ANALYZING MULTI-VIEW MODELS OF SOFTWARE SYSTEMS

Christophe Damas

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Engineering Sciences

November 30, 2011

Computing Science Engineering
ICTEAM
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Examining Board:

Prof. Axel van LAMSWEERDE (UCL), Supervisor

Prof. Olivier BONAVENTURE (UCL), Chairperson

Prof. Charles PECHEUR (UCL), Secretary

Prof. Emmanuel LETIER (Univ. College London, UK)

Prof. Baudouin LE CHARLIER (UCL)

Prof. Jeff KRAMER (Imperial College London, UK)

Abstract

The model-driven elaboration, validation, and documentation of system requirements and designs calls for rich system models. To be comprehensive, such models need to cover the intentional, structural, operational and behavioral dimensions of the target system. Building adequate, complete, and consistent models is not an easy task. Techniques should therefore be available for detecting and fixing modeling errors early and in a systematic way. The thesis provides an integrated set of tool-supported techniques for analyzing system models along the behavioral, operational and intentional dimensions.

Our analysis techniques mainly focus on *decision-based* process models. The latter capture processes where the application of specific tasks and their sequencing depends on explicit decisions that are based on the state of the environment in which the process operates. Decision-based processes in areas such as healthcare are often critical in terms of timing and resources. The language of High-Level Message Sequence Charts is extended with guards, timing and resource constructs to model such processes. This language allows us to naturally integrate stakeholder material including multi-agent scenarios, decision trees and flowchart fragments.

The proposed analysis techniques allow us to highlight a variety of modeling problems early and incrementally on partial models, including: inadequate decisions resulting from inaccurate or outdated information about the environment; incompleteness of decisions; unreachability of specific tasks along certain process paths; undesirable non-determinism in process decisions; and violations of temporal constraints. These techniques are demonstrated on the incremental building and analysis of a complex model of a real protocol for cancer therapy.

Other techniques are proposed for analyzing interaction scenarios, specified as Message Sequence Charts, and state machines, specified as Labeled Transition Systems. The latter techniques allow us to infer a significant class of goals and domain properties from scenarios and to generate state invariants on each node of a state machine behavior model.

Many of the analysis techniques proposed are based on different instantiations of a generic algorithm that propagates decorations through a model graph until a fixpoint is reached.

Acknowledgments

Foremost, I am grateful to Axel van Lamsweerde, my supervisor, for his guidance, and encouragement. This work would not have been possible without his human, technical and financial support.

I thank the members of the jury, Charles Pecheur, Emmanuel Letier, Baudouin Le Charlier, Jeff Kramer and Olivier Bonaventure for their useful comments on earlier versions of this thesis. In particular, I am grateful to Baudouin Le Charlier who introduced me to the topic of Abstract Interpretation. It helped me elaborating the different proofs of the thesis.

Many thanks go to Bernard Lambeau, with whom I am working for the past eight years. The main results of this thesis have been done in collaboration with him. Working with him is a great pleasure.

I benefited from many discussions with colleagues of the KAOS group, Renaud De Landtsheer, Emmanuel Letier, Antoine Cailliau, Christophe Ponsard and Robert Darimont.

I thank Dr. Francois Roucoux for sharing his medical knowledge. He gave me precious comments about the case-study of this thesis. I also thank Dr. Yves Humblet and the medical staff at the Chemotherapy and Radiotherapy units of Saint-Luc hospital for their time and experience sharing.

I am grateful to the friends and colleagues at INGI who have made working on this thesis more enjoyable, sharing lunches, discussions and advices. In particular, I thank Jose, Jean-Noel and the members of the INGI running team, Sebastián “Speedy”, Boriss and Pierre.

I also thank my parents, sister and brothers, nephews and friends.

This research was supported by the Regional Government of Wallonia (ReQuest project, RW Conv. 315592 and GISELE project, RW Conv. 616425).

Contents

List of Figures

Chapter 1 Introduction

Models are increasingly recognized as an effective means for elaborating requirements and exploring designs. A *model* is an abstract representation of the target system, where key features are highlighted, specified and inter-related to each other [Lam09]. Models provide multiple benefits:

- They force stakeholders and analysts to be more precise in their system description.
- They allow them to abstract from multiple details in order to focus on key system aspects.
- They provide a basis for early detection and fixing of errors.

Models are widely used at different steps of the software development process.

Software designs may be captured by Model Driven Engineering (MDE) technologies such as UML models [OMG03]. Such methodologies focus on modeling the application domain rather than the computing or algorithmic aspects. The code of the application can be then written based on these models. Sometimes, code fragments can even be generated from such models.

Software requirements may emerge from KAOS [Lam09] or NFR/i* [Myl92, Yu93] models. Such models explain how system objectives may be refined into lower-level goals. KAOS puts more emphasis more on semi-formal and formal reasoning about behavioral goals, i.e. goals that prescribe system behaviors declaratively. In NFR/i*, the emphasis is more on qualitative reasoning on soft goals, i.e. goals that cannot be established in a clear-cut sense.

Work processes may be represented by graphical models. Such models may take various forms: flowchart style, e.g. Activity Diagrams [OMG03],

BPMN [OMG08] or task trees, e.g. Little-Jil [Cla08]. Such models may be just graphical representations, e.g. Activity Diagrams or BPMN, or have a strong formal semantics, e.g. Little-Jil or YAWL [Van05].

To play a significant role, a good model should meet the following requirements [Lam09]:

- *Adequate*: the model should adequately represent the essence of the target system while abstracting unnecessary details;
- *Complete*: the model should capture all pertinent facets of the system [Nis89, Rum91, Nus94];
- *Multilevel*: the model should capture the system at different levels of abstraction and precision to enable stepwise elaboration and validation;
- *Comprehensible*: the model should be easy to understand by the people who need to use them;
- *Precise* and *analyzable*: the model should be accurate enough to support useful forms of analysis.

A formal model supports the latter requirement. The semantics of a modeling language provides precise rules of interpretation that allow many of the problems with natural language to be overcome. Formal specifications may be manipulated by automated tools for a wide variety of purposes. Analyses may be done at surface level, e.g. by query-based checks on model [Lam09], or more deeply according to different types of approaches:

- Model-checking of behavior models [Cla86, Que82]. A model-checker verifies if a behavior model satisfies a specification and if not, generates a counterexample. Model-checking can be done on state-based behavior models, e.g. [McM93], or event-based ones, e.g. [Mag06].

- Abstract interpretation [Cou77]. Abstract interpretation aims at discovering and proving properties of a model or program by sound approximations of their behaviors.
- Model synthesis. Some models may be synthesized automatically from other models. For example, behavior models or goals may be synthesized from scenarios [Lam99, Whi00, Mak01, Uch03, Lam11].

Such formal techniques may have limitations.

- They are in general not available for early model analysis, when the models are still partial or when they are only described at a high-level of granularity. Early detection of errors avoids the difficulties of late fixing of anomalies disseminated at different places of a large, complex model. In addition, early analysis contributes to the model elicitation process.
- They are in general not available for high-level models that are understandable by stakeholders.
- In particular, the techniques available for analyzing process models are fairly operational and applicable to a restricted class of models. For example, formal decision nodes are not supported; the latter add much complexity in terms of covered traces. Moreover, a lot of interesting analyses can be performed on decision nodes. These limitations are particularly harmful in safety-critical areas such as medical processes that often involve complex decision nodes.

The thesis aims at providing an integrated set of tool-supported techniques for early analysis of high-level system models along the operational, behavioral and intentional dimensions.

- The *intentional dimension* captures the system objectives to be satisfied by cooperation of the agents forming the system being considered.
- The *operational dimension* captures the different tasks and decisions to be applied in order to operationalize some objective.
- The *behavioral dimension* captures the admissible behaviors of the system in terms of sequences of events.

Analyses along the behavioral and intentional dimension

Annotating state machine models with state invariants provides multiple benefits:

- the understandability and documentation of the state machine is improved;
- the invariants can be used for validation and error detection;
- code generators may use them for improving code quality;
- they can be used by analysis tools for more efficient analysis.

The thesis presents algorithms for generating state invariants on nodes of an event-based state machine specified by a Labeled Transition Systems (LTS). The computed state invariants are Boolean expressions on fluents. The latter are state predicates whose truth values are determined by the occurrence of initiating and terminating events [Gia03].

The thesis also presents an algorithm for generating a significant class of goals and domain properties from scenarios. The goal specifications can then be used to formally check or derive goal refinements, generate hazard and threat conditions, detect conflicts, and generate further scenarios that satisfy them [Lam09].

A method is also provided for inferring goals specified in Fluent Linear Temporal Logic (FLTL) from a set of Message Sequence Charts (MSC).

FLTL turns to be a convenient formalism for specifying state-based properties over an event-based operational model. The inferred properties follow specific property patterns; they are under the responsibility of single agents. The inferred property specifications need to be validated by the user. If a generated property turns to be inadequate, the user is asked to provide a counterexample scenario which will enrich the scenario collection. This technique reuses the invariant generation algorithm to infer property specifications.

Analyses along the operational dimension

Decision-based processes capture processes where the application of specific tasks and their sequencing depends on explicit decisions; the latter are based on the state of the environment in which the process operates. Decision-based processes in areas such as healthcare are often critical in terms of timing and resources. The thesis presents complementary techniques for analyzing decision-based processes. These techniques support a variety of checks on decision-based process models.

- *Guard analysis*. The guards on the various alternatives at a decision point in a task flow are verified to cover all possible cases (no missing branch), to not overlap (deterministic decisions), and to be satisfiable (reachability of subsequent tasks).
- *Detection of inadequate decisions*. A decision can be *inadequate* if it relies on incorrect information about the environment. This is typically due to state variables becoming outdated or inaccurate because of intermediate, possibly unexpected events. For example, in a cancer therapy process, the blood platelet level used at some critical decision point might not be the patient's actual one. Every decision point in a task flow is verified to be adequate.

- *Verification of temporal constraints on the process*. This technique may detect that some timing requirement is violated along some paths of the process model. A similar technique may be applied for other non-functional requirements on the process such as cost constraints, dose constraints in a therapy process and the like.
- *Verification of task preconditions*. The techniques may detect that some preconditions are violated along some path in the model.

To enable those different types of analyses, we have extended the modeling language of High-Level Message Sequence Charts (hMSC) with guards, timing and resource constructs. The resulting language, called Guarded High-Level Message Sequence Charts, provides simple notations that are close to the informal sketches provided by process stakeholders while having a precise formal semantics.

The integration of event-based and state-based specification styles is achieved by letting guards refer to fluents [Gia03] and tracking variables. *Tracking variables* are state variables tracking environment quantities that are not directly observable by the agents involved in the process.

To define the formal semantics of guarded hMSCs, Guarded Labeled Transition Systems (g-LTS) are introduced as an intermediate event-based formalism; they allow state transitions to be labeled by guards or events. The trace semantics of a g-LTS is defined in terms of LTS. Our analyses are performed at g-LTS level in order to keep state enumeration implicit.

These analyses differently instantiate the same algorithm for propagating abstract decorations through a g-LTS. This generic algorithm is intended to be as easy to instantiate as possible; no instantiation-specific proofs of each instantiated version are required. For each type of check, we need to instantiate the abstract decoration together with the rule for propagating decorations through a single transition.

Contributions

The main contributions of the thesis are the following:

- A dedicated algorithm for decorating each LTS state with state invariants on fluents together with its proof of correctness.
- An interactive algorithm for inferring safety properties from positive scenarios.
- The language of guarded hMSCs, a high-level formalism for process modeling that combines event-based and state-based style, together with its formal semantics.
- A generic algorithm for decorating each state of a guarded LTS with abstract decorations together with its proof of correctness.
- A variety of analysis techniques on decision-based process models:
    - to analyze decision alternatives against completeness, disjointness, and satisfiability.
    - to detect inadequate decisions due to inaccurate or outdated information about the environment.
    - to detect violations of non-functional requirements on timing, resource usage, cost and the like.

  These analyses are all based on instantiations of our abstract decoration algorithm. Such instantiations require minimal effort on the analysts who just need to provide the domain of the possible decorations together with a function that describes how the decorations are propagated through events.
- An evaluation of the proposed techniques on a real case study of significant size. We show in particular how these techniques may drive the model elaboration for a complex safety-critical therapy process for rectal cancer.

Organization

The thesis is structured as follows.

Chapter 2 describes some required background on Message Sequence Charts, High-Level Message Sequence Charts, Labeled Transition Systems, fluents and goals.

Chapter 3 describes the algorithm for generating state invariants on labeled transition systems. Our technique for inferring safety properties from scenarios is then presented as an application of this algorithm.

Chapter 4 introduces Guarded High-Level Message Sequence Charts, as a language for modeling decision-based processes. Guarded LTS are also introduced then as an intermediate language. The chapter describes their syntax and semantics.

Chapter 5 describes our generic algorithm for generating abstract decorations on guarded LTS.

Chapter 6 describes our different techniques for analyzing decision-based processes. Those analyses are defined as instantiations of the abstract algorithm of chapter 5.

Chapter 7 illustrates and evaluates the techniques described in the thesis on the elaboration of a process model for cancer therapy.

Chapter 8 discusses related work on the modeling and analysis of behavior and process models and the generation of invariants on programs.

Chapter 9 concludes the thesis and discusses open issues for further work.

Chapter 2 Modeling Multiple Views of Software-Intensive Systems

This section introduces some necessary background on interactions scenarios (Section 2.1), behavior models (Section 2.2) and intentional models (Section 2.3). A simple train system fragment will be used as a running example to illustrate modeling constructs. The scenarios involve three agent instances: a train controller, a train actuator/sensor, and a passenger. The train controller controls operations such as *start*, *stop*, *open doors*, and *close doors*. A safety goal requires train doors to remain closed while the train is moving. If the train is not moving and a passenger presses the alarm button, the controller must open the doors in emergency. When the train is moving and the passenger presses the alarm button, the controller must stop the train first and then open the doors in emergency.

2.1 Interaction scenarios as Message Sequence Charts

A *scenario* is a temporal sequence of interactions among system components. The word *system* refers here to the software-to-be together with its environment. A system is made of active components, called *agents*, that control system behaviors. Some agents form the environment; others form the system-to-be.

As scenarios are concrete and support a narrative style of description, they prove very effective in the early stage of the requirements engineering process to elicit or validate useful information from concrete examples. However, scenarios raise coverage problems as they only capture partial histories of interaction among system component instances. Moreover, they often leave the actual requirements implicit [Lam98a].

2.1.1. Basic Message Sequence Charts

Message sequence charts (MSCs) are commonly used for capturing multi-agent scenarios [ITU96]. A *MSC* is composed of vertical timelines associated with agent instances and horizontal arrows representing interactions among them. A timeline label declares the class of the corresponding agent instance. An arrow label captures an interaction event among the source and target agent instances; the event is synchronously controlled by the source and monitored by the target. A MSC timeline defines a total ordering on incoming/outgoing events whereas an entire MSC defines a partial ordering on all events.

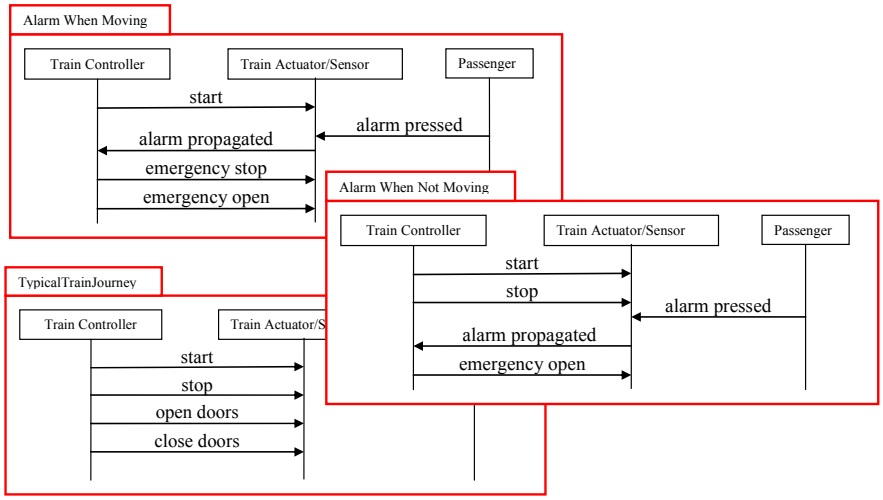


Figure 2-1 MSCs for the little train example

A MSC defines a partial order on events; it describes a set of event sequences. Such event sequences are called linearizations. The *linearizations* of a MSC capture all sequences of events that respect the total ordering defined by the timelines.

Figure 2-1 shows three simple MSC scenarios for the little train example. For example, *Alarm When Moving* captures the following scenario: "*The train is started by the controller; the latter then waits for external stimuli. A*

passenger presses the alarm button; the alarm is propagated to the controller; the latter then issues commands to stop the train and open the doors in emergency".

Note that MSCs have more sophisticated features such as conditions, timers, coregions, etc [ITU96]. In this thesis, we limit the language to a very simple form in order to be usable and understandable by stakeholders.

2.1.2. High-Level Message Sequence Charts (hMSC)

High-level MSCs (hMSCs) are directed graphs where each node is a MSC or a finer-grained hMSC. Edges indicate the acceptable orderings among scenarios. They allow for scenario sequencing, repetition, and reuse. We can thereby break up a complex scenario into manageable parts and specify how such parts relate to each other.

Figure 2.2 shows an hMSC for the little train example. All nodes of the graph should be refined. The task *Alarm When Moving* is refined in Figure 2.1.

The semantics of MSCs and hMSCs used in the thesis is the one introduced in [Uch03]. It is defined in terms of labeled transition systems and parallel composition, see Section 2.2 hereafter.

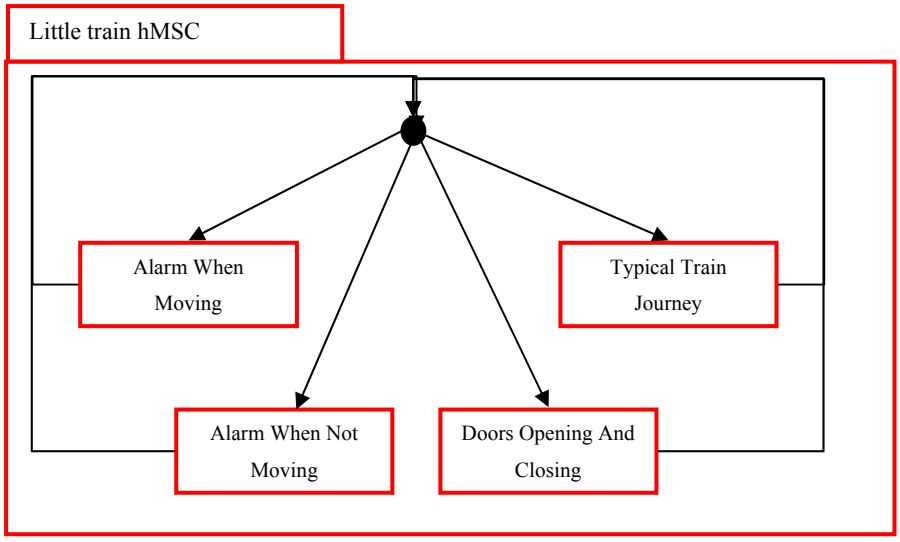


Figure 2-2 hMSC for the little train example

2.2 Behavior models as Labeled Transition Systems

A *system* is behaviorally modeled as a set of concurrent state machines – one per agent. Each agent is characterized by a set of states and a set of transitions between states. Each transition is labeled by an event. Our state machines are labeled transition systems (LTS) [Mil89].

