable exception handling mechanism might also simplify decisions and their corresponding guards in specific situations.
- The process models should also be extended with goals

underlying tasks. Goals provide the rationale for tasks; moreover, they prove to be more stable than the tasks operationalizing them [48]. In our experience, some critical tasks appear difficult to refine into subtasks (e.g., the *StaffMeeting* task in Fig. 10). A goal refinement tree may then appear much more appropriate. The integration of goals would also enable complementary analyses [48]. Goal refinements may be checked for completeness; incomplete refinements lead to the identification of new goals and therefore new tasks operationalizing them. Missing tasks or missing paths could thereby be found. Obstacle analysis might further improve the model by generating unexpected risk conditions and exploring corresponding resolutions through countermeasure goals and tasks [1, 50].

- The g-HMSC process variables are currently limited to propositional variables; more complex domains should be supported.
- The mechanism for refining tasks might be improved through multiple time granularities and macro-events [10] in order to remove the need for action events having durations.

New constructs require extending the language, its semantics, and the analyses accordingly. A good tradeoff should however be kept between expressiveness, analyzability, and usability by process stakeholders.

On the *analysis* side, the feedback provided by the tool should be improved. Currently, the tool highlights error states in the process model and provides a corresponding state invariant. As additional feedback, a counterexample trace should be generated to help understanding the root causes of the problem.

Domain properties [48] should also be integrated in the model in order to simplify generated decorations such as preconditions or invariants. By removing redundant information known as domain properties the resulting decorations would sometimes be more compact and more understandable.

As discussed in Section IX.D, some instantiations of the generic decoration algorithm might raise efficiency concerns in the process state space exploration. Abstract interpretation techniques might help increase efficiency at the price of introducing approximations.

The integration of multiple interfering process models is another challenging issue. For example, a patient following both diabetes and colorectal cancer therapies is exposed to "feature interaction" problems: a medication contributing positively to the patient's state along one process might contribute negatively to that state along the other process. Goals might prove useful for detecting conflicts among the processes operationalizing them [49]. We might also use the model checker described in [25] to detect interferences in the parallel composition of the process models. There are different stages for detecting and resolving conflicts –at modeling time or at runtime during process enactment. This should be further investigated in order to know which approach would work best.

New instantiations of our generic decoration algorithm should be considered as well –for verifying other non-functional process requirements or for other uses in the model building process. For example, other instantiations recently allowed model engineering operators to be defined, including the *union* of paths from multiple models, the *restriction* of a model to specific paths, the *projection* of a model on a specific set of tasks, and the *merge* of concurrent models [24].

ACKNOWLEDGMENT

Warmest thanks are due to our medical colleagues François Roucoux, Yves Humblet and Pierre Scalliet, and to the medical staff at the Chemotherapy and Radiotherapy units of the UCL Saint-Luc University Hospital for their time and experience sharing. Thanks also to Baudouin Le Charlier for helping us clarify the relationship and difference between abstract interpretation and our approach. We are grateful to George Avrunin and the reviewers for useful comments and clarification questions on earlier versions of this paper. The layered organization of our evaluation in Section IX arose from discussions with Emmanuel Letier.

The work reported herein was partially supported by the Regional Government of Wallonia (GISELE project nr. 616425 and PIPAS project nr. 1017087).

REFERENCES

[1] D. Alrajeh, J. Kramer, A. van Lamsweerde, A. Russo and S. Uchitel, "Generating obstacle conditions for requirements completeness", *Proc. ICSE'2012: 34th Intl. Conf. on Software Engineering*, Zürich, ACM-IEEE, pp. 705-715, 2012.

[2] R. Alur and D. L. Dill, "A Theory of Timed Automata", *Theoretical Computer Science*, Vol. 126 No. 2, pp. 183-235, 1994.

[3] S. Bandinelli, A. Fuggetta, C. Ghezzi and L. Lavazza, "SPADE: an environment for software process analysis, design, and enactment". In *Software Process Modelling and Technology*, A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), Research Studies Press Ltd., Taunton, UK, pp. 223-247, 1994.

[4] L. Baresi, F. Consorti, M. Di Paola, A. Gargiulo, and M. Pezzè, "LEMMA: a language for easy medical model analysis", *Journal of Medical Systems*, Vol. 21 No 6, pp. 369-388, 1997.

[5] S. Bäumler, M. Balser, A. Dunets, W. Reif, J. Schmitt, "Verification of medical guidelines by model checking: a case study", *Proc. SPIN 2006*, pp. 219-233, 2006.

[6] F. Belardinelli, A. Lomuscio, and F. Patrizi, "Verification of GSM-based artifact-centric systems through finite abstraction", *Service-Oriented Computing*, Springer-Verlag, pp. 17-31, 2012.