A *LTS* is an automaton defined by a structure (Q, Σ, δ, q_0) where Q is a finite set of states, Σ is a set of event labels, δ is a transition function mapping Q x Σ to P(Q), and q_0 is the initial state. (P(Q) is the powerset of Q, the set of all subsets of Q).

A *trace* of a LTS (Q, Σ, δ, q_0) is a finite sequence of events $<e_1,\ldots,e_n>$, with $e_i \in \Sigma$, such that there exists $q_1,\ldots,q_n$ with $q_{i+1} \in \delta(q_i, e_{i+1})$ for all i: $0 \leq i < n$.

A LTS trace *reaches* a state q if the LTS may be in this state after having performed that event sequence from the initial state.

Complex systems can be modeled through parallel composition of LTS components [Hoa85]. The parallel composition of two LTS *P* and *Q,* denoted by *P* ‖ *Q,* models their joint behavior. The composed model behaves asynchronously but synchronizes on shared events.

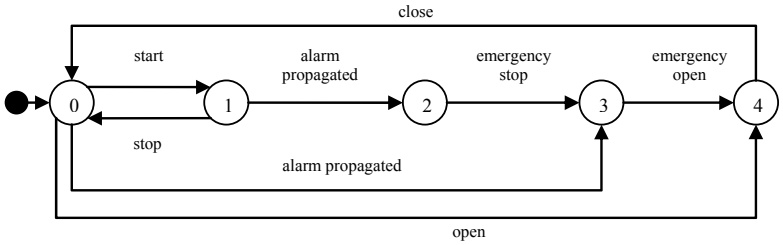


Figure 2-3 LTS for the train controller

As an example, Fig. 2-3 shows a LTS for the train controller in the little-train example.

The semantics of MSCs and hMSCs is defined in terms of LTS and parallel composition [Uch03]. A MSC timeline defines a unique finite LTS trace that captures a corresponding agent behavior. Similarly, the semantics of an entire MSC is defined in terms of the LTS modeling the entire system. MSCs define traces of the system LTS, that is, the parallel composition of each agent LTS.

2.3 Intentional models as goal graphs on fluents

A *goal* is a prescriptive statement of intent whose satisfaction requires the cooperation of agents forming the system. Unlike goals, *domain properties* are descriptive statements about the environment – such as physical laws, organizational rules, etc. Goals are structured into AND/OR refinement graphs showing how they contribute to each other [Lam09].

In the thesis, we will formalize goals and domain properties in Fluent Linear Temporal Logic (FLTL) [Gia03]. This formalism is convenient for

specifying temporal logic properties over an event-based operational model. Miller and Shanahan define fluents as "*time-varying properties of the world that are true at particular time-points if they have been initiated by an event occurrence at some earlier time-point, and not terminated by another event occurrence in the meantime. Similarly, a fluent is false at a particular time-point if it has been previously terminated and not initiated in the meantime*" [Mil99].

A fluent *Fl* is thus a proposition defined by

- a set $Init_{Fl}$ of initiating events,
- a set $Term_{Fl}$ of terminating events,
- an initial value $Initially_{Fl}$ that can be true or false.

The sets of initiating and terminating events must be disjoint. A fluent definition takes the form:

$$fluent\ Fl = < Init_{Fl}, Term_{Fl} > initially\ Initially_{Fl}$$

In the little train example, the fluents *doors_closed* and *moving* can be defined as follows:

$$fluent\ doors\_closed = <\{close\}, \{open\ doors, emergency\ open\}> initially\ true$$

$$fluent\ moving = <\{start\}, \{stop, emergency\ stop\}> initially\ false$$

A fluent *Fl* holds at some time if either of the following conditions holds:

(a) *Fl* holds initially and no terminating event has yet occurred;

(b) some initiating event has occurred and no terminating event has occurred since then.

Fluents are often represented by fluent automata [Gia03]. As an example, Fig. 2-4 shows a fluent automaton for the fluent

$$fluent\ Fl = < Init_{Fl}, Term_{Fl} > initially\ false.$$

Fluent automata are composed of two states, one in which the fluent does not hold (in the figure, the state labeled by 0), and one in which it does (in the

figure, the state labeled by 1). The initial state of the automaton is determined by the value of the attribute $Initially_{Fl}$. The automaton moves into the *holding* state when an action from its initiating set occurs, and to the *not holding* state when an action from its terminating set occurs.

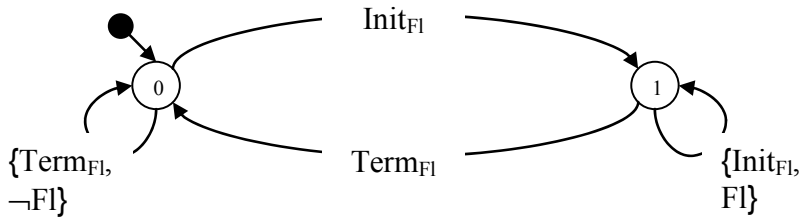


Figure 2-4 Example of fluent automata

A fluent is *controlled* by an agent if the agent controls all initiating and terminating events of the fluent. It is *monitored* by an agent if the agent controls or monitors all initiating and terminating events of the fluent.

The FLTL assertions for goals and domain properties use standard operators for temporal referencing such as:

- o (at the next smallest time unit),
- ◊ (some time in the future),
- □ (always in the future),
- $\mathcal{U}$ (always in the future until),
- $\mathcal{W}$ (always in the future unless),
- → (implies in the current state),
- ⇒ (always implies),

see [Man92]. For example, the goal "the doors shall remain closed while the train is moving" can be formalized in terms of the above fluents as follows:

$$DoorsClosedWhileMoving = \Box\ (moving \rightarrow doors\_closed)$$

Safety properties are properties stipulating that some "bad thing" may not happen [Alp86]. Safety properties formalized in FLTL may be represented by an automaton, called tester automaton [Gia03]. A *tester* for a property is a LTS extended with an error state such that every trace leading to the error state violates that property [Gia03]. A tester automaton is useful for model-checking safety properties on a LTS model.

Fig. 2-5 shows the tester LTS for the goal *DoorsClosedWhileMoving* (the error state is the black one). Any event sequence leading to the error state from the initial state corresponds to an undesired system behavior.

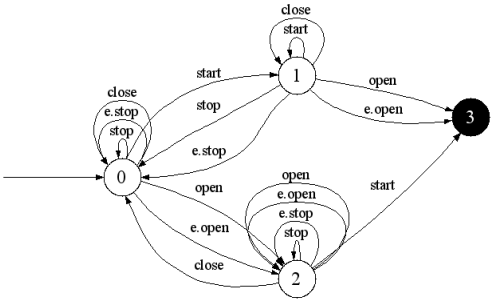


Figure 2-5 Tester LTS for the goal *DoorsClosedWhileMoving*

2.4 Summary

For better completeness, a model should capture all pertinent facets of the system [Nis89, Rum91, Nus94]. This chapter briefly introduced some formalisms for capturing the intentional and behavioral facets of the system.

- The *behavioral dimension* describes the admissible behaviors of the system in terms of sequence of events. The behavioral dimension is captured by scenarios and state machines. Scenarios represent typical examples of system behaviours through sequences of desired interactions among the considered agents, e.g., the software-to-be and its environment. State machines describe classes of admissible

behaviors in terms of sequence of state transitions resulting from the occurrence of events.

- The *intentional dimension* describes the system objectives to be achieved by agents. It is captured by goals which are prescriptive statements of intent whose satisfaction requires the cooperation of agents forming the system. Goals are defined in Fluent Linear Temporal Logic (FLTL). FLTL is convenient for specifying temporal logic properties over an event-based operational model.

The operational dimension of a system will be captured by process models that are introduced in Chapter 4.

Chapter 3 Generating State Invariants on Event-Based System Models

A *state invariant* on a state machine model is an assertion on a specific state which holds every time this state is visited.

The annotation of state machines with state invariants provides multiple benefits:

- the understandability and documentation of the state machine is improved;
- the invariants can be shown to the analyst for validation and error detection;
- they can be used by analysis tools for more efficient analysis [Jef98];
- they can be used by code generators to improve the quality of the generated code. When the entire code of the application cannot be fully generated, the generated fragments must be readable by developers. The choice of variables in the generated code, their names and property annotations may then be quite useful.

Section 3.1 presents an algorithm that decorates each state of a LTS with an invariant consisting of a conjunction of fluents or their negation. Section 3.2 extends this first algorithm to decorate each state with the most specific state invariant on fluents. Section 3.3 presents an application of those algorithms to the generation of goal specifications from scenarios.

3.1 Generating conjoined literals on fluents

We first show how fluents can decorate a node on a single LTS trace. Then we consider the general case of node decoration for a node pertaining to

multiple traces. The generation algorithm is presented next and then illustrated on a simple example.

3.1.1. Fluent values for a node on a single trace

As introduced in Section 2.3, a fluent *Fl* is true after a LTS trace σ if and only if one of the following conditions holds [Gia03]:

- Fl holds initially and no terminating event has occurred in σ,
- Some initiating event has occurred in σ with no terminating event occurring since then.

The following recursive version of this definition is used by our generation algorithm. A fluent *Fl* is true after a finite LTS trace σ if and only if one of the following conditions holds:

- σ is empty and *Fl* holds initially;
- the last event of σ belongs to the initiating events;
- the last event *e* of σ does not belong to the terminating events and the fluent is true after σ' where $\sigma = <\sigma', e>$.

3.1.2. Fluent values for a node on multiple traces

The value of a fluent *Fl* at a LTS node *q* is not necessarily either true or false. There might be two LTS traces σ_1 and σ_2 reaching *q*, such that the value of *Fl* after σ_1 is true and the value of *Fl* after σ_2 is false.

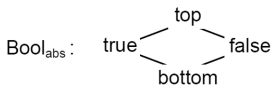
For example, consider the train controller LTS in Fig. 2.3 and the following fluent:

emergency = <{alarm propagated}, {start}> initially false.

The value of this fluent at the initial state after the LTS trace σ_1=<*start*, *stop*> is false. The value of the fluent at the initial state after the LTS trace σ_2=<*alarm propagated*, *emergency open*, *close doors*> is *true*.

The value of *emergency* at that state will in such cases be set to the value *top*. The value "top" for a fluent at a LTS node means that the value of the fluent can be either *true* or *false* at that state depending on the trace leading to it. A state might also be unreachable from the initial state. In that case, the value of *Fl* at that state will be set to the value *bottom*.

The possible values of a fluent at a LTS node thus belong to a lattice $Bool_{abs}$ defined as follows:



In this lattice, the supremum operator *sup* is logical disjunction ($\vee$); the partial order $\leq$ is logical implication ($\supset$).

3.1.3. Generating atomic invariants on fluents

The specification of our generation algorithm is given in Fig 3.1.

GIVEN a LTS model (Q, Σ, δ, q_0)
a *fluent* $Fl = < Init_{Fl}, Term_{Fl} >$ *initially* $Initially_{Fl}$,

FIND *a node decoration function*
$Q \rightarrow Bool_{abs}$
defined as follows:
deco (q) =
'true' iff *Fl* is true for all LTS traces reaching state *q*
'false' iff *Fl* is false for all LTS traces reaching state *q*
'top' iff *Fl* is true for some LTS traces reaching state *q* and false for others
'bottom' iff state *q* is not reachable from the initial state

Figure 3-1 Specification of the LTS decoration algorithm on a single fluent

The idea behind the algorithm is the following. At each step, every state has a decoration. At the beginning, in every state, the fluent value is initialized to *bottom* except for the initial state where the fluent is initialized to the initial value provided by its definition. The algorithm propagates fluent values in the state machine according to the propagation rule in Fig. 3-2, derived from the above recursive characterization of fluent values.

In Fig. 3-2, *source* $\in$ *Q, event* $\in$ Σ, *target* $\in$ δ*(source, event),* and $\vee$ is the supremum operator on the lattice $Bool_{abs}$.

```
if deco(source) ≠ bottom then
        if event ∈ Init_Fl then
                newVal := true
        else
                if event ∈ Term_Fl then
                        newVal := false
                else
                        newVal := decor(source)
        deco(target) := deco(target) ∨ newVal
```

Figure 3-2 Propagation rule for a node decoration

The algorithm applies this propagation rule for each transition until a fixpoint is reached. At this point, no state decoration is changing anymore if the propagation rule is further applied on each transition.

Based on this decoration propagation rule, the algorithm for one fluent is given in Fig. 3-3. The algorithm keeps track of the set *ToExpl* of states that have been updated and to which the propagation should be further applied. The algorithm terminates when the set *ToExpl* is empty, that is, when there is no state that must further propagate its value.

The algorithm always terminates because the set *ToExpl* will eventually be empty; a state can change his decoration at most twice, namely, from bottom to either true or false, and from true or false to top. The reason is that the lattice is finite and the decoration can only go up in the lattice in view of the supremum operator.

```
Input: A LTS (Q,Σ,δ,q0)
       A Fluent Fl = <Init_Fl, Term_Fl> initially Initially_Fl
Output: deco: Q → Bool_abs  as defined in Fig 3.1.

deco(q0) := Initially_Fl
forall q ∈ Q \ {q0} do deco (q) := bottom
ToExpl := {q0}
while ToExpl ≠∅ do
        source := getOne(ToExpl)
        ToExpl := ToExpl \ {source}
        forall (target,event) such that target ∈ δ(source, event) do
                if  event ∈ Init_Fl then
                        newVal := true
                else
                        if  event ∈ Term_Fl then
                                newVal := false
                        else
                                newVal := deco(source)
                if not (newVal ⊃ deco(target)) then
                        deco(target) := newVal ∨ deco(target)
                        ToExpl := ToExpl ∪ {target}
return deco
```

Figure 3-3 LTS decoration algorithm on a single fluent

The formal correctness of our algorithm is given in Annex A. The proof roughly consists in showing that, when the algorithm terminates, each fluent value after a single trace reaching the state *q* is lower in the lattice than the decoration of *q*.

3.1.4. Example

Fig. 3-4 illustrates the first three steps of our algorithm for the following fluent on the *Train Controller* state machine shown in Fig. 2-3 :

moving = <{start}, {stop, emergency stop}> initially false

(Step 0) We initialize the values of *moving* for each state to *bottom* except for the initial state where *moving* gets its initial value, see Fig 3-4.a.

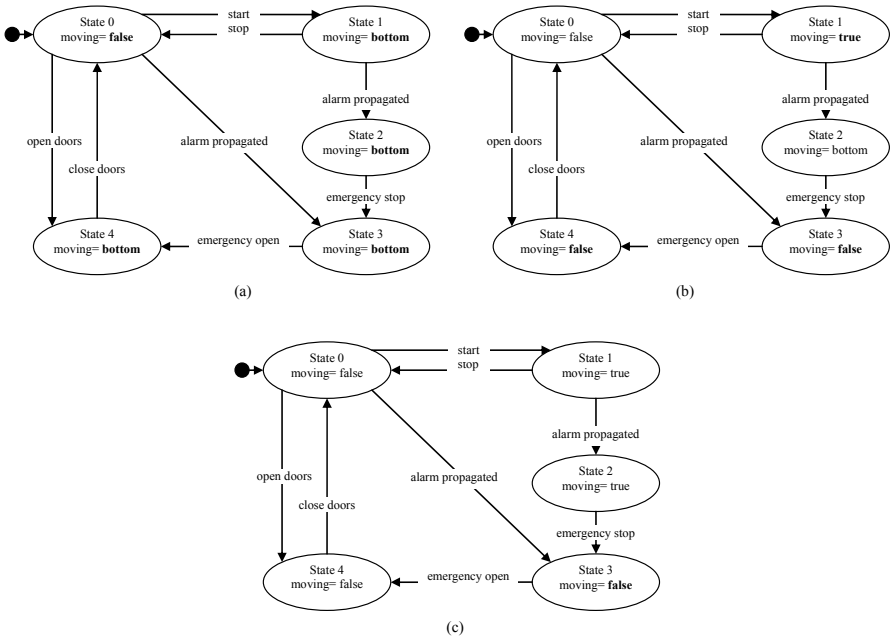


Figure 3-4 Executing the algorithm for fluent *moving*

(Step 1) *State 0* is in *ToExpl*. We propagate its value to all its successor states. There are three outgoing transitions to *State 1*, *State 3* and *State 4*, respectively. The fluent value for *State 0* and *State 2* remains the same. For *State 1*, *newVal* gets *true* because *start* belongs to the initiating events; therefore *moving* gets (*bottom* $\vee$ *true),* that is, *true*. We thus add *State 1* to the set *ToExpl*. For *State 3* and *State 4*, the events *alarm propagated* and *open doors* do not belong to the initiating or terminating events. The fluent values for *State 3* and *State 4* are (*bottom* $\vee$ *false)*, that is, *false*. We add thus *State 3* and *State 4* to *ToExpl*, see Fig 3-4.b.

(Step 2) We need to choose one element in *ToExpl*. Let us choose *State 1*. This state has two outgoing transitions. Because *stop* is a terminating event, *moving* at *State 0* gets (*false* $\vee$ *false),* that is, *false*. Because the decoration of *State 0* has not changed, we do not add *State 0* to *ToExpl*. The event *alarm propagated* does not belong to the initiating or terminating events of the

fluent; therefore, *moving* at *State 2* gets (*bottom* $\vee$ *true),* that is*, true*. We then add *State 2* to *ToExpl*, see Fig 3-4.c.

The process goes on until *ToExpl* becomes empty. The decorations computed after *Step 2* turn to be the right ones. A fixpoint is reached in this case after *Step 2*; all new propagations will leave the state decorations unchanged.

3.1.5. From atomic invariants to conjoined literals on fluents

A straightforward way of generating a state invariant at each LTS node is to apply the previous algorithm once for each fluent and then take the conjunction of the results. For example, let the algorithm separately compute the following atomic invariants for the initial state:

moving = false,

doors_closed = true.

A state invariant is obtained by taking the conjunction of all such atomic invariants in that state. *Top* values will be dropped from this conjunction as they yield no information.