[7] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen, *Systems and Software Verification – Model Checking Techniques and Tools*. Springer-Verlag, 2001.

[8] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: concretizing the convergence of model checking and program analysis", *Proc. CAV'2007: Intl. Conf. on Computer-Aided Verification*, LNCS 4590, Springer-Verlag, pp. 504-518, 2007.

[9] N. Bjorner, A. Browne and Z. Manna, "Automatic generation of invariants and intermediate assertions", *Theoretical Computer Science* Vol. 173 No. 1, pp. 49-87.

[10] I. Cervesato and A. Montanari. "A calculus of macro-events: progress report", *Proc. TIME'2000: 7th IEEE Intl. Workshop on Temporal Representation and Reasoning*, pp. 47-58, 2000.

[11] B. Chen, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, "Automatic fault tree derivation from Little-JIL process definitions", *Proc. SPW 2006: Software Process Workshop*, Shanghai, LNCS 3966, Springer-Verlag, pp. 150-158, 2006.

[12] B. Chen, G. S. Avrunin, E. A. Henneman, L. A. Clarke, L. J. Osterweil, P. L. Henneman, "Analyzing medical processes", *Proc. ICSE'2008: 30th Intl. Conf. on Software Engineering*, ACM-IEEE, pp. 623-632, 2008.

[13] S. Christov, B. Chen, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, D. Brown, L. Cassells, and W. Mertens, "Rigorously defining and analyzing medical processes: an experience report". In H. Giese (Ed.): *MODELS 2007 Workshops*, LNCS 5002, Springer-Verlag, pp. 118-131, 2008.

[14] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: an opensource tool for symbolic model checking", *Proc. CAV'2002: Intl. Conf. on Computer-Aided Verification*, LNCS 2404, Springer-Verlag, pp. 359-364, 2002.

[15] E.M. Clarke and E.A. Emerson, "Automatic verification of finite-State concurrent systems using temporal logic specifications", *ACM Trans. Program. Lang. Systems* Vol. 8 No. 2, pp. 244-263, 1986.

[16] M. R. Cohen, "Causes of medication errors", *Medication Errors*, M.R. Cohen (Ed.), Jones and Bartlett, Toronto, pp. 55-66, 1999.

[17] D. Cohn and R. Hull, "Business Artifacts: A Data-centric approach to modeling business operations and processes", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, Vol. 32 No. 3, pp. 3-9, 2009.

[18] P. Cousot, "Abstract interpretation", *ACM Computing Surveys* Vol. 28, No 2, pp. 324-328, 1996.

[19] P. Cousot, "Abstract interpretation based formal methods and future challenges", *Informatics*, Springer-Verlag, pp. 138-156, 2001.

[20] 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*, pp. 238-252, 1977.

[21] C. Damas, *Analyzing Multi-View Models of Software Systems*. Ph.D. Thesis, Université catholique de Louvain, 2011.

[22] 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, pp. 1056-1073, 2005.

[23] C. Damas, B. Lambeau and A. van Lamsweerde, "Scenarios, goals, and state machines: a win-win partnership for model synthesis", *Proc. FSE'06, 14th ACM International Symp. on the Foundations of Software Engineering*, Portland (OR), pp. 197-207, 2006.

[24] C. Damas, B. Lambeau, and A. van Lamsweerde, "Transformation operators for easier engineering of medical process models", *Proc. SEHC'2013: 5th ICSE Workshop on Software Engineering in Health Care*, San Francisco, ACM-IEEE, 2013.

[25] C. Damas, B. Lambeau, F. Roucoux, and A. van Lamsweerde, "Analyzing critical process models through behavior model synthesis", *Proc. ICSE'2009: 31st Intl. Conf. on Software Engineering*, ACM-IEEE, pp. 441-451, 2009.

[26] E. De Maria, A. Montanari, and M. Zantoni, "An automaton-based approach to the verification of timed workflow schemas", *Proc. TIME 2006*, pp. 87-94, 2006.

[27] M. Dumas, W. van der Aalst, A. ter Hofstede, *Process-Aware Information Systems*. Wiley, 2005.

[28] R. Eshuis, "Symbolic model checking of UML activity diagrams", *ACM Trans. on Software Eng. and Methodology (TOSEM)* Vol. 15, No. 1, pp. 1-38, 2006.

[29] A. Finkelstein, J. Kramer and B. Nuseibeh (Eds.), *Software Process Modelling and Technology*. John Wiley & Sons, 1994.

[30] J. Fox, N. Johns, A. Rahmanzadeh, "Disseminating medical knowledge: the ProForma approach", *Artif. Intell. Med.* Vol. 14, pp. 157-181, 1998.