In our example, the computed state invariant at the initial state will then be:

$\neg$ *moving* $\wedge$ *doors_closed.*

The algorithm implemented in our tool is an equivalent, optimized version where the value of all fluents is calculated within a single loop (rather than one loop per fluent).

3.2 Generating most specific state invariants

The algorithm in the previous section may be used to document every LTS state with literals corresponding to the value of each fluent. However, the computed invariants might be too general.

For example, if $((A \vee B) \wedge \neg C)$ is the most "accurate" invariant at some state, the invariants for each fluent will be *A="top"*, *B="top", C="false";* the computed state invariant will be $\neg C$. We thus cannot say anything about the value of *A* and *B* at this state as *A* and *B* can be true or false. This invariant is too general; it does not tell us that the value of *A* and *B* are related - in this case, *A* and *B* cannot both be false at the same time.

The algorithm in this section extends the previous one by computing the most specific invariant at each state.

A *most specific invariant* for a state *q* must meet two properties:

- It should not be too specific, that is, it should be an invariant satisfied by all fluent assignments of traces reaching *q*.
- It should not be too general, that is, each fluent assignment satisfying the invariant should result from at least one trace reaching *q*.

The previous algorithm generates decorations satisfying the first condition but not the second one. As an example, $\neg A \wedge \neg B \wedge \neg C$ satisfies $\neg C$ but will never result from a trace reaching that state.

The organization of this section is similar to the one of Section 3.1. We first show how Boolean expressions on fluents can decorate a node on a single LTS trace. Next we consider the general case of node decoration for a node pertaining to multiple traces. From this, we give the algorithm and illustrate it on a simple example.

3.2.1. Fluent assignments for a node on a single trace

We are interested here in the condition characterizing each LTS state along a single trace in terms of the values of all fluents involved. Let us call *TraceCond(σ)* such condition right after a LTS trace σ. This condition will be a conjunction over the fluents involved - each having a single value,

either true or false, that can be computed using the fluent definition in Section 3.1.1. We will represent the set of all such conjunctions by 2^{Φ}.

Example

In the train running example, the condition after an empty trace is

$$TraceCond(<>) = \neg moving \wedge doors\_closed,$$

as the initial value of *moving* is *false* while *doors_closed* is initially *true*. After a trace only consisting of the event *start*, the value of this condition is equal to

$$TraceCond(<start>) = moving \wedge doors\_closed,$$

as *start* is an initiating event of *moving*.

(End of example)

As the example suggests, the value of *TraceCond* after a trace $<\sigma,e>$ can be computed from the value of *TraceCond* after the trace σ and the event *e*.

Let us introduce an auxiliary function *Effect* that computes the propagation of a Boolean expression on fluents through a single event:

$$TraceCond(<\sigma, e>) = Effect(TraceCond(\sigma), e).$$

Effect(TraceCond(σ*), e)* can be computed in two steps as follows:

- First, we remove from *TraceCond(*σ*)* all fluent literals for which *e* is an initiating or terminating event.
- Next, for each fluent *Fl* for which e is an initiating event, we add *Fl* as conjunct; and for each fluent *Fl* for which *e* is a terminating event, we add $\neg Fl$ as conjunct.

Example

As previously seen,

$$TraceCond(\varnothing) = \neg moving \wedge doors\_closed.$$

Effect(TraceCond(∅), start) can be computed by performing the two steps. After the first step, we obtain *doors_closed* because *start* is an initiating event of *moving*. The second step adds *doors_closed* as conjunct:

$$Effect(TraceCond(\emptyset), start) = moving \wedge doors\_closed.$$

(End of example)

To formally support the first step, we define an operator that drops a variable from a Boolean formula without affecting its satisfiability. For a variable v dropped from a formula B, we will denote this operator by $B[\backslash v]$. The *Drop* operator is defined as follows:

$$B[\backslash v] =_{def} B|_{v=true} \vee B|_{v=false}.$$

In this definition, $B|_{v=val}$ is B where each occurrence of variable v is replaced by the value *val*.

Example

$$\begin{aligned}(P \wedge Q)\,[\backslash P] &= (P \wedge Q)|_{P=true} \vee (P \wedge Q)|_{P=false} \\ &= (true \wedge Q) \vee (false \wedge Q) \\ &= Q \vee false \\ &= Q.\end{aligned}$$

(End of example)

The *Drop* operator is extended to support a set of variables. The *Drop* of a set of variables $V = \{v_1,\ldots,v_n\}$ from a Boolean formula B is defined as follows

$$B[\backslash V] = ((((B[\backslash v_1])[\backslash v_2])[\backslash v_3])\ldots\,[\backslash v_n])$$

Example

$(P \wedge Q \wedge R)\ [\backslash\{P,Q\}] = ((P \wedge Q \wedge R)|_{P=true} \vee (P \wedge Q \wedge R)|_{P=false})[\backslash Q]$

$= ((true \wedge Q \wedge R) \vee (false \wedge Q \wedge R))[\backslash Q]$

$= (Q \wedge R)[\backslash Q]$

$= R.$

(End of example)

Note that the *Drop* operator may be distributed over disjunctions:

$(A \vee B)\ [\backslash V] = A[\backslash V] \vee B[\backslash V]$

Using the *Drop* operator, we can now define the function Effect more formally:

$Effect(B, e) = (B[\backslash(\{v_i\} \cup \{v_t\})]) \wedge (\wedge_i v_i) \wedge (\wedge_t \neg v_t)$

where $\{v_i\}$ *is the set of all fluents* v_i *such that e is among their initiating event,*

$\{v_t\}$ *is the set of all fluents* v_t *such that e is among their terminating event.*

Example: In the train example, let us assume that the decoration at a LTS node is

$moving \wedge doors\_closed$.

The event *stop* is a terminating event for the fluent *moving*. Therefore, we get

$Effect((moving \wedge doors\_closed), stop)$

$= (moving \wedge doors\_closed)[\backslash moving] \wedge \neg moving$

$= ((true \wedge doors\_closed) \vee (false \wedge doors\_closed)) \wedge \neg moving$

$= doors\_closed \wedge \neg moving$

(End of example)

Using the *Effect* function, we can define *TraceCond* formally. Consider a set of fluents Fl_i defined as follows

$fluent\ Fl_i = \langle \{Init_{Fli}\}, \{Term_{Fli}\} \rangle\ initially\ Initially_{Fli}.$ $\quad (1 \leq i \leq n)$

The fluent assignment after a trace σ can be computed as follows:

- $TraceCond(\{\}) = \bigwedge_i InitiallyFl_i$
- $TraceCond(<\sigma,e>) = Effect(TraceCond(\sigma), e)$

3.2.2. Fluent assignments for a node on multiple traces

If we now consider multiple traces leading to a node, the decoration of this node will not necessarily belong to 2^{Φ}. There might be two LTS traces σ_1 and σ_2 reaching a state q such that $TraceCond(\sigma_1) \neq TraceCond(\sigma_2)$.

For example, let us consider a system with two fluents *F1* and *F2*. Let us assume two traces σ_1 and σ_2 reaching a state q, such that:

$TraceCond(\sigma_1) = \neg F1 \wedge \neg F2$

and $TraceCond(\sigma_2) = F1 \wedge \neg F2$.

Our algorithm should decorate each state with their most specific invariant. If there are no other LTS traces reaching q, the decoration $deco(q)$ should:

- be implied by the two path conditions, that is,

  $TraceCond(\sigma_1) \supset deco(q)$, $TraceCond(\sigma_2) \supset deco(q)$

- not be implied by fluent assignments different from $TraceCond(\sigma_1)$ and $TraceCond(\sigma_2)$.

The decoration of state q will therefore be set to

$TraceCond(\sigma_1) \vee TraceCond(\sigma_2)$.

In our example, it will reduce to $\neg F2$.

The most specific invariant for a node will be a disjunction of fluent assignments produced by traces reaching this node. We will denote the set of possible decorations for a node by the powerset $P(2^{\Phi})$. An element of $P(2^{\Phi})$, i.e. a set of formulas of 2^{Φ}, is represented by the disjunction of such formulas.

$P(2^{\Phi})$ forms a lattice where

- the supremum operator *sup* is disjunction,
- the partial order $\leq$ is logical implication,
- the bottom and top elements are *false* and *true,* respectively.

3.2.3. Invariant generation algorithm

The specification of our algorithm for generating the most specific state invariant is given in Fig. 3-5. The algorithm computes a decoration for each state that should meet the two following conditions:

- (INV) The computed decorations should be state invariants, i.e., for all traces σ reaching a state q, *TraceCond*(σ) implies *deco(q)*.
- (MSI) The computed decorations should be most specific invariants, i.e., each fluent assignment satisfying the invariant should result from at least one trace reaching q.

GIVEN *a LTS model* (Q, Σ, δ, q_0)
A set Φ *of fluents* $Fl_i = \langle Init_{Fli}, Term_{Fli} \rangle$ *initially* $Initially_{Fli}$

FIND *a node decoration function*
$deco: Q \rightarrow P(2^{\Phi})$
such that for any state q:

(INV) For any LTS trace σ *reaching q:*
$PathCond(\sigma) \supset deco(q)$

(MSI) For any fluent assignment $v \in 2^{\Phi}$ *such that*
$v \supset deco(q)$,
there exists a LTS trace σ *reaching q such that*
$PathCond(\sigma) = v$

Figure 3-5 Specification of the LTS invariant generation algorithm

The algorithm has the same structure as the one computing decorations with single fluent only.

It also proceeds by symbolic execution until a fixpoint is reached. At each step, every state has a decoration.

- Initially, the decoration is *false* for every state except the initial state; the latter is decorated by the conjunction of initial values of each fluent.
- The algorithm propagates fluent literals through the state machine according to the propagation rule in Fig. 3-6 where *source* $\in$ *Q, event* $\in$ *Σ, target* $\in$ *δ(source, event)*. When such propagation terminates, a state decorated with *false* means that no OR-combination of fluent literals holds in that state, that is, the state is unreachable.

```
if deco(source) ≠ false then
        newVal := Effect(deco(source), event);
        deco(target) := deco(target) ∨ newVal
```

Figure 3-6 Propagation rule for a node decoration with multiple fluents

Fig. 3-7 shows the extended invariant generation algorithm that computes most specific invariants. The set *ToExpl* collects all states whose decoration did change and to which propagation should be further applied.

```
Input: A LTS (Q,Σ,δ,q0)
       A set Φ of fluents Fli = <Init_Fli, Term_Fli> initially Initially_Fli
Output: deco: Q → P(2^Φ) as defined in Fig. 3-5

deco(q0) := ∧i InitiallyFli
forall q ∈ Q \ {q0} do deco(q) := false
ToExpl := {q0}
while ToExpl ≠∅ do
        source := getOne(ToExpl)
        ToExpl := ToExpl \ {source}
        forall (target,event) such that target ∈ δ(source, event) do
                newVal := Effect(deco(source), event)
                if not (newVal ⊃ deco(target)) then
                        deco(target) := newVal ∨ deco(target)
                        ToExpl := ToExpl ∪ {target}
return deco
```

Figure 3-7 Algorithm for LTS decoration with most specific invariants

The algorithm terminates when ToExpl is empty. This set will eventually be empty as a state can change its decoration at most $2^{|\Phi|}$ times, where $|\Phi|$ is the number of fluents. In the worst case, a state will be decorated by the disjunction of all possible literal combinations, that is, *true*. As there are *n* states to decorate, the complexity of the algorithm is $O\ (n \times 2^{|\Phi|})$. The algorithm is thus linear in the number of states but in the worst case exponential in the number of fluents. Fortunately, the number of fluents is in practice not very large.

The formal correctness of the algorithm is proved in Annex A.

3.2.4. Example

Fig. 3-8 illustrates the first three steps of the algorithm in Fig. 3-7 for the following fluents on the *Train Controller* state machine shown in Fig. 2-3:

fluent moving = <{start}, {stop, emergency stop}> initially false

fluent doors_closed = <{close doors},

{open doors, emergency open}> initially true

(Step 0). All decorations are initialized to *false* except the initial state, which gets the initial condition $\neg moving \wedge doors\_closed$.

(Step 1) *State 0* is in *ToExpl*. We propagate its decoration to *State 1*, *State 3* and *State 4*. The event *alarm_propagated* does not belong to initiating or terminating events of fluents; the propagated decoration *newVal* is simply the decoration of *State 0*, i.e., $\neg moving \wedge doors\_closed$. *State 1* and *State 4* are decorated by the disjunction of *newVal* with their old decoration:

for *State 1*, $(moving \wedge doors\_closed) \vee false$,

for *State 4*, $(\neg moving \wedge \neg doors\_closed) \vee false$.

State 1, State 3 and *State 4* are then added to *ToExpl* as their decorations have changed.

(Step 2) Let us assume that *State 1* is chosen in *ToExpl.* Its decoration is propagated to its only successor, *State 2*. The event *alarm_propagated* does not belong to initiating or terminating events of fluents; the propagated decoration *newVal* is simply the decoration of *State 1*. *State 2* is therefore decorated with

(*moving* $\wedge$ *doors_closed)* $\vee$ *false,*

which reduces to

moving $\wedge$ *doors_closed.*

State2 is therefore added to *ToExpl.*

The process goes on until *ToExpl* becomes empty. The decorations computed after *Step 2* turn to be the right ones. A fixpoint is reached in this case after two steps as all new propagations do not change state decorations.

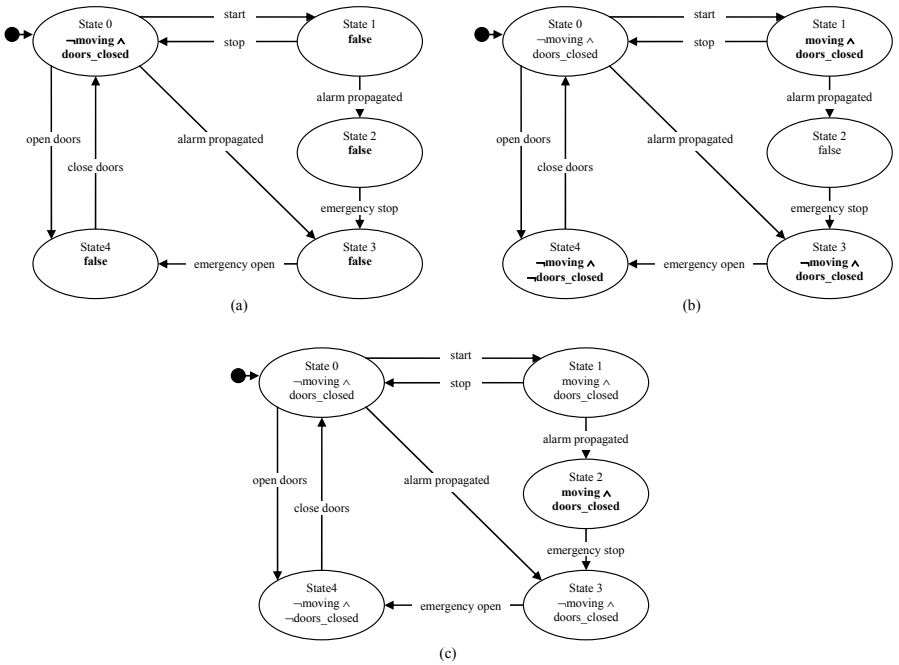


Figure 3-8 Executing the algorithm for fluent moving and doors_closed

3.3 Application: generating goal specifications from scenarios

Generated invariants are useful for different types of model analysis. This section shows how safety properties may be inferred from scenarios using generated invariants. Other applications of invariant generation techniques will be found in Chapter 6.

Scenario descriptions are fragmentary, operational, and leave the underlying goals, assumptions, and domain properties implicit. On the other hand, goal specifications can be used to formally check or derive goal refinements, generate hazard and threat conditions, detect conflicts, and generate further scenarios that satisfy them [Lam09].

We focus on specific property patterns that are under responsibility of single agents. An inferred property will therefore be a requirement on a software agent, an assumption on an environment agent, or a domain property [Lam09].

Our procedure for inferring property specifications from scenarios is specified as follows:

GIVEN a set of positive scenarios,
a set of fluents,

FIND a conjunctive set of properties covering all input scenarios, and taking the form:

- $\square\,(A \rightarrow C)$ (*Maintain/Avoid* properties)
- $\square\,(A \rightarrow oC)$ (*Immediate Response* properties)

where A and C are state assertions without temporal operators.

The inference of *Maintain/Avoid* properties is detailed in Section 3.3.1 whereas the inference of *Immediate Response* properties is discussed in Section 3.3.2.

3.3.1. Inferring *Maintain/Avoid* goals

Let us consider the two positive scenarios in Fig.3-9. The state predicates *S0, S1, S2, S3, S4, S5* along agent *B*'s timeline are fluent conjunctions that hold between the corresponding interactions. The following assertion on agent *B* then holds in any of *those* states:

$$S0 \vee S1 \vee S2 \vee S3 \vee S4 \vee S5$$

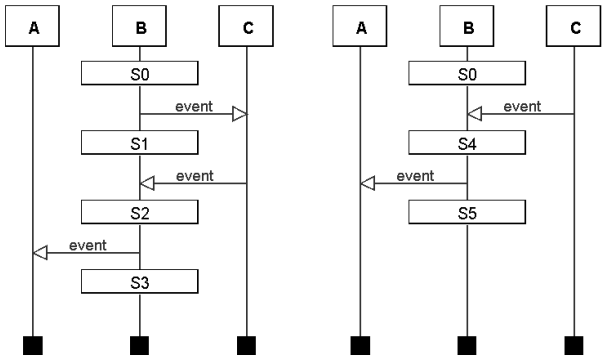


Figure 3-9 State predicates along timelines of an agent in multiple scenarios

Let *Inv* denote this assertion. If those two scenarios were to cover all states of agent *B*'s LTS, we could infer that *Inv* is preserved under all behaviors of this agent and get the safety property:

$$\Box\ Inv.$$

As the set of positive scenarios in which the agent is involved is likely not to be complete, this generalization can be unsound; it must therefore be validated by the user.

It is thus crucial for the candidate property to be in structured form and easy to understand. To achieve this we transform it into a minimal conjunctive normal form (CNF). This yields a logically equivalent set of simpler and shorter candidate invariants.

Rejection of one of these by the user means that the scenario collection is not complete. The user should then be asked to provide a counterexample to the rejected invariants, to be covered as additional positive scenario.

The procedure for inferring *Maintain/Avoid* properties on a single agent is now detailed step by step on our running example.

We start from the positive scenarios in Fig.2.1 together with the following fluent definitions:

fluent doors_closed = <{close doors}, {open doors, emergency open}>
initially true

fluent moving = <{start}, {stop, emergency stop}> initially false

fluent alarmed = <{alarm propagated}, {emergency open}>
initially false

Step 1: Compute fluent-based state predicates along the agent's timelines

To obtain the local state predicates S_i along the agent's timelines, we need to decorate these timelines with the fluents holding at the corresponding time points.

As we are looking for prescriptions on this agent, we restrict ourselves to the fluents *controlled* by the agent. Properties of form $\Box\ (A \rightarrow C)$, where A contains monitored fluents and C contains controlled ones, would be unrealizable by the agent [Let02a]; we are not making the synchrony hypothesis where agents can react instantly to their environment. Said otherwise, if A becomes *true*, the agent needs at least one (smallest) time unit to set B to *true*.

Conjunctions of controlled fluents that hold at specific points of an agent timeline are called *controlled predicates*. The algorithm for decorating timelines with controlled predicates is a simplified version of the algorithm in Section 3.2, particularized to a MSC timeline instead of an entire LTS.

Let us focus on the *TrainController* agent. It controls the fluents *doors_closed* and *moving* as it performs their initiating and terminating events.

The decoration is generated from top to bottom. First, we annotate the initial state *S0* of the timeline with the initial fluent values, yielding

$$\neg moving \wedge doors\_closed,$$

as *moving* is initially *false* and *doors_closed* is initially *true*. Recursively, we compute the next state decoration using the above definitions of fluents *moving* and *doors_closed*. After the *start* event, the decoration is:

$$moving \wedge doors\_closed,$$

because *start* is an initiating event of *moving*; the fluent *doors_closed* does not change as *start* is not among its initiating/terminating events. We continue until the end of the scenario is reached (see Fig. 3-10).

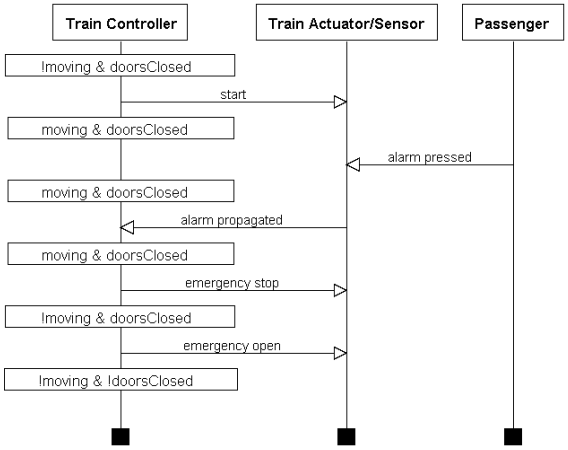


Figure 3-10 Train controller decorated with controlled fluents

As MSCs define a partial order, a MSC can have multiple linearizations. The linearizations of a MSC capture all sequences of events that respect the total ordering defined by the timelines. In case of multiple linearizations, state predicates should be computed for all such linearizations.

Step 2: Form the candidate invariant

A single scenario contributes to the candidate global invariant through the following assertion:

$$InvSc = \bigvee_{i \in 0...n} cp_i$$

where cp_i is the controlled predicate decorating the timeline right after the first i events.

In our example we get (see Fig.3-9):

$$\neg moving \wedge doors\_closed \vee moving \wedge doors\_closed$$
$$\vee moving \wedge doors\_closed \vee moving \wedge doors\_closed$$
$$\vee \neg moving \wedge doors\_closed \vee \neg moving \wedge \neg doors\_closed$$

Similarly we compute the contributions of all other input scenarios to obtain the candidate global invariant:

$$Inv = \bigvee_j InvSc_j$$

Step 3: Normalize the candidate invariant in minimal CNF form

The candidate global assertion is then transformed into a minimal conjunction of disjunctions, using standard CNF techniques [Rus95]. In our example, *Inv* is reduced to

$$\neg moving \vee doors\_closed.$$

Step 4: Generalize the candidate CNF invariant

The generalization to any state is simply achieved by prefixing *Inv* with the "always" FLTL operator. In our example, we get

$$\Box (\neg \text{moving} \vee \text{doors\_closed}).$$

Step 5: Validate each conjoined property with the user

Each conjunct in the generalized CNF invariant is shown to the user for validation. There is a readability issue here as each conjunct can be presented in alternative, equivalent ways. In our example, we obtain one requirement only that can be presented in either of the following forms:

- $\square\ (\neg\, moving \vee doors\_closed)$
- $\square\ (\ moving \rightarrow doors\_closed)$
- $\square\ (\neg\, doors\_closed \rightarrow \neg\, moving)$
- $\square\ \neg\, (\neg\, doors\_closed \wedge moving)$

Heuristics for increased readability are needed here. To mimic "if… then…" rules, we favored implications with a minimal number of negation symbols. Other heuristics might deserve further attention.

The user may react in three ways to the candidate properties presented.

- *Accept:* The property is added to the specification.
- *Reject:* This means that the current scenario collection does not cover all agent states. The user is then asked to provide a counterexample to the rejected property in order to enrich the collection with a new positive scenario to be covered. Property inference thus turns to be an effective way of checking whether a scenario collection is complete enough and, if not, soliciting new behavior examples.
- *Don't know:* the generated property might be hard to understand. In this case, it might be ignored.

3.3.2. Inferring *Immediate Response* goals

The second property pattern we can infer takes the form

$$\square\ (A \rightarrow \mathbf{o}\ C)$$

where A may contain monitored fluents and C contains controlled fluents only. This stimulus-response property pattern arises fairly frequently in reactive event-based systems.

Such properties are also important as they may yield *trigger* and *required* preconditions on operations [Lam09]. Goals of the form $\Box(A \rightarrow \mathbf{o}\ C)$ may be operationalized by two operations [Let02b]:

- the first with domain precondition $\neg C$, domain postcondition C, and required trigger condition A;
- the second with domain precondition C, domain postcondition $\neg C$ and required precondition $\neg A$.

As an example, the goal $\Box$ *(alarmed* $\rightarrow o \neg$ *moving)* requires the train to stop just after a passenger pushes on an alarm button. This goal yields the required trigger condition *alarmed* on the operation *Stop*, and the required precondition $\neg$ *alarmed* for the operation *Start*.

Such properties are not necessarily closed under stuttering [Lam94]; their satisfaction by an event trace may be affected by insertion or removal of unobservable events. In the *Immediate Response* properties we consider, the "next" operator means "in the next smallest time unit" and refers to the alphabet of the input scenarios. We are protected against stuttering as long as such properties are not combined with others defined on another alphabet.

The inferred *Immediate Response* properties must thus be handled with care – e.g., they should not be used when composing the system with components on different alphabets. In any case, they need to be validated by the user as well.

The inference of *Immediate Response* properties is fairly similar to *Maintain/Avoid* properties. We briefly highlight the differences.

Step 1: Compute fluent-based state predicates along the agent's timelines.

Each agent timeline is decorated with the agent's controlled *and monitored* fluents by use of the algorithm in Fig. 3-7, for each linearization of each MSC. Conjunctions of controlled and monitored fluents that hold at specific points on an agent timeline are called *state predicates*. For our *TrainController* agent, the timelines are decorated using three fluents: the controlled fluents *moving* and *doors_closed*, and the monitored fluent *alarmed*.

Step 2: Form the candidate invariant.

The single-scenario invariant takes a different form here, namely,

$$InvSc = \vee_{i \in 0...n-1} (sp_i \wedge o\, cp_{i+1})$$

where sp_i is the state predicate right after the first i events and cp_i the controlled predicate after those i events.

The global candidate invariant on all scenarios is then:

$$Inv = \vee_j InvSc_j$$

Step 3: Normalize the candidate invariant in minimal CNF form.

For the three positive scenarios in Fig. 2-1 we obtain:

$$\begin{aligned} Inv = \; & (\neg alarmed \vee o \neg moving) \\ & \wedge (doors\_closed \vee \neg moving) \\ & \wedge (doors\_closed \vee o \neg moving) \\ & \wedge (\neg moving \vee o\, doors\_closed) \\ & \wedge (o\, doors\_closed \vee o \neg moving) \\ & \wedge (doors\_closed \vee o\, doors\_closed) \\ & \wedge (\neg alarmed \vee moving \vee o \neg doors\_closed) \end{aligned}$$

Such disjunctions can be rewritten as implications taking the form

$$A \rightarrow o\, C,$$

where A is a conjunction and C a disjunction.

Step 4: Generalize the candidate CNF invariant.

This step is similar; we just add the "□" prefix.

Step 5: Validate candidate properties with the user.

Here we perform some filtering on the properties shown to the user.

- Properties without "next" operator are not presented as they are already shown as *Maintain/Avoid* properties.
- Properties where all fluents are prefixed with the "next" operator are not presented for a similar reason, e.g.,

  □(o doors_closed ∨ o ¬ moving)
- Tautologies resulting from built-in LTS semantic assumptions are not shown.

Let us explain the last point. Consider the following fluents:

fluent $A = \langle a_{Init}, a_{Term} \rangle$

fluent $B = \langle b_{Init}, b_{Term} \rangle$

together with the assertion □ $(A \wedge B \rightarrow o\ A \vee o\ B)$

This assertion is a tautology in view of the LTS *one-input assumption*. The latter states that exactly one input event occurs at every state transition. As a_{Term} and b_{Term} cannot occur at the same time, if A and B are true in the current state, either A or B will be true in the next state.

A tautology tester is a universal LTS, that is, a LTS accepting any sequence of events. The error state for such testers is unreachable. No scenario can violate the property. We should obviously only produce properties whose tester is not the universal LTS.

In our example, there is no tautology. Five generated properties are shown to the user. Among them, two are worth pointing out:

- $\Box$ *(alarmed* $\rightarrow$ *o* $\neg$ *moving)* states that "when the train is in alarm state, the train controller must stop the train immediately";
- $\Box$ *(alarmed* $\wedge$ $\neg$ *moving* $\rightarrow$ *o* $\neg$ *doors_closed)* states that "when the train is in alarm state and not moving, the train controller must open the doors immediately";

Here again, the user is asked to provide a counterexample scenario when a property is rejected.

3.3.3. Discussion

Our technique generates safety properties on any model that can be decorated with fluents. It might for example be applied to LTS and guarded high-level Message Sequence Charts (see Chapter 4). As opposed to scenarios, state machines and process models are assumed to be complete. All generated properties are therefore expected to hold; user validation is then no longer necessary.

Our property inference procedure might in such cases overwhelm the user with a large number of properties, especially if the number of fluents is high. To avoid this problem, a tool might propose only properties containing specific fluents that were previously selected by the user. The tool could answer user queries about correlations between specific fluents, e.g., *on these scenarios, is there a goal that contains only occurrences of the fluents "moving" and "doors_closed" ?*. This might easily be done during the first step by decorating agent's timelines with fluents selected by the user.

In case of many properties being generated, the ones that are easier to understand might be filtered. One metric for determining whether a property is too hard to understand might be to count the number of different fluents in

it; the tool might then refrain from showing properties with more than *n* fluents.

The effectiveness of the technique might then depend on specific choices of fluents. What makes a good choice of fluents and how to identify these is an interesting open issue worth considering.

3.4 Summary

This chapter introduced two algorithms for decorating each state of a LTS with an invariant over fluents. The first algorithm decorates each state with an invariant consisting of a conjunction of fluents or their negation. The second algorithm decorates them with a Boolean expression capturing the most specific invariant. The invariants generated by the first algorithm are simple and more understandable; they can be used to improve the understandability and the documentation of LTS state machines (as experienced by developers of the LTSA toolset where they have been implemented). The invariants generated by the second algorithm are more complex, but more accurate; they can be used by analysis tools for more efficient analysis.

As a first example of analysis based on invariant generation, this chapter further introduced a new technique for generating goal specification from scenarios. This technique requires state invariants along the timelines of the scenarios provided. This generation is performed by instantiating the decoration algorithm to a MSC timeline instead of an entire LTS.

Chapter 6 will present another type of analysis requiring invariant generation as well, namely, the checking that guards at a decision point in a process model are complete, disjoint and satisfiable (see Section 6.1).

Chapter 4 Modeling Decision-Based Process Models

For effective software support, real-world processes should be captured through some adequate model [Dum05]. Process and workflow modeling languages have therefore flourished, e.g., UML Activity Diagrams [OMG03], BPMN [OMG08], or Little-Jil [Wis00], to cite a few languages in use. Those modeling languages will be reviewed in greater detail and compared in Chapter 8 (Related Work). At this point, we may notice that none of these languages allow process decisions to be captured in a formal way.

The models amenable to formal analysis should obviously be as close as possible to the material provided by process stakeholders. Our recent experience in assembling clinical process fragments supplied by medical staff led us to the observation that such stakeholders naturally think in terms of

- therapy *scenarios* involving interacting agents with multiple exceptions,
- sequencing of *phases* composed of *tasks*,
- *decisions* regulating subsequent flows (possibly nested to form decision trees),
- goals and properties on *state* variables about patients.

Our observations find confirmations in the literature on medical workflows, e.g., [Kai05], [Fox98], [Han06].

As no process language supports those features altogether, we have extended the language of high-level Message Sequence Charts, meeting the two first requirements, with guards in order to meet the two last ones.

This chapter introduces *guarded hMSC* as a process modeling language together with guarded LTS (g-LTS) as an intermediate language. The latter language will be used for a variety of analyses upwards the LTS language used for model-checking and animation. The guarded hMSC language has a formal trace semantics defined in terms of g-LTS, the latter having a trace semantics defined in terms of LTS. Having a formal semantics in terms of LTS allows us to reuse the LTS existing framework and tools, e.g. model-checking techniques implemented in the LTSA toolset [Mag06].

This section is organized as follows. Section 4.1 introduces guarded hMSCs. Section 4.2 introduces events associated with process tasks. Section 4.3 introduces the various types of variables used in guarded hMSCs. Section 4.4 shows how guarded hMSCs may be further specified through an initial context condition that describes the initial values of variables. Section 4.5 introduces the semantics of guarded hMSC in terms of LTS. Section 4.6 shows how tasks can be annotated with useful information such as preconditions and timing aspects.

4.1 Process models as guarded hMSCs

This section introduces the guarded hMSC language. Section 4.1.1 overviews the modeling constructs. Section 4.1.2 describes the graphical syntax.

4.1.1. Modeling constructs

A *guarded hMSC* is a directed graph with different kinds of nodes.

- Task nodes capture process tasks. A task is a unit of work to be achieved by collaboration of the agents involved in the process. A task is seen as a sequence of interactions among agents. Each interaction defines an event. In the meeting scheduler example, the task *Constraint Acquisition* can be described as follows: the

scheduler sends constraint request to every intended participant (*constraint_request*); then every participant sends their constraints to the scheduler (*constraint_sending*).

- Decision nodes capture process decisions. A *decision node* states specific conditions for tasks along outgoing branches to be performed. It is characterized by a set of guards that are Boolean expressions on variables, each associated with a specific branch. Variables used in guards are described in Section 4.2.
- Initial and terminal nodes which represent the start and the end of the process, respectively.

A task may be refined into subtasks and decisions. A task refinement is captured by a finer-grained guarded hMSC or an interaction scenario if it is not further refined.

They are two types of arcs.

- Arcs outgoing from tasks or the initial node are called *continuations;* they prescribe how nodes must be sequentially composed. A task should be totally completed before moving into a subsequent node.
- Arcs outgoing from decisions nodes are guarded transitions; a guard must be evaluated to true for the corresponding arc to be followed.

The meta-model of guarded hMSC is given in Fig. 4-1. For clarity, attributes such as task names and pre-conditions are not shown.

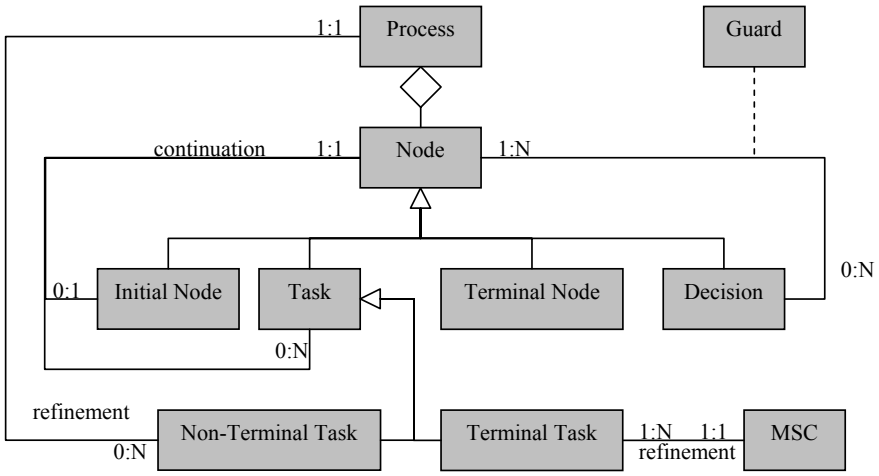


Figure 4-1 Meta-Model of Guarded hMSC

4.1.2. Graphical syntax

A guarded hMSC can be represented graphically or textually. We will mainly refer to the graphical concrete syntax; see [Lam11] for the textual language based on guarded commands used in the tool. Graphically, a task will be represented by a box. A decision will be represented by a diamond; a guard labels each outgoing branch of the decision. In simple cases where there are only two branches, such expression may be moved up inside the decision node with 'yes', "no' labels being attached to the corresponding branch. A refinement of a terminal task is represented by a MSC.