[31] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems", *Proc. ESEC/FSE'2003*: *European Software Engineering Conference and ACM International Symp. on the Foundations of Software Engineering*, Helsinki, pp. 257-266, 2003.

[32] P. Gonzalez, A. Griesmayer, and A. Lomuscio, "Verifying GSM-based business artifacts", *Proc. 19th Intl. Conference on. Web Services (ICWS),* IEEE, pp. 25-32, 2012.

[33] M. Han, T. Thiery, X. Song, "Managing exceptions in medical workflow systems", *Proc. ICSE'06, 28th Intl. Conf. on Software Engineering*, Shanghai, ACM-IEEE, pp. 741-750, 2006.

[34] D. Harel and A. Naamad, "The STATEMATE semantics of Statecharts", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 5 No. 4, pp. 293-333, 1996.

[35] M. S. Hecht, *Flow analysis of computer programs*. Elsevier Science, 1977.

[36] C.L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications", *ACM Trans. on Software Eng. and Methodology (TOSEM)* Vol. 5 No. 3, pp. 231-261, 1996.

[37] E. Henneman, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, C. Andrzejewski, K. Merrigan, R. Cobleigh, K. Frederick, E. Katz-Bassett, and P. Henneman, "Increasing patient safety and efficiency in transfusion therapy using formal process definitions", *Transfusion Medicine Reviews*, Vol 21 No 1, pp. 49-57, 2007.

[38] R. Hull, E. Damaggio, F. Fournier, M. Gupta, F. T. Heath III, S. Hobson, M. Linehan, S. Maradugu, A. Nigam, S. Sukaviriya, and R. Vaculin, "Introducing the Guard-Stage-Milestone approach for specifying business entity lifecycles", *Proc. Intl. Workshop on Web Services and Formal Methods,* 2010.

[39] Institute of Medicine, "A New Health System for the 21st Century". In *Crossing the Quality Chasm*, National Academy Press, Washington (D.C.), pp. 23-38, 2001.

[40] ITU, *Message Sequence Charts*. Recommendation Z.120, Intl. Telecom Union, Telecom. Standardization Sector, 1996.

[41] R. Jeffords and C. Heitmeyer, "Automatic generation of state invariants from requirements specifications", *Proc. FSE'1998: 6th ACM Symp. Foundations of Software Engineering*, Los Alamitos (CA), pp. 56-69, 1998.

[42] G.T. Jun, J.R. Ward and Z. Morris, "Health care process modelling: which method when?", *International Journal for Quality in Health Care*, Vol. 21 No.3, pp. 214-224, 2009.

[43] S. Kaiser and S. Miksch, *Modeling Computer-Supported Clinical Guidelines and Protocols: A Survey.* Vienna Univ. Technology, Report Asgaard-TR-2005-2, 2005.

[44] L.T. Kohn, J.M. Corrigan, M.S. Donaldson (eds.), *To Err is Human: Building a Safer Health System*. National Academy Press, Washington (D.C.), 1999.

[45] N. Kokash C. Krause and E. de Vink, "Reo+ mCRL2: A framework for model-checking dataflow in service compositions", *Formal Aspects of Computing* Vol. 24 No.2, pp. 187-216, 2012.

[46] B. Lambeau, *Synthesizing Multi-Model Views of Software Systems*. Ph.D. Thesis, Université catholique de Louvain, 2011.

[47] A. van Lamsweerde, "Formal specification: a roadmap". In *The Future of Software Engineering,* A. Finkelstein (Ed.), ACM Press, pp. 147-159, 2000.

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

[49] A. van Lamsweerde, R. Darimont, E. Letier, "Managing conflicts in goal-driven requirements engineering", *IEEE Transactions on Software Engineering* Vol. 24 No. 11, pp. 908-926, 1998.

[50] A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering", *IEEE Transactions on Software Engineering* Vol. 26 No. 10, pp. 978-1005, 2000.

[51] A. van Lamsweerde and M. Sintzoff, "Formal derivation of strongly correct concurrent programs", *Acta Informatica* Vol.12 No. 1, Springer-Verlag, pp. 1-31, 1979.

[52] A. van Lamsweerde and L. Willemet, "Inferring declarative requirements specifications from operational scenarios", *IEEE Trans. on Software. Engineering* Vol. 24 No. 12, pp. 1089-114, 1998.

[53] 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, pp. 134-152, 1997.

[54] B. S. Lerner, "Verifying process models built using parameterised state machines", *Proc. ISSTA'2004: ACM SIGSOFT Symp. Software Testing and Analysis*, pp. 274–284, 2004.