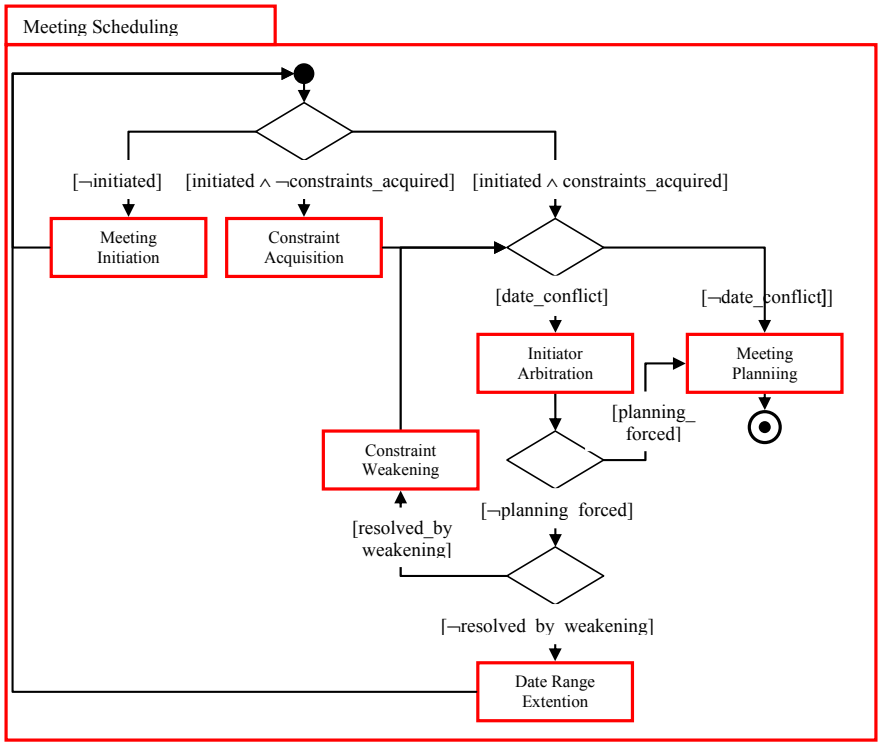


Figure 4-2 Guarded hMSC for the Meeting Scheduler example

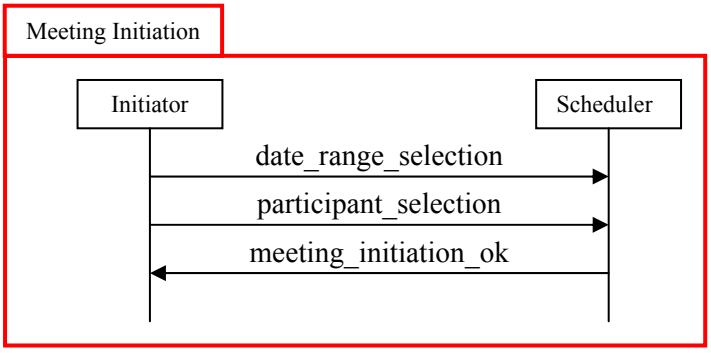


Figure 4-3 MSC for the terminal task *Meeting Initiation*

For example, a guarded hMSC for the Meeting Scheduler [Fea97] example is given in Fig. 4-2. A meeting initiator issues a meeting request, specifying the expected participants and the date range within which the meeting should take place. The scheduler contacts the participants to acquire their

constraints. A date conflict occurs when no date can be found that fits all participant constraints. If no date conflict occurs, the meeting is planned at a date meeting all constraints. Otherwise, the initiator may force the meeting planning with the maximal presence of participants, may extend the date range, or may request some participants to weaken their constraints; in the last two cases, a new scheduling cycle is then required.

All tasks may be refined. As an example, Fig. 4-3 shows a MSC refinement of the task *Meeting Initiation*.

4.2 Task-related events

Event-based analyses should be available even when all tasks have not been completely refined in MSCs. Events associated with unrefined tasks should therefore be added to those of the MSCs.

Process events will be of two types:

- Events contained in the different MSCs. Such events are instantaneous.
- Events that encode task application. Two events T_{start} and T_{end} will be associated with the start and the end of the task T, respectively. The addition of these events provides other benefits:
    - As we will see in the next section, variables are defined in terms of events. They may be defined more accurately if the begin and end of a task are distinguished.
    - Such event can be used to synchronize every process agent so as to prevent the start of a task when its predecessor in

the graph is not yet completed. start/end events are monitored by all agents. Therefore, every agent knows when a task can be started.

In contrast with MSC events, those events may be used to define a task duration; we will come back to this in Section 4.6.2. For this reason, such events will be referred as durational events.

4.3 Process variables

The variables appearing in the decision nodes of a g-hMSC process model prescribe which paths are to be followed in specific process instances. For the process to be realizable, in an event-based model, these variables must be defined entirely in terms of events controlled or monitored by the process [Zav97]. In other words, all state changes of a variable should be captured as events in the model.

A guarded hMSCs may involve two kinds of variables: fluents and tracking variables. We briefly recall fluents in Section 4.3.1. Tracking variables are detailed in Section 4.3.2.

4.3.1. Fluents

A fluent *Fl* is a Boolean variable 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. A fluent definition takes the form:

fluent Fl = < $Init_{Fl}$, $Term_{Fl}$ > initially $Initially_{Fl}$

As an example, the fluent *constraints_acquired* can be defined as follows:

fluent *constraints_acquired* =

<{*Constraint Acquisition*$_{end}$}, {*DateRange Extension*$_{end}$}>

initially *false*

As seen in Section 2.3, a fluent may be represented by a fluent automaton.

4.3.2. Tracking variables

In general, capturing every state change of a variable as an event in the model can make the model complex, unreadable and unmanageable. This is particularly the case in frequent situations where the process depends on environment quantities that are not directly observable by the agents involved in the process.

Tracking variables are variables intended to reflect such quantities as accurately as possible through some observable counterpart. The decisions in a process will be based on current values of tracking variables rather than their actual, unobservable counterpart. To remain accurate in spite of being approximations of the corresponding environment quantity, the values of such variables need to be updated periodically by performing measures (see Fig. 4-4).

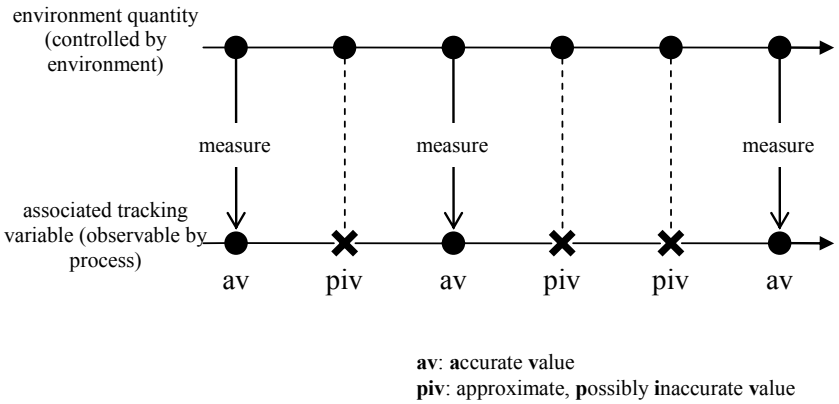


Figure 4-4 Behavior of an unobservable environment quantity and its tracking variable

In presence of tracking variables, specifying every possible result of measures as explicit events can make the process model unnecessarily complex as the next example will suggest. Consider a medical process that

tracks three parameters of the patient: the level of platelets, the level of white blood cells, and the level of red blood cells. These parameters are measured during a blood test. For simplicity, consider that the value of these parameters can be *low* or *normal*. Fig. 4.5 shows the decomposition of a blood test until all possible measure results are made explicit as events. *Blood Test* is decomposed into tasks *Blood Extraction* and *Blood Analysis*. The *Blood Analysis* task must be refined to specify that each parameter may be measured as *low* or *normal*. *Blood Analysis* is thereby refined in six MSCs composed of only one event that captures every possible measure result. For example, *low_platelet_measure* is an event controlled by the blood analyst that expresses that the result of the platelet level measure is *low*. In this example, the tracking variable *platelet_low* can be defined by a fluent which gets true after *low_platelet_measure* and gets false after *normal_platelet_measure*.

As we can see, the *Blood Analysis* refinement is fairly complex. It just does not make sense to enumerate all possible combinations of measure results. Think about a blood test tracking dozen of parameters, as it is the case in real situations.

Instead of explicitly enumerating all possible measure results, we will therefore specify them implicitly by declaratively describing tracking variables through events that periodically measure the corresponding environment variable. For example, *platelet_low* can be defined as follows,

trackingVar platelet_low = {Blood Analysis$_{end}$}.

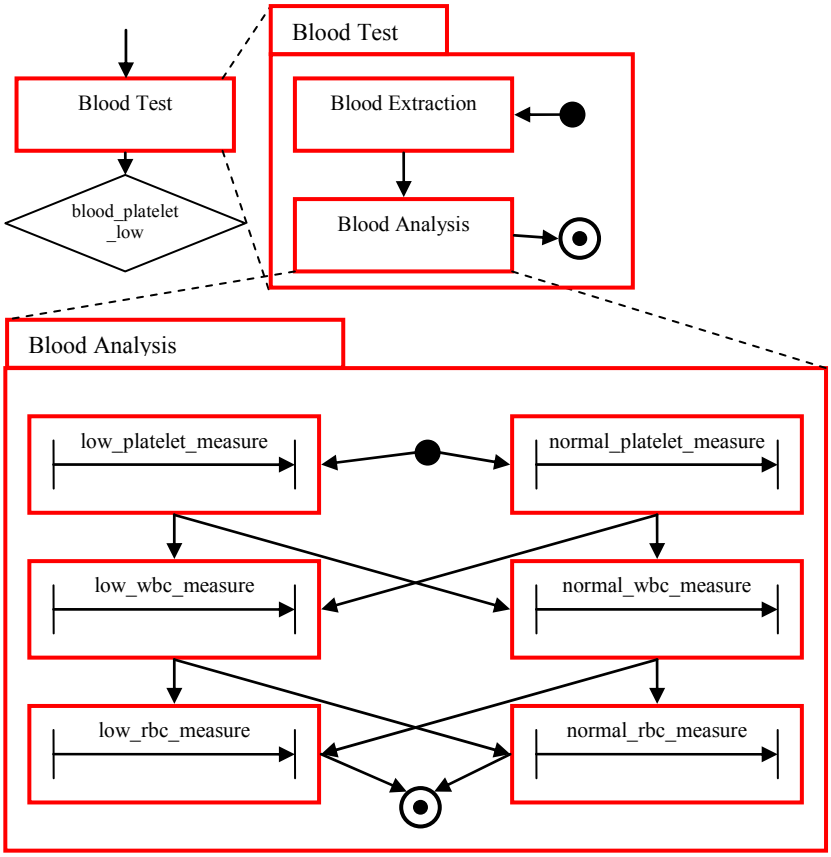


Figure 4-5 Blood Test refinement making all possible measure results explicit

This variable expresses that at the end of a blood analysis, the variable will receive the current, *accurate* value of the platelet level. The tracking variable *platelet_low* then will not change between two blood analyses, as the platelet level can only be monitored through a blood analysis. In the example of Fig. 4-4, if the two other variables are also specified like that, the refinement of *Blood Analysis* can be dropped as it can be automatically generated from tracking variable definitions.

More generally, a *tracking variable* is defined by the set of events measuring the value of its environment counterpart:

trackingVar *tV* = {*MeasureEvent*}

In contrast with fluents, such measure events do not define the value of the tracking variable – only the fact that the value of the tracking variable has been updated and is accurate as it equals the current value of the corresponding environment variable.

An initial value of the tracking variable may be given (see Section 4.3).

Semantically, the behavior of a tracking variable may be described by a "non-deterministic" fluent whose initial and terminal events are the measure events of the tracking variable. From this fluent, a fluent automaton may be generated (see Fig. 4-6). This automaton can be used to model-check properties defined over fluents and tracking variables. The technique to model-check Guarded LTS is detailed in [Lam11].

As the above example may suggest, tracking variables are very common in the medical domain. Such variables may also be present in other application domains. For example, the following tracking variable is worth considering for meeting scheduling:

trackingVar date_conflict =

{Constraint Acquisition$_{end}$, Constraint Weakening$_{end}$}

This variable declaration states that we know whether there is an actual conflict whenever meeting constraints are acquired or weakened.

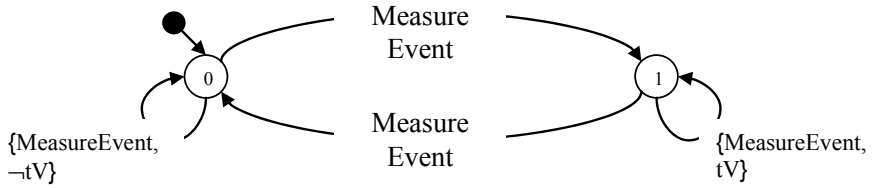


Figure 4-6 Example of tracking variable automaton

4.4 Initial context conditions

A flexible process model should make it possible to capture a process where different instances may follow different paths depending on different initial conditions.

In our running example, the model should cover different paths dependent on user's specifics. Different paths in the model will be followed dependent on whether a meeting was initiated before, the constraints were acquired before, etc.

In the definition of fluents used in a guarded hMSC, we will just omit the initial value $Initially_{Fl}$ for appropriate fluents in order to express that they may be initially *true* for some instances and *false* for others. Initial values are thus defined at instance level, not at class level.

For example, the fluent *meeting_initiated* hereafter specifies that some meetings might be already initiated in the first place while others are not:

$$fluent\ initiated = \langle \{Meeting\ Initiation_{end}\}, \{\} \rangle$$

When initial values are left unspecified for such variables, we may want to state that certain combinations of initial values are to be ruled out in view of properties known from a companion goal model [Lam09].

For example, assuming that the initial values of fluents *initiated* and *constraint_acquired* were not specified, we know anyway that the constraints of the participants cannot be acquired when a meeting is not initiated. Any initial state should meet this property.

To support this, a guarded hMSC may be annotated with an *initial context condition* C_0 constraining the acceptable initial values of variables. In our example, the initial condition might be:

$$constraints\_acquired \supset initiated,$$

meaning that at the initial state, the constraints of a meeting can have been acquired only if the meeting has already been initiated. Initial values of variables are no longer needed then.

4.5 From guarded hMSCs to guarded LTS

This section introduces guarded LTS (g-LTS) as an intermediate formalism between guarded hMSCs and LTS. Roughly, a g-LTS is a transition system with transitions labeled by guards or events.

g-LTS provide a convenient milestone on the way from a guarded hMSC to the corresponding LTS, in particular, for determining the set of traces accepted by the guarded hMSC. As a structured form of LTS, a g-LTS representation avoids state explosion. It is easier to understand and facilitates code generation. Moreover, interesting analyses may be performed at g-LTS level, see Chapter 5.

4.5.1. Guarded LTS

A *guarded LTS* (g-LTS) is defined by a structure (Q, Σ, Φ, Θ, δ, q_0, C_0) where

- Q is a finite set of states,
- Σ is a set of event labels,
- Φ is a set of fluents defined on Σ,
- Θ is a set of tracking variables defined on Σ,
- $\delta \subseteq Q \times (\Sigma \cup \{\tau\} \cup 2^{\Phi+\Theta}) \times Q$ is a labeled transition relation,
- q_0 is the initial state, and
- C_0 is an initial condition playing the same role as in guarded hMSCs.

The label τ is used to denote an internal event that cannot be observed by the environment of a LTS.

Fig. 4-7 shows a g-LTS derived from the guarded hMSC from Fig. 4-2 and Fig. 4-3. In a g-LTS, transitions are labeled either by a guard or by an event. A *guard* is a truth assignment on literals. Intuitively, the guard must be evaluated to *true* for its transition to be activated.

For readability of figures, a set of guarded transitions between the same source and target states will be represented by *{[guard]}*, where *guard* is the disjunction of guards on those transitions.

For example, the transition labeled *{[¬initiated]}* from state 0 to state 1 in Fig. 4-7 actually corresponds to a set of guarded transitions containing, for example, the transition labeled by

[¬initiated ∧ constraint_acquired ∧ date_conflict ∧ planning_forced ∧ resolved_by_weakening].

The g-LTS events are those coming from MSC tasks, e.g., *participant_selection* in Fig. 4-3, or durational events capturing the *start* and *end* of tasks, e.g., *Initiator Arbitration*$_{start}$.

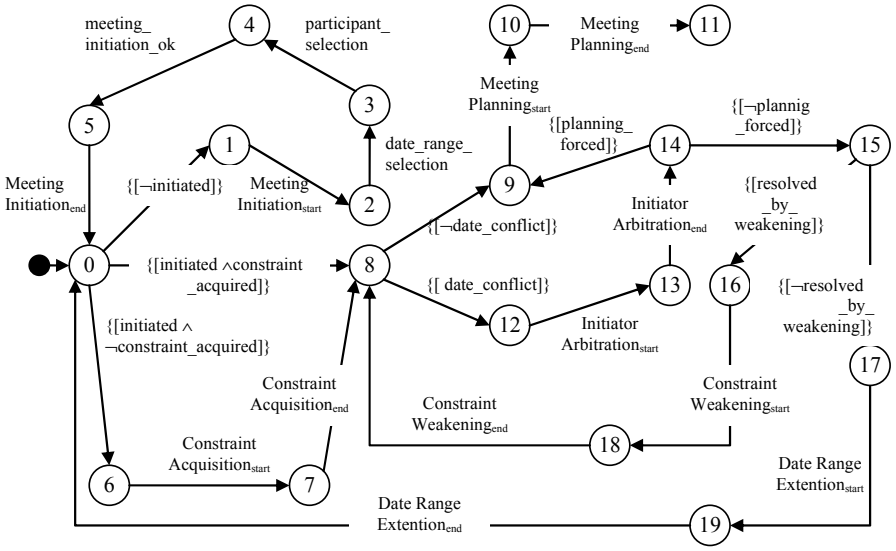


Figure 4-7 g-LTS corresponding to Guarded hMSC of Fig. 4-2 and Fig. 4-3

A guarded hMSC can be rewritten as a g-LTS having the same traces. The rewriting algorithm is explained in detail in [Lam11]. It extends [Uch03] to take a guarded hMSC as input and a g-LTS for the global process as output. The latter abstracts from the agents and captures the set of global behaviors covered by the guarded hMSC.

4.5.2. Trace semantics of g-LTS

The semantics of g-LTS is defined in terms of event traces involving no guards at all.

Let G denote the g-LTS $(Q, \Sigma, \Theta, \Phi, \delta, q_0, C_0)$. An *execution* of G from q_0 is a pair $(Init, <l_0,\ldots>)$ where

- *Init* is an initial variable assignment, mapping every variable to *true* or *false*,
- $<l_0,\ldots>$ is a sequence of labels $l_i \in \Sigma \cup \{\tau\} \cup 2^{\Phi+\Theta}$, some being events and others being guards.

Such execution is accepted by G from q_0 iff the following *acceptance conditions* are met for every i:

- *trace inclusion*: $\exists\, q_{i+1} \in Q$ s.t. $(q_i, l_i, q_{i+1}) \in \delta$
- *admissible start*: $Init \models C_0$
- *guard satisfaction*: $S_i \models l_i$ if $l_i \in 2^{\Phi}$,

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

The first condition states that the label sequence is accepted by the automaton. The second condition states that the initial variable assignment must meet the initial condition C_0. The third condition ensures that all guards are met along the sequence.

An *event trace* of G from q_0 with respect to *Init* is an 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 second condition.

[Lam11] provides an algorithm for building a LTS containing every event trace accepted by a g-LTS. The trace-equivalent LTS is a parallel composition of LTS ensuring the various acceptance conditions. The algorithm given in [Lam11] does not deal with tracking variables but only with fluents. This algorithm can easily be extended to tracking variables by specifying them as non-deterministic fluents.; a tracking variable may be encoded with a fluent whose initiating and terminating events are its measure events.

Having a LTS semantics allows us to reuse the existing LTS framework, e.g., the technique and the LTSA toolset for checking a LTS against properties or animating it.

4.6 Additional constructs

Tasks in a guarded hMSC may be annotated with information required by specific analyses such as the task precondition, its duration, cost, resource needed, etc. Such annotations are optional; they are useful when we want to check corresponding constraints on the model that involve them.

4.6.1. Task preconditions

For some analyses it may be useful to annotate a task with its precondition. A task *pre-condition* 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. Pre-conditions were originally introduced by Turing [Tur49] and are abstractions used in a lot of languages.

In our framework, preconditions are Boolean expressions on fluents and tracking variables. For example, the precondition of task *Meeting Planning* is

$\neg$ *date_conflict*.

Post-conditions are not introduced as they are already captured implicitly in the initiating and terminating events of fluent definitions. For example, consider the fluent *constraints_acquired* defined as follows

fluent constraints_acquired =

$<\{Constraint\ Acquisition_{end}\}, \{DateRange\ Extension_{end}\}>$

From this definition, we can derive the post-condition of task *Constraint Acquisition* as the end of the task is referred in the initiating event. Its postcondition is thus:

constraints_acquired.

We can do the same with *DateRange Extension*. As the end of this task is referred in the terminating event, its post-condition is

$\neg$ *constraints_acquired*.

These post-conditions should be extended if initiating or terminating events of other fluents refer to the end of those tasks.

4.6.2. Time-related constructs

Processes in certain domains can be time-critical. The modeling language should make it possible to annotate tasks with timing information so that we can check whether required time constraints are met by the model.

In our modeling framework, a *duration interval* can be associated with any task not refined further in another g-hMSC. It captures the minimum and maximum time taken by a task (minimum and maximum time should belong to the set of naturals N^+).

Table 4.1. Task durations for meeting scheduling (in days)

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

Table 4.1 illustrates durations of unrefined tasks involved in meeting scheduling.

In the following chapters, we will use the following function for checking that time constraints are met in the process model:

$$duration : \Sigma \rightarrow P(N^{+})$$

As MSC events are instantaneous, only durational events will be associated with a positive duration. The *duration* of a durational event corresponding to the end of a task (i.e. $task_{end}$) is the set of integer belonging to [min, max] where *min* and *max* are the minimum and maximum time taken by a task. The duration of the other events is the set containing only the element "0".

4.6.3. Other non-functional constructs

Tasks may be annotated by further information referred to in non-functional constraints about the process. Such information may refer to resources used, doses, costs, etc.

For example, a task in the medical domain can be annotated by some dose to be delivered in the task (drug dose, radiotherapy dose, etc.). For example, the task associated with a radiotherapy session might be annotated with a dose 1,8 Gray (Gy).

Such information can then be used to detect violations of non-functional requirements on the process (overdose, underdose).

4.7 Summary

This section introduced, guarded hMSCs to model multi-agent processes involving decisions. To allow end-users to understand process models, we restrict the input language to a very simple one. A flowchart-style formalism allows analysts and stakeholders to model decomposable tasks, interaction scenarios and decisions. Although graphical, the language has a formal semantics that can be used by model analysis tools, as will be seen in Chapter 5 and 6.

Tasks are captured in an event-based style as they are seen as sequence of interaction events among agents. On the other hand, decisions and preconditions are captured in a state-based style as they capture conditions on the current state of the process. Combining these paradigms in a single formalism provides multiple benefits. Such combination allows more flexibility to stakeholders for describing their process. Tools can analyze guarded hMSCs by reasoning about system histories, in terms of event traces, or in terms of state traces.

Modeling decisions formally is a key feature of our language; we know of no other high-level process language supporting them. Such decisions depend on variables defined on process events, including tracking variables that mimic environment state variables whose values are periodically accurate. Non-functional requirements are supported through additional task annotations.

The guarded hMSC language has limitations. Other high-level constructs are missing. Currently, variables in guarded hMSC are only defined on Boolean domains; variables on bigger domains such as integers would be useful for defining finite loops, which are currently not supported. Exception handling mechanisms [Ler10] are not supported. This would be very useful especially in medical domain where exceptions might occur at anytime. Such additional

constructs could increase the expressiveness of the process language at the price of simplicity and understandability by domain experts.

Chapter 5 Generating Abstract Decorations on Guarded Transition Systems

The previous chapter introduced guarded hMSC as a process modeling language. Thanks to the formal semantics of the language, different tool-supported analyses can be performed on the models. Examples of interesting analyses include:

- the verification of timing properties,
- the verification that the guards at a decision node are complete and disjoint,
- specific checks for particular application domains – such as the verification of dose requirements in medical processes.

The thesis presents a *generic* approach allowing analysts to easily customize tool-supported analyses for detecting specific types of flaws in guarded hMSCs.

Such analyses proceed in three steps:

1. A g-LTS is first generated from the guarded hMSC [Lam11].
2. Each state of this g-LTS is decorated with computed quantities that are meaningful to the specific check the user wants to perform.
3. The computed quantities are then used to perform the analysis.

This chapter focuses on the second step. Rather that providing a different algorithm for each different type of decoration (precondition, timing, cost, dose, etc.), we propose a fully generic algorithm for decorating g-LTS nodes with placeholders.

A *placeholder* is an abstract quantity that should be instantiated for the specific analysis considered; examples of placeholders are the time elapsed

from the initial state or the dose of radiation received by a patient during treatment.

For each type of check, we need in Step 2 to instantiate the placeholder and the rule for propagating placeholders through a single state transition. Chapter 6 will illustrate Step 3 on a variety of checks.

Our generic algorithm is aimed at being as easy to instantiate as possible. To instantiate the decoration algorithm, the analyst will not be required to reason about guards within the g-LTS. Moreover, as will be seen, we do not need to specify the lattice structure involved in a specific instantiation; and no instantiation-specific proofs are required.

The algorithm in this chapter significantly generalizes the decoration algorithms in Chapter 3 in order to achieve the following objectives:

- support tracking variables, in addition to fluents, in the generation of state invariants;
- generate other types of decorations, in addition to state invariants, to enable reasoning about timing, resources, doses, and so forth.

Section 5.1 defines the concept of placeholder and discusses the propagation of its values along a single execution. Section 5.2 describes how Boolean expressions on fluents and tracking variables can decorate a g-LTS node on a single execution. Section 5.3 describes the values of placeholders and Boolean expressions for a node pertaining to multiple executions. Section 5.4 presents our generic decoration algorithm. Section 5.5 discusses the instantiation process required to generate concrete decorations for use in a variety of analyses. Section 5.6 illustrates the algorithm on a simple example. Section 5.7 provides some further discussion on the proposed approach.

5.1 Placeholder value for a node on a single execution

A *placeholder* is an abstract variable that should be instantiated for a specific analysis it is associated with. The values of a placeholder belong to a domain denoted *PlDomain*. In this domain, there is an element p_0 which is the value of the placeholder after an empty execution.

A *propagate* function has to be provided that specifies how placeholder values are to be propagated through a single state transition.

Note that this constraints placeholders; it requires that the value of a placeholder at some state only depends on its value at the previous state and the incoming event. Placeholders may thus not depend on subsequent events; e.g., the variable capturing the remaining time before reaching a task *T* may not be modeled as a placeholder.

The *propagation* function has the following signature:

$$propagate : PlDomain \times \Sigma \rightarrow P(PlDomain).$$

A placeholder can have multiple propagation results. This is needed to support placeholders that cannot be deterministically determined after the occurrence of an event. For example, the duration of a task may vary depending on agents performing it; the duration of a task is then captured by an interval.

Along a single execution, the propagation of a placeholder *p* through an event *e* should be an element of the set *propagate(p, e)*.

Example: Instantiation for timing

To capture elapsing time through tasks having some minimum/maximum duration, PlDomain will be the set N^+ of naturals.

- The value of the placeholder after an empty execution will be

  $$p_0 = 0.$$

- The function *propagate* will be defined as follows:

  propagate(pl, e) = pl (+) *duration(e),*

  where the "(+)" operator "adds" a duration interval to a time point according to the following definition:

  $(+) : \ N^+ \times P(N^+) \rightarrow P(N^+)$

  $t\ (+)\ [min, max] = [t + min, t + max]$

(End of example)

Let us denote by *PlValue* the value of the placeholder after a g-LTS execution.: Let σ denote a sequence of labels of the g-LTS and <> denote the empty sequence. A placeholder value can be computed recursively:

- At the initial state, its value is equal to p_0.

  $PlValue(Init, <>) = p_0$

- The value does not change through a guard.

  $PlValue\ (Init, <\sigma, label>) = PlValue\ (Init, \sigma)$

  if *label* is a guard

- Through an event, the placeholder value is updated using its *propagate* function.

  $PlValue\ (Init, <\sigma, label>) = p$

  $where\ p \in propagate(PlValue(Init, \sigma), label)$

  if *label* is an event

Example: Instantiation for timing

Let be *x* the value of *PlValue* after a g-LTS execution *(Init,* σ*)*. Consider our Meeting Scheduler running example (see Fig. 4-7). Suppose that the event $Constraint\_Acquisition_{end}$ is firable after this execution. After this event along that same execution, we have

$PlValue\ (Init, <\sigma, Constraint\_Acquisition_{end}>) \in (x\ (+)\ [1,7]),$

since the duration of the task *Constraint_ Acquisition* can last between 1 and 7 days (see Table 4.1).
(End Of example)

5.2 Assigning values to fluents and tracking variables along a single execution

To propagate placeholder values correctly through guards, we need to know at each state the values of all fluents and tracking variables; a placeholder value should not be propagated through a guard if the current variable assignment does not satisfy the guard.

Section 3.2.1 introduced *TraceCond(σ)* as a condition characterizing each LTS state along a LTS trace σ in terms of the values of all fluents involved.

In a similar way, we now define

ExecGCond(Init, <$l_1,\ldots,l_n$>)

as a condition characterizing each g-LTS state along a g-LTS execution in terms of the values of all fluents and tracking variables.

This condition will be a conjunction over fluents and tracking variables, each having a single value - either true or false. We will represent the set of all such possible conjunctions by $2^{\Phi+\Theta}$.

The condition *ExecGCond(Init,σ)* can be computed as follows:

- At the initial state, the value of *ExecGCond* is the initial variable assignment *Init*.
- Through a guard, if the assignment of variables satisfies the guard, the execution may continue through the guard and the variable assignment remains the same.
- Through an event *e*, the assignment is updated by making *true* (resp. *false*) all fluents whose initiating (resp. terminating) events contains

e, and by making non-deterministically *true* or *false* all tracking variables whose measure events contains *e*.

Example

Let us consider a g-LTS where one fluent and one tracking variable are defined:

fluent $F = <\{e\},\{f\}>$

$tV\ T = \{e\}$

Consider a g-LTS execution with $\neg F \wedge \neg T$ as initial variable assignment. For this execution, the value of *ExecGCond* after an empty execution is simply the initial variable assignment:

$ExecGCond(Init, \{\}) = \neg F \wedge \neg T$

Suppose that the event *e* can be fired from the initial state. From the initial variable assignment *Init,* after the event *e*, the fluent *F* is *true* as *e* is an initiating event of *F*; the tracking variable T may be *true* or *false* as *e* is a measure event of T as well. Therefore, $ExecGCond(Init, \{e\})$ may have two possible values:

- $F \wedge T$
- $F \wedge \neg T$

Said otherwise, $ExecGCond(Init, \{e\})$ should satisfy *F* and should be a complete assignment of variables.

(End Of example)

Let us define the new value of *ExecGCond* after an event *e* more formally. In view of event *e*, $ExecGCond(Init, <\sigma,e>)$ may be computed in three steps as follows:

1. We remove from $ExecGCond(Init, \sigma)$ all fluent literals for which *e* is an initiating or terminating event and all tracking variables for which *e* is a measure event.

2. For each fluent *Fl* for which e is an initiating event, we add *Fl* as conjunct; and for each fluent *Fl* for which *e* is a terminating event, we add $\neg Fl$ as conjunct.
3. For each tracking variable *tV* for which e is a measure event, we add non-deterministically *tV* or $\neg tV$ as conjunct.

To support, the first two steps, we adapt the function *Effect* introduced in Section 3.2.1; *Effect* computes the effect of an event of a Boolean expression of fluents. This function is adapted as follows in order to handle tracking variables in addition to fluents.

$$Effect(B, e) = (B[\setminus(\{v_i\} \cup \{v_t\} \cup \{tV\})]) \wedge (\bigwedge_i v_i) \wedge (\bigwedge_t \neg v_t)$$

where

$\{v_i\}$ is the set of all fluents v_i such that e is among their initiating events,

$\{v_t\}$ is the set of all fluents v_t such that e is among their terminating events,

{tV} is the set of all tracking variables tV such that e is among their measure events.

B[\v] is the Boolean formula B where the variable v has been dropped (see Section 3.2.1)

In addition to changing the value of fluents according to fluent definitions, the function *Effect* also drops all tracking variables for which the event is a measure event.

Example

In the previous example,

$$Effect(\neg F \wedge \neg T, e) = (\neg F \wedge \neg T)\ [\setminus\{F,T\}] \wedge F = F$$

(End Of example)

Using this adapted *Effect* function, we can define *ExecGCond* formally as follows:

ExecGCond(Init, <>) = Init.

ExecGCond (Init, <σ, l>) = ExecGCond(Init, σ) if *l* is a guard

ExecGCond (Init, <σ, l>) = x

where x |= Effect(ExecGCond(Init, σ), l) and $x \in 2^{\Phi+\Theta}$

if *l* is an event

The latter formula means that *ExecGCond (Init, <σ,e>)* should satisfy *Effect(ExecGCond(Init, σ), e)* and should be a complete assignment of variables.

Example

By instantiating the last formula, we have,

ExecGCond(¬ F ∧ ¬ T, <e>) = x where *x |=F* and $x \in 2^{\Phi+\Theta}$.

In other words, *ExecGCond(¬ F ∧ ¬ T, <e>)* should satisfy *F* and should be a complete assignment of variables. Two values are possible:

- $F \wedge T$
- $F \wedge \neg T$

(End of example)

5.3 Generating decorations of a g-LTS node

Different g-LTS executions reaching the same node may have different variable assignments and different placeholder values. In order to correctly propagate placeholders through guards, we need to keep track of the *mapping between the variable assignments and their possible associated placeholder values.*

To visualize this, consider Fig. 5-1(a). Suppose that a state is decorated with an interval, meaning that in this state the placeholder may have as value any element within this interval. We cannot compute the decoration after a guard without knowing the relation between the values of the variable *A* and the interval valid for such values. In Fig 5-1(b), we have this information; for

example, when the variable A is true, the placeholder may belong to the interval [3, 4]; in this case, we can compute the decoration of the target states.

Note that the placeholder domain might be infinite. This raises an issue of termination of the algorithm. To address this, in cases where the placeholder domain is infinite, we will ask the user to provide a finite subdomain of *PlDomain*, called *PlFiniteDomain*. If there exist g-LTS executions yielding placeholder values that do not belong to *PlFiniteDomain*, the algorithm will use the element *Out_Of_Bounds* to represent such executions.

Our algorithm will therefore decorate each g-LTS state with the following function:

decostate: $2^{\Phi+\Theta} \rightarrow P\ (FinitePlDomain \cup \{Out\_Of\_Bounds\})$.

This function captures all possible placeholder values in the finite domain for a given assignment of fluent and tracking variables. If the placeholder can have values outside the finite domain, the element *Out_Of_Bounds* will be used to represent such values.

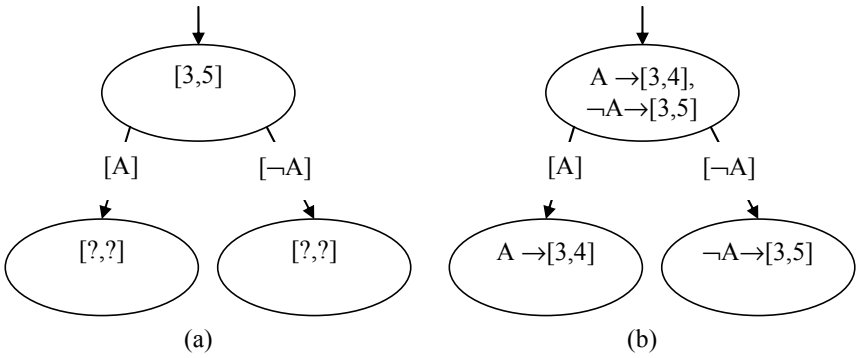


Figure 5-1 Propagating decorations through guards

The *decostate* function forms a lattice $(\leq, \wedge, \vee, \bot, \mathrm{T})$ defined as follows:

$f \leq g \quad$ *if* $dom_f \subseteq dom_g$ *and* $\forall x \in dom_f, f(x) \subseteq g(x)$

$\bot = \varnothing$

$T = \{(x \mapsto (FinitePlDomain \cup \{Out\_Of\_Bounds\})) \mid x \in 2^{\Phi \cup \Theta}\}$

$f \vee g = \{(x \mapsto f(x) \cup g(x)) \mid x \in dom_f \cap dom_g\}$

$\cup \quad \{(x \mapsto f(x)) \mid x \in dom_f \setminus dom_g\}$

$\cup \quad \{(x \mapsto g(x)) \mid x \in dom_g \setminus dom_f\}$

$f \wedge g = \{(x \mapsto f(x) \cap g(x))) \mid x \in dom_f \cap dom_g\}$

The partial order in this lattice is defined by two conditions:

- the inclusion of function domains; it corresponds to the possible variable assignments valid at a state;
- the inclusion of placeholder sets for each variable assignment.

The bottom element in the lattice is the empty function, corresponding to an unreachable state. The top element corresponds to a state where all assignments are possible, the disjunction of elements of $dom_{decostate}$ being *true*; the values for all assignments cover all possible values of the codomain.

The supremum of two *decostate* functions f and g yields a function $f \vee g$ whose domain is the union of the domain of f and g. On domain elements where both f and g are defined, the value of $f \vee g$ is defined as the union of their respective value. On elements where only f (resp. g.) is defined, $f \vee g$ is simply reduced to f (resp. g).

The infimum operator is its natural counterpart.

As in Chapter 3, we will compute the *most specific mappings*, that is, the most specific range of values that characterize the mapping for any execution leading to that state.

The use of a finite domain requires an additional hypothesis on the *propagate* function defined in Section 5.2. This hypothesis is required in order to ensure that the computed decorations are correct, namely,

For any $Pl \in PlDomain \setminus PlFiniteDomain$, and $e \in \Sigma$

$propagate(Pl,e) \cap PlFiniteDomain = \varnothing$.

This hypothesis states that the propagation through an event of an element outside the finite domain remains outside the finite domain. This hypothesis may seem fairly strong; for example, the values of a placeholder representing an unbounded but resettable clock may not been computed using our algorithm.

5.4 The abstract decoration algorithm

The specification of our algorithm to decorate a g-LTS with placeholder values that are valid in the corresponding state is given in Fig. 5-2.