[55] B. S. Lerner, S. Christov, L.J. Osterweil, R. Bendraou, U. Kannengiesser, and A. Wise, "Exception handling patterns for process modeling", *IEEE Transactions on Software Engineering* Vol. 36 No. 2, pp. 162-183, 2010.

[56] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*. Second Edition, John Wiley & Sons, 2006.

[57] Mapofmedecine, www.mapofmedicine.com/solution/whatisthemap/, 2013.

[58] S. Miksch, Y. Shahar, and P. Johnson. "Asbru: A task-specific, intention-based and time-oriented language for representing skeletal plans", *Proc. KEML'97: 7th Workshop on Knowledge Engineering - Methods & Languages,* pp 9-19, 1997.

[59] R. Miller, M. Shanahan, "The Event Calculus in classical logic – alternative axiomatisations", *Linkoping Electronic Articles in Computer and Information Science* Vol. 4 No. 16, pp. 1-27, 1999.

[60] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[61] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer-Verlag, 2004.

[62] OMG, *UML 2.0 Superstructure Specification*, 2003.

[63] OMG, *Business Process Modeling Notation*, v1.1, 2008.

[64] D. Park, J. Skakkebæk, and D. Dill, "Static analysis to identify invariants in RSML specifications" *Proc. FTRTFT'98: 5th Intl. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 1486, Springer-Verlag, pp. 133-142, 1998.

[65] M. Peleg, S. Tu, J. Bury, P. Ciccarese, J. Fox, R. A. Greenes, R. Hall, P. Johnson, N. Jones, A. Kumar, S. Miksch, S. Quaglini, A. Seyfang, E. Shortliffe, and M. Stefanelli, "Comparing computer-interpretable guideline models: a case-study approach", *Journal of the American Medical Informatics Association* Vol.10 No.1, pp. 52-68, 2003.

[66] M. Pradhan, M. Edmonds, and W. Runciman, "Sequence diagrams for visualizing healthcare processes". In *Quality in Healthcare : Process, Best Practice & Research Clinical Anaesthesiology* Vol. 15 No. 4, pp. 555-571, 2001.

[67] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CAESAR", *Proc. 5th Intl. Symp. on Programming*, LNCS 137, Springer-Verlag, pp. 337-351, 1982.

[68] A. Rae, D. Jackson, P. Ramanan, J. Flanz, and D. Leyman, "Critical feature analysis of a radiotherapy machine", *Reliability Engineering & System Safety* Vol. 89 No. 1, pp. 48-56, 2005.

[69] M.S. Raunak and L.J. Osterweil, "Resource management for complex, dynamic environments", *IEEE Transactions on Software Engineering* Vol. 39 No. 3, 2013.

[70] F. Roucoux, R. Florquin, C. Quintens and V. Remouchamps, "Applying a computerized care pathway orchestration system in the hospital", *Proc. ECPC'2013: European Care Pathways Conference*, Glasgow, June 2013.

[71] A. Rozinat and W. M. P. van der Aalst, "Conformance checking of processes based on monitoring real behavior", *Information Systems* Vol. 33 No. 1, pp. 64-95, 2008.

[72] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios", *IEEE Trans. Softw. Engineering* Vol. 29 No. 2, pp. 99-115, 2003.

[73] W. M. P. van der Aalst, "Verification of workflow nets", *Proc. Application and Theory of Petri Nets 1997,* LNCS 1248, Springer-Verlag, pp. 407-426, 1997.

[74] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns", *Distributed and Parallel Databases* Vol. 14 No.1, pp. 5-51, 2003.

[75] W. M. P. van der Aalst and A. H. M. ter Hofstede, "YAWL: yet another workflow language", *Information Systems* Vol. 30 No. 4, pp. 245-275, 2005.

[76] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, P. Dupont, "STAMINA: a competition to encourage the development and assessment of software model inference techniques", *Empirical Software Engineering* Vol. 18 No.4, pp. 791-824, 2013.

[77] J. Whittle and J. Schumann, "Generating statechart designs from scenarios", *Proc. ICSE'2000: 22nd Intl. Conference on Software Engineering*, Limerick, ACM-IEEE, pp. 314-323, 2000.

[78] A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton, "Using Little-JIL to coordinate agents in software engineering", *Proc. ASE'2000: Automated Software Engineering Conference*, Grenoble, IEEE, pp. 155-163, 2000.

[79] P. Y. H. Wong and J. Gibbons, "Formalisations and applications of BPMN", *Science of Computer Programming* Vol. 76 No. 8, pp. 633-650, 2011.

[80] P. Y. H. Wong and J. Gibbons, "Property specifications for workflow modelling', *Science of Computer Programming* Vol. 76 No.10, pp. 942-967, 2011.