GIVEN *a guarded LTS* $(Q, \Sigma, \Phi, \Theta, \delta, q_0, C_0)$
a placeholder domain PlDomain
a finite placeholder domain $PlFiniteDomain \subseteq PlDomain$
a propagate function $PlDomain \times \Sigma \rightarrow P(PlDomain)$ *such that*
for any $Pl \in PlDomain \setminus PlFiniteDomain$ *and* $e \in \Sigma$:
$propagate(Pl,e) \cap PlFiniteDomain = \varnothing$.
the value of the placeholder after an empty execution $p_0 \in PlFiniteDomain$

FIND *a node decoration function*
$deco: Q \rightarrow (2^{\Phi+\Theta} \rightarrow P(FinitePlDomain \cup \{Out\_Of\_Bonds\}))$
such that for any state q:
(INV1) *for any g-LTS execution* γ *reaching q,*
if $PlValue(\gamma) \in PlFiniteDomain$ *then* $PlValue(\gamma) \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$
(INV2) *for any g-LTS execution* γ *reaching q,*
if $PlValue(\gamma) \notin PlFiniteDomain$ *then* $Out\_Of\_Bounds \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$
(MSM1) *for any* $Pl \in PlDomain$ *such that* $Pl \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$,
there exists a g-LTS execution γ *reaching q such that:* $PlValue(\gamma) = Pl$
(MSM2) *if* $Out\_Of\_Bounds \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$,
then there exists an execution γ *reaching q such that*
$PlValue(\gamma) \notin PlFiniteDomain$

Figure 5-2 Specification of the g-LTS abstract decoration algorithm

At every state, the algorithm computes placeholder values for each variable assignment. These placeholder values must meet the following conditions:

- Each placeholder value produced by a g-LTS execution reaching state q should belong to the computed decoration of q, i.e. for all executions γ reaching a state q,
    - (INV1) if *PlValue(γ)* belongs to the finite domain of placeholders then it belongs to the codomain of *deco(q;*.
    - (INV2) otherwise *Out_Of_Bonds* belongs to the codomain of *deco(q)*.
- The computed placeholder values for a state q should be produced by at least one execution reaching q, i.e.,
    - (MSM1) Each placeholder value in the codomain of *deco(q)* should be a placeholder value produced by at least one execution reaching q.
    - (MSM2) If *Out_Of_Bounds* belongs to the codomain of *deco(q)*, there exists at least one execution yielding a placeholder value outside the placeholder finite domain.

Similarly to the algorithm in Chapter 3, the algorithm in Fig. 5-3 proceeds by propagation of state decorations to their successors until a fixpoint is reached. In Fig. 5-3, the symbols $\leq$ and $\vee$ correspond to the partial order and the supremum of the lattice introduced in Section 5.3, respectively. Let us explain how this algorithm works.

The initial decoration is defined as follows:

$$\{(x \mapsto \{p_0\}) \mid x \in 2^{\Phi \cup \Theta} \text{ and } x \models C_0\}$$

The domain of this function corresponds to all assignments satisfying the initial condition of the g-LTS. They are all mapped to the initial value of the placeholder p_0.

The decoration of the initial state is then propagated to its successors.

- Through a guard, the decoration function is restricted to the specific Boolean assignment to fluents and tracking variables defined by *label* ("$\triangleleft$" is the domain restriction operator).
- Through an event, the domain of the function is updated as explained in Section 5.2. The range of the function is updated using the procedure *finite_propagate* which defines how a state decoration should be propagated through the given event within the finite domain. If the output of the *propagate* function given by the user is outside the finite domain, the elements outside the finite domain are replaced by the value *Out_Of_Bonds*. The propagation of *Out_Of_Bonds* through an event remains outside the finite domain by our hypothesis on the *propagate* function. All the mappings "assignment-placeholder values" are accumulated through a supremum operator on sets, defined as follows:

$$\bigvee \{d_1, d_2, d_3, \ldots, d_n\} = (((d_1 \vee d_2) \vee d_3) \ldots . d_n)$$

where $\vee$ is the supremum operator of the lattice.

Propagated decorations are thus accumulated through the supremum operator of the lattice.

The algorithm keeps track of the set *ToExpl* of states that were updated and to which the propagation should be applied. It terminates when the set *ToExpl* is empty, that is, when there is no state that must further propagate its value.

The correctness of the algorithm is proved in Annex B.

```
Input: A g-LTS (Q,Σ,Φ,Θ,δ,q0,C0)
       A placeholder domain PlDomain
       A finite placeholder subdomain PlFiniteDomain ⊆ PlDomain
       A propagate function PlDomain x Σ → P(PlDomain)
       The value for the placeholder after an empty path p0 (p0∈ PlFiniteDomain)
Output: deco: Q → (2^(Φ+Θ) → (P(PlFiniteDomain) ∪ {Out_Of_Bonds})))


decorate (…) =
    deco(q0) := { x ↦ {p0} | x ∈ C0};
    forall q ∈ Q \ {q0} do
        deco(q) := ∅;
    ToExpl := {q0};
    while ToExpl ≠ ∅ do
        source := getOne(ToExpl);
        ToExpl := ToExpl \ {source};
        forall (target,label) such that target ∈ δ do
            if (label ∈ 2^(Φ+Θ))
                newVal := {label}◁ deco(source) ;
            else
                newVal := ⋁{(x) ↦ finite_propagate(deco(source)(y),label))
                                        | y ∈ dom_deco(source) and x |= Effect(y,label) and x ∈ 2^(Φ+Θ)};
            if not (newVal ≤ deco(target)) then
                deco(target):= newVal ⋁ deco(target)
                ToExpl := ToExpl ∪ {target};
    return deco;

finite_propagate(set, e) =
    toreturn := ∅;
    forall x ∈ set
        if (x == Out_Of_Bonds) toreturn := toreturn ∪ {Out_Of_Bonds};
        else     val := propagate(x,e);
            if (val \ PlFiniteDomain ≠ ∅)
                val := (val ∩ PlFiniteDomain) ∪ {Out_Of_Bonds};
        toreturn := toreturn ∪ {val};
    return toreturn;
```

Figure 5-3 Generic algorithm for abstract g-LTS decoration

5.5 Instantiating the decoration algorithm to a specific analysis

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

(a) A placeholder domain *PlDomain* is defined so as to meet the desired type of analysis.

(b) An initial placeholder value p_0 is chosen.

(c) A *propagate* function is defined to make it precise how the events transform the value of the placeholder of the corresponding source state through their outgoing transitions.

(d) If PlDomain is not finite, a finite subdomain is defined.

5.6 Example

In this section, we instantiate the algorithm in order to compute at each g-LTS state *the time elapsed from the initial state*. More instantiations will be given in Chapter 6. Next, we show the algorithm instantiation in action on a simple example.

(a) Here, the placeholder will represent the possible time elapsed from the initial state. PlDomain is the set N^+ of naturals.

(b) The *propagate* function is defined as follows:

propagate(pl, e)

return *pl (+) duration(e);*

where the "(+)" operator "adds" a duration interval to a set of points according to the following definition:

$$(+)_{\_}: N^+ \times P(N^+) \rightarrow P(N^+)$$

$$t \,(+)\, [min, max] = [t + min, t + max]$$

(c) The initial time is $p_0=0$

(d) As this domain is infinite, we define PlFiniteDomain by restricting PlDomain to the interval *[0,MAX]* where *MAX* is some upper time bound to be chosen by the user.

Let illustrate the first steps of the instantiated algorithm on the simple example of Fig. 5-4.

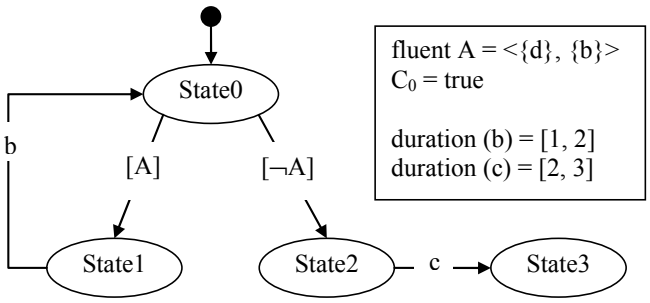


Figure 5-4 Executing the algorithm for decorating g-LTS states with time

(Step 0) We initialize the decorations for each state to ∅ except for the initial state where the decoration is

$deco(State0) = \{A \rightarrow \{0\}, \neg A \rightarrow \{0\}\}$.

The latter is the function where each variable assignment meeting the initial context C_0 is associated with the initial placeholder value.

(Step1) *State0* is in *ToExpl*. We propagate its decoration to all its successor states, i.e., *State1* and *State2*. To compute the decoration of *State1*, since *[A]* is a guard, we restrict the decoration of *State0* to the specific assignment *A*. The decoration of *State1* becomes

$deco(State1) = \{A \rightarrow \{0\}\}$.

Similarly, the decoration of *State2* becomes

$deco(State2) = \{\neg A \rightarrow \{0\}\}$.

State1 and *State2* are added in *ToExpl* as their decoration has changed.

(Step 2) Let us assume that *State1* is chosen in *ToExpl*. Its decoration is propagated to its only successor, *State0*. The event *b* is a terminating event of the fluent *A* and its duration belongs to the interval *[1, 2]*.

Since

$Effect(A, b) = \neg A$,

$finite\_propagate(\{0\}, b) = \{1, 2\}$,

newVal gets the following value:

$newVal = \{\neg A \rightarrow \{1, 2\}\}$.

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

$deco(State0) = \{\neg A \rightarrow \{0\} \cup \{1, 2\}\} \cup \{A \rightarrow \{0\}\}$

$= \{A \rightarrow \{0\}, \neg A \rightarrow \{0, 1, 2\}\}$.

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

(Step 3) Let us assume that *State0* is chosen in *ToExpl*. Its decoration is propagated to *State1* and *State2*. For *State1*, *newVal* has the same value as its decoration; the decoration of *State1* is not updated. For *State2*, *newVal* gets the value:

$newVal = \{\neg A \rightarrow \{0, 1, 2\}\}$.

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

$deco(State2) = \{\neg A \rightarrow \{0, 1, 2\}\}$.

State2 is added to *ToExpl*.

(Step4) *State2* is the only element in *ToExpl* and is chosen. Its decoration is propagated to *State3*. As the event *c* does not belong to initiating and terminating events of *A* and as the duration of *c* belongs to the interval *[2, 3], State3* is decorated with

$deco(State3) = \{\neg A \rightarrow \{2, 3, 4, 5\}\}$.

State3 is added in *ToExpl*. As *State3* was the only element of *ToExpl* and has no successors, the algorithm terminates.

5.7 Discussion

Our generic technique is aimed at allowing analysts to easily create new analyses that provide accurate results. However, such analyses might raise efficiency problem in some cases. The abstract decoration algorithm symbolically explores the entire process state space, captured in a structured

and compact way through a g-LTS. The potential state explosion problem is tackled in various ways.

- The g-hMSC refinement mechanism naturally supports *local* and *incremental* checks. We may thereby perform checks on small g-LTS to reduce the state space size. Such small checks require only the initial context to be computed in terms of fluents and tracking variables at the initial state of the small g-LTS.
- The decoration generation cost may be distributed among *multiple* checks. One decoration computation may be used for several analyses (see Chapter 6).

Decoration generation remains of course subject to exponential blow-up in worst-cases. The implementation of the GISELE tool, supporting the generic algorithm as well as some instantiations, is slightly different from the pseudo-code given in Fig. 5-3 for efficiency reasons. The tool decorates each state with the following function:

decostate: $P(2^{\Phi+\Theta}) \rightarrow P\ (FinitePlDomain \cup \{Out\_Of\_Bonds\})$.

The difference is that the elements of the domain may be any Boolean expressions over variables ($P(2^{\Phi+\Theta})$) rather than complete assignments of variables ($2^{\Phi+\Theta}$). This allows us to have more compact representations for variable assignments having the same placeholder values. Moreover, for that same objective of compactness, the domain of *decostate* uses binary decision diagrams (BDDs); sets of placeholders are represented through sets of intervals.

As proved in Annex B, our algorithm computes the most specific mapping for each state. This mapping might not be always necessary for all analyses. The use of abstract interpretation [Cou77, LeC94] might significantly improve the efficiency of the instantiated algorithms by finding "good" approximations for these mappings. *Abstract interpretation* formalizes the

idea that a formal proof can be done at some level of abstraction where irrelevant details about the semantics and the specification are ignored. This amounts to proving that an *abstract semantics* satisfies an *abstract specification*. Together with the "concrete" domain of placeholders we are considering here, the user should provide an "abstract" domain for the instantiated placeholder with an abstract semantics tailored to the specific analysis we want to perform. Such abstract domain is in general a lattice.

Algorithms similar to the one proposed here could be used to compute abstract decorations on g-LTS states [Cou75a, Cou75b]. However, the use of abstract interpretation techniques requires dedicated proofs for each specific abstraction. In particular, the abstract semantics must be proved to be correct with respect to the concrete one.

5.8 Summary

This chapter presented a generic algorithm for decorating each state of a g-LTS with placeholders. A placeholder is an abstract variable that should be instantiated for a specific analysis it is associated with.

The algorithm was designed to be easy to instantiate. For each type of placeholder, the analyst only needs to instantiate the rule for propagating placeholders through a single transition. The instantiator does not need to reason about guards nor define a lattice structure specific to the instantiation.

This generic algorithm may be used to perform different kinds of analyses on guarded hMSCs. Examples of such analyses are given in the next chapter.

Chapter 6 Decoration-Based Model Analysis

In this chapter, the generic algorithm from Chapter 5 is instantiated to a variety of analyses on process models. Section 6.1 proposes a technique for verifying that guards at a decision node are disjoint, complete and locally satisfiable. Section 6.2 shows how task preconditions can be verified or generated. Section 6.3 presents our technique for detecting potentially inadequate decisions. Section 6.4 describes a technique for detecting violations of time constraints on the process.

6.1 Guard analysis on decision-based process models

The three following guard-related checks are worth considering in process models involving decisions.

- *Guard completeness:* At any process step where a decision node is reached, the guards on the alternative outcomes must cover all possible cases; no guarded branch may be missing. Otherwise a process run might be blocked forever at this node with no guard evaluated to true and, correspondingly, no task being prescribed when the missing guard is true.
- *Guard disjointness:* At any process step where a decision node is reached, two guards on different outcomes may not overlap; they may not be both evaluated to true. We generally want to preclude non-deterministic process decisions where different courses of action are taken in different process runs applied to the same situation. In clinical processes, for example, every patient with the same state must be treated identically.

- *Guard satisfiability at decision point:* At any process step where a decision node is reached, the guards on the outcomes must all be satisfiable. Otherwise the subsequent tasks prescribed along some branch would be unreachable.

Such checks are close in spirit to those supported in [Hei96] for SCR tables. Beyond different formalisms there is a notable difference, however. The checks here must account for the *contextual condition* holding at the point in the process model where the decision node is reached. This contextual condition may be computed by instantiating the decoration algorithm from Chapter 5 as discussed hereafter.

6.1.1. Placeholder instantiation

The computation of state invariants in terms of fluents and tracking variables is a quite simple application of the algorithm in Chapter 5.

- As we only need information about values of fluent and tracking variables, no specific placeholder needs to be defined.
- Each element of $2^{\Phi+\Theta}$ will be mapped to the same element.
- *PlDomain* will be composed of only one element.
- The *propagate* function will be the identity function and the initial decoration is the only element of *PlDomain*.

6.1.2. Using decorations to analyze guards

Once contextual conditions are generated as state invariants, the checks are automated through satisfiability checking.

Consider a set of guards g_i starting from a state q. The contextual condition C can be computed as follows

$$C = \vee \, dom(deco(q))$$

(a) To verify *guard completeness*, we need to check that:

$C \models \vee_i g_i$

In case of incompleteness, our tool returns all variable assignments that are not covered by the guards.

(b) To verify *guard disjointness*, we need to check that:

for every pair g_i, g_j: $g_i \wedge g_j \wedge C \models false$.

In case of overlap, the tool returns all variable assignments that meet several guards.

(c) To verify *guard satisfiability* at a decision node, we need to check that:

for every g_i: $g_i \wedge C$ is satisfiable.

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

6.1.3. Example

Our tool was used to perform those three types of guard analysis discussed on the meeting scheduler running example (see Fig. 4.2).

Let us analyze the first decision node with the following guards:

$[\neg initiated]$,

$[initiated \wedge \neg constraints\_acquired]$,

$[initiated \wedge constraints\_acquired]$.

The tool first computes the contextual condition holding just before the decision node. This condition obtained is

$\neg constraints\_acquired$.

The tool reveals that the three guards are complete and disjoint. However, the last guard is unnecessary as the conjunction of the last guard with the contextual condition is unsatisfiable:

$(initiated \wedge constraints\_acquired) \wedge \neg constraints\_acquired = false$

The problem is easily fixed by removing that last guard.

For easier understanding by stakeholders, the second guard might then be manually simplified to

[initiated].

6.2 Generating task preconditions

Tasks in a guarded hMSC process model can be annotated with preconditions. We might therefore like to check that the preconditions, when given, cannot be violated through paths in the model. Even better, we might like to generate them when they are not given.

6.2.1. Placeholder instantiation

To check or generate task preconditions, we first need to generate state invariants in terms of fluent and tracking variables. The instantiation is therefore similar to the instantiation in Section 6.1. No specific placeholder need to be defined.

6.2.2. Using decorations to check or generate preconditions

To check that the precondition PRE_T of a task T is never violated, we just need to verify the following formula is verified with a SAT solver:

for every *source* node of a T_{start} event,

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

When task preconditions are not provided by the modeler, our tool uses the same decorations to generate them.

For a task T, the tool retrieves all source states of an event T_{start}, and takes the disjunction of their decoration domains. The preconditions inferred in this way might need to be further simplified manually in view of domain properties.

6.2.3. Example

In our running example (Fig. 4-2), the precondition $\neg$ *date_conflict* of the *Meeting Planning* task is checked and found to be incorrect. The reason is that the planning could be enforced even if there is a conflict. To fix the problem, two resolutions can be possible:

- change the precondition of *Meeting Planning*
- remove the possibility for the initiator to force the planning of a meeting.

If the first resolution is chosen, the precondition can be inspired from the one generated by the tool. The precondition given by the tool is:

$$initiated \wedge constraints\_acquired \wedge (planning\_forced \vee \neg date\_conflict)$$

6.3 Checking the adequacy of decisions

As tracking variables capture information about quantities in the process environment, we would like to check that *every time they are used, they accurately reflect their actual counterpart in the environment.*

This is particularly important for tracking variables appearing in decision nodes of a guarded hMSC model. Inaccurate values for tracking variables at decision points may result in *inadequate decisions*, where the outgoing branch taken differs from the one that should be taken with accurate values.

We will consider two different sources of inaccuracy resulting in potentially inadequate decisions:

- some tasks affect the environment quantity without updating the tracking variable accordingly (Section 6.3.1),
- the value of a tracking variable becomes outdated after some time (Section 6.3.2).

6.3.1. Task-dependent inadequacies

Consider the tracking variable *date_conflict* in the process model for meeting scheduling in Fig. 4-2. A value *true* means that every date in the initiator's date range is excluded by at least one participant. This value is expected to be accurate right after all participant constraints have been acquired. It might become inaccurate when the date range is extended; a conflict-free date might now be found in the extended set of dates.

To check that this variable is adequately used, we introduce a fluent that we will call *accuracy meta-fluent*.

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

In our example, the meta-fluent definition is thus:

$$fluent\ date\_conflict\text{-}Accurate = \langle \{Constraint\ Acquisition_{end}\}, \{Date\ Range\ Extension_{end}\} \rangle initially\ false$$

Any tracking variable may have an associated *accuracy meta-fluent*.

- The *initiating events* of this fluent are all events that synchronize the tracking variable with its environment counterpart (e.g., $Constraint\ Acquisition_{end}$ in our example).
- The terminating events are the ones potentially affecting the environment quantity while not appearing among the measure events of the tracking variable (e.g., $Date\ Range\ Extension_{end}$ in our example).
- The initial value of the meta-fluents is *false*.

A *task-dependent inadequate decision* is a decision that relies on incorrect information about the environment due to the occurrence of intermediate, possibly unexpected events.

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

Placeholder instantiation

Here the placeholder corresponds to the possible value of the accuracy meta-fluent.

- The domain of the placeholder is *{true, false}* with $p_0 = false$ as initial value for the placeholder.
- The *propagate* function consists in updating the value of the fluent:

  propagate (pl, event)=

  if *(event* $\in$ $Init_{ac}$*)* ***return*** *{true}*

  else ***if*** *(event* $\in$ $Term_{ac}$*)* ***return*** *{false}*

  else *return {pl}*
- As *PlDomain* is finite, it is equivalent to *PlFiniteDomain*.

Using decorations to check decision inadequacies

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

for any Assign $\in$ *dom(deco(source)):*

false $\notin$ *deco (source)(Assign)*

If the formula is not verified, we know that there is a task sequence reaching the decision node in the model such that the value of the tracking variable at

this node is inaccurate; the decision on which subsequent path to follow is therefore potentially inadequate.

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

Example

The decision node *date_conflict?* in Fig. 4-2 is checked to be adequate using our technique. The decoration of *state 8* in Fig. 4-7 ensures that *date_conflict-Accurate* is *true* at this point. Indeed, new constraints are immediately re-acquired every time the date range is extended (see Fig. 1). The cancer therapy process fragment modeled in Chapter 7 will show an example where an inadequate decision is detected by our technique.

6.3.2. Time-dependent inadequacies

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

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

fluent *tV-Accurate* = <Init_{Ac}, Term_{Ac}> **duration** Dur_{Ac}

This fluent holds iff

- an initiating event has occurred and, since then,
- no terminating event has occurred and the elapsed time is less than Dur_{Ac} (in discrete time units).

A *time-dependent inadequate decision* is a decision that relies on out-of-date information about the environment.

Placeholder instantiation

For checking time-dependent decision inadequacies, the placeholder values will belong to the set of naturals in the interval [0, Dur_{Ac}]. The placeholder captures the time remaining before the value of the tracking variable becomes outdated. Beyond that time, the accuracy meta-fluent will necessarily be *false* which means that the associated tracking variable may no longer accurately reflect its environment counterpart. When the accuracy meta-fluent is known to be *false* (e.g., immediately after a terminating event), the value of the placeholder is considered to be *0*.

As PlDomain is a finite domain, we have:

PlDomain= PlFiniteDomain.

The initial value of the placeholder is 0.

Given the accuracy meta-fluent

tV-Accurate = <$Init_{Ac}$, $Term_{Ac}$> **duration** Dur_{Ac},

the *propagate* function is defined as follows:

propagate(pl, e):

if $label \in Init_{Ac}$ ***return*** $\{Dur_{Ac}\}$

else ***if*** $label \in Term_{Ac}$ ***return*** $\{0\}$

else return $pl \ (-) \ duration(event)$.

where the "$(-)$" operator "removes" a duration interval to a time according to the following definition:

$$(-): \ N^+ \times P(N^+) \rightarrow P(N^+)$$

$$T \ (-) \ [min, max] = \{x \mid x \in N^+, T - max \leq x \leq T - min\}$$

The *propagate* function updates the value of the placeholder as follows:

- if the event is among the initiating events of the accuracy meta-fluent, the placeholder is reset;
- if the event is among the terminating events of the accuracy meta-fluent, the placeholder is set to zero;
- otherwise, the placeholder is decremented by the duration of the task given by the label of the source state without going beyond the lower bound.

Using decorations to check inadequacies

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

$$\textit{for any } Assign \in dom_{deco(source)}\textit{:}$$

$$0 \notin deco(source)\,(Assign).$$

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

Example

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

fluent date_conflict-*Accurate* =

$<\{\text{ConstraintAcquisition}_{end}\}, \{\text{DateRangeExtension}_{end}\}>$

duration 15

Our tool finds that the decoration of state 8 in Fig. 4-7 contains the following mapping:

(date_conflict = false, resolve_by_weakening = true,...) |→ {0,...})

This means that there exists a process path where a date conflict seems resolved after constraint weakening but with participant constraints possibly outdated. This might happen, for example, if the weakening task is performed twice in a row by the initiator, each time taking the maximum duration of 7 days (see Table 4-1). Some acquired participant constraints are then inadequately considered accurate for more than 15 days while the meeting is being planned.

To resolve this problem, the model might be changed so that the meeting could be cancelled under certain conditions, or date range extension could be the only possibility for the second round, or participants could be asked to confirm and/or update their constraints after 15 days, etc.

6.4 Analyzing timing requirements

For time-critical processes such as those found in the medical domain, we would like to *infer* or *verify* critical timing properties.

- We might want to know how long a process can take, in best or worst situations for example.
- We might want to impose timing requirements on the process (or task), and check that these requirements are always met in the model.

Complex time constraints generally require the use of dedicated tools such as temporal model checkers [Lar97]. This section describes a more lightweight, decoration-based technique for inferring or checking simple timing properties.

6.4.1. Placeholder instantiation

Let us recall the placeholder instantiation already given in Section 5.6. The placeholder here will represent the possible time elapsed from the initial state.

PlDomain is the set of naturals.

The propagate function is defined as follows

propagate(pl, e):

return *pl (+) duration(e);*

where the "(+)" operator "adds" a duration interval to a set of points according to the following definition:

$$(+) : N^+ \times P(N^+) \rightarrow PN^+$$

$$t\ (+)\ [min, max] = [t + min, t + max]$$

The initial time is $p_0=0$

As the domain here is infinite, we define PlFiniteDomain by restricting PlDomain to the interval [0,MAX] where MAX is some upper time bound to be chosen by the user.

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

In particular, inferring or verifying the *minimum and maximum time taken by a process* is achieved by looking at the fixpoint decoration generated at the last process state (e.g., state 11 in Fig. 4-7), taking the least and greatest time points in the codomain of the decoration.

This technique may also be used to infer or verify time constraints on tasks. The annotation of a refined task by a time constraint is either a requirement we would like to verify or a preliminary estimate to be replaced by a more accurate duration interval inferred from the refinement.

In the latter case we locally apply the instantiated decoration generation algorithm on the refining sub-process. This requires knowing the initial condition C_0 of the sub-process itself. This condition is directly obtained by use of the other algorithm instantiation described in Section 6.1.

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

The instantiation of the abstract decoration algorithm discussed in this section can be extended in a fairly obvious way for calculating other cumulative properties such as *costs* or *doses* in medical processes (drug doses, radiation doses, etc.). We may thereby check non-functional constraints about costs, doses or resource usage. The next chapter will show an example of underdose detection.

6.4.2. Example

In the meeting scheduling example, the minimal overall processing time is 3 days while the maximal one is *Out_Of_Bounds*. Reaching this upper bound means that date conflicts might remain unresolved so that the process keeps going. We might then increase this upper bound and see what happens.

6.5 Summary

This chapter introduced a variety of tool-supported techniques for analyzing guarded hMSCs process models, namely,

- for verifying that guards at a decision node are disjoint, complete and locally satisfiable,
- for verifying or generating task preconditions,

- for detecting potentially inadequate decisions due to the occurrence of unexpected events or due to outdated information about the environment.
- for detecting violations of time constraints on the process.

All the analyses proposed in this chapter proceed in three steps:

1. A g-LTS is first generated from the guarded hMSC [Lam11].
2. Each state of this g-LTS is automatically decorated with placeholders that are meaningful to the specific check the user wants to perform. This decoration is performed by instantiating the algorithm given in Chapter 5. To instantiate the algorithm, the user needs to provide a placeholder domain and a *propagate* function that describes how placeholder values are propagated through transitions.
3. The placeholder values are then used in order to perform the analysis.

Such analyses will be illustrated in Chapter 7 on a real case-study where a process model for cancer therapy is incrementally elaborated and analyzed.

Chapter 7 Evaluation: Incremental Building and Analysis of a Process Model for Cancer Therapy

7.1 Introduction

A US survey revealed that 98000 patients die every year in the U.S. due to medical errors [Koh99]. According to the U.S. Institute of Medicine, a large number of medical errors can be attributed to flaws in healthcare processes, leading the involved actors to make errors or not succeed in avoiding them [IOM01]. Such flaws are often due to a lack of coordination among the various medical actors involved in a complex process.

Medical treatments become increasingly complex, requiring interventions of multiple medical actors. They can be complex to the point that no medical actor can precisely explain the different steps of a treatment. An actor can precisely explain the steps she is involved in without a clear view of the global picture of the treatment.

A model capturing the various steps of a medical treatment together with their interrelationships is a good vehicle for communication, coordination and detection of possible flaws in work procedures. Such models can be used for documentation, analysis, explanation and enactment of healthcare processes.

We have used our process modeling and analysis techniques in a joint project with cancer therapists at the UCL university hospital in Brussels.

The guarded hMSC formalism appeared especially appropriate for modeling medical processes; it supports therapy scenarios involving interacting agents, sequencing of phases composed of tasks and decision on event-based and state-based variables. As we experienced it, the language is fairly close to

the informal one used by the medical stakeholders who provided us initial material consisting of flowchart sketches. Moreover, the guarded hMSC models we built afterwards were fully understandable by them.

Building an adequate, consistent and complete model for complex therapies is far from easy. The model sketches initially provided to us by medical staff were containing a large number of flaws. The use of formal models forced modelers to be more precise and allowed them to improve their understanding of the problem.

Our tool-supported analyses guided modelers by highlighting model fragments containing flaws to be fixed.

This chapter presents a step-by-step construction of a process model for treatment of rectal cancer using the GISELE toolset. This tool implements the analyses discussed in Chapter 6.

Our input included a documented protocol defining multiple options and decisions to be taken, the corresponding diagnosis and treatment procedures, together with strict constraints on therapy timing, medication dose and accuracy of medical test results. Parts of this protocol were summarized in flowchart form by the process stakeholders.

This chapter shows our modeling and analysis techniques in action on that specific treatment process; we will see how our techniques may help medical staff detect and correct flaws in process models early in their elaboration.

As the GISELE supporting tool was used during the elaboration of the model, Section 7.2 briefly introduces the main functionalities of this toolset. Section 7.3 outlines a general methodology we followed for building decision-based process models. Section 7.4 discusses the step-by-step construction of the model for rectal cancer treatment. Finally, Section 7.5 discusses the approach.

7.2 Tool Support

This section presents a *user* view of the GISELE toolset that supports the techniques presented in the thesis. For more details about the architecture and implementation of this toolset, the reader is referred to [Lam11].

The GISELE toolset was designed in collaboration with Bernard Lambeau. The implementation itself was done by Bernard Lambeau only [Lam11].

The GISELE toolset has a web front-end to implementations of the different analyses presented in Chapter 6, namely:

- local analysis of decision nodes against completeness, disjointness, and satisfiability,
- pre-conditions checks,
- detection of task-dependent and time-dependent decision inadequacies,
- verification of non-functional requirements about timing and doses.

In addition to those analyses, the toolset includes a model-checker that checks guarded hMSC models against temporal properties formalized in FLTL [Dam09, Lam11].

Fig. 7-1 shows a tool screenshot. The main GUI is divided in three panels. The left panel, called *"Tasks"*, allows the user to navigate through various tasks. The central panel presents the graphical representation of the current guarded hMSC model. The right panel, called *"Outline",* contains a textual specification of this model.

In the *Outline panel*, the problematic nodes are highlighted in red (*Surgery* and *StaffMeeting* in Fig. 7-1); the non-problematic ones are highlighted in green (all nodes in Fig. 7-1 except *Surgery* and *StaffMeeting*). Problematic nodes are nodes where a check failed - revealing a decision node being inadequate, or with incomplete, overlapping or unsatisfiable guards, or a task node whose pre-condition, timing or dose constraints might be violated.

When clicking on a red item, a pop-up window appears, showing the cause of the problem.

As an example, the pop-up window for the *Surgery* task is shown in Fig. 7-2. This window explains that the pre-condition to be met is *surgery_envisioned* while the invariant actually computed just before the execution of *Surgery* is *emergency*. The precondition can be violated as the state invariant does not imply the precondition. The tool returns a counterexample giving all variable assignments valid before the task execution that violates the task precondition. In the figure, the counterexample is

$$emergency \wedge \neg surgery\_envisioned.$$

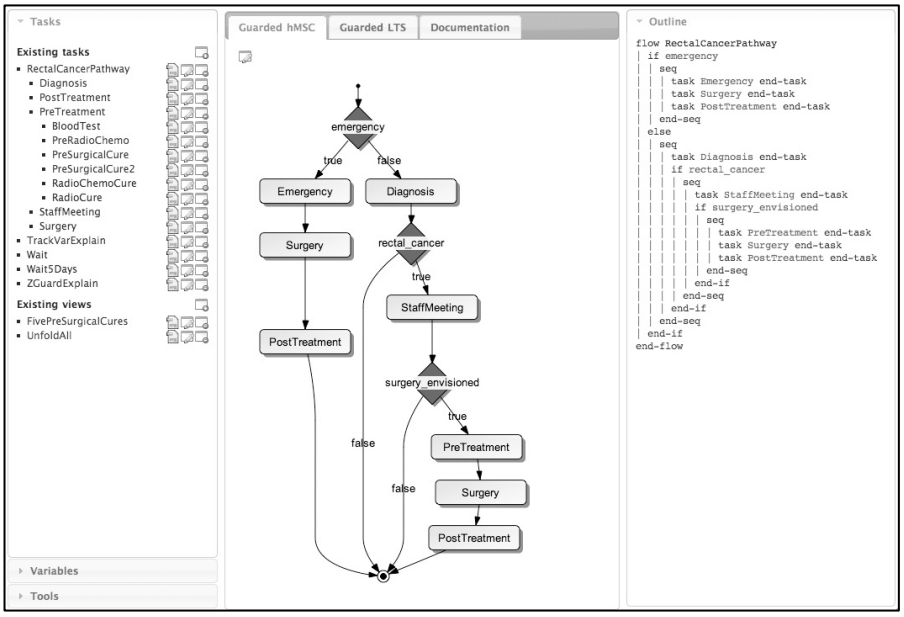


Figure 7-1 Global screenshot of the toolset front-end

An example of pop-up window for a problematic decision node is given in Fig. 7-3. In this figure, the guards of the decision nodes overlap; the guard g_3 overlaps with guard g_1 and guard g_2. For each pair of overlapping guards, the tool gives all variable assignments that satisfy them.

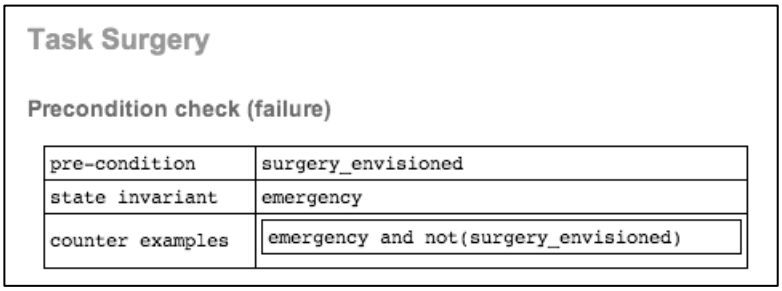


Figure 7-2 Pop-up window for a problematic task due to pre-condition violation

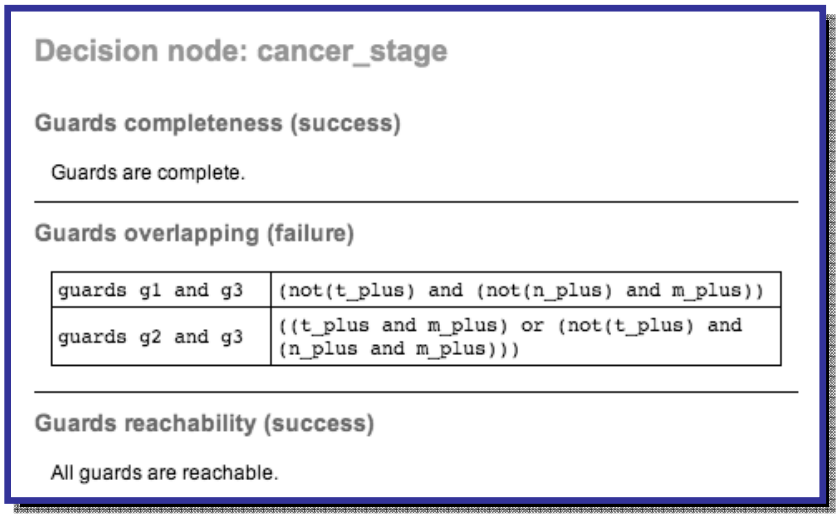


Figure 7-3 Pop-up window for a problematic decision node due to overlapping guards

As mentionned in Section 4.6, tasks may be annotated with timing, dose and precondition information. A window for defining such information is shown in Fig 7-4.

To enable incremental checks, two preconditions, doses and durations are given, an expected and a required one.

- The expected one is used by our tool when we want to check that the higher-level tasks containing the considered task satisfy their required property.
- The required one is used when we want to check that the considered task satisfies its required property.

An example of check result is given in Fig. 7-5. It shows a duration violation. The duration of the task *FivePreSurgicalCure* is required to be less than 40 days. The tool has computed that there are task executions that may take 42 days. The most specific invariant after such executions is the assertion *already_waited* (where *already_waited* is a fluent).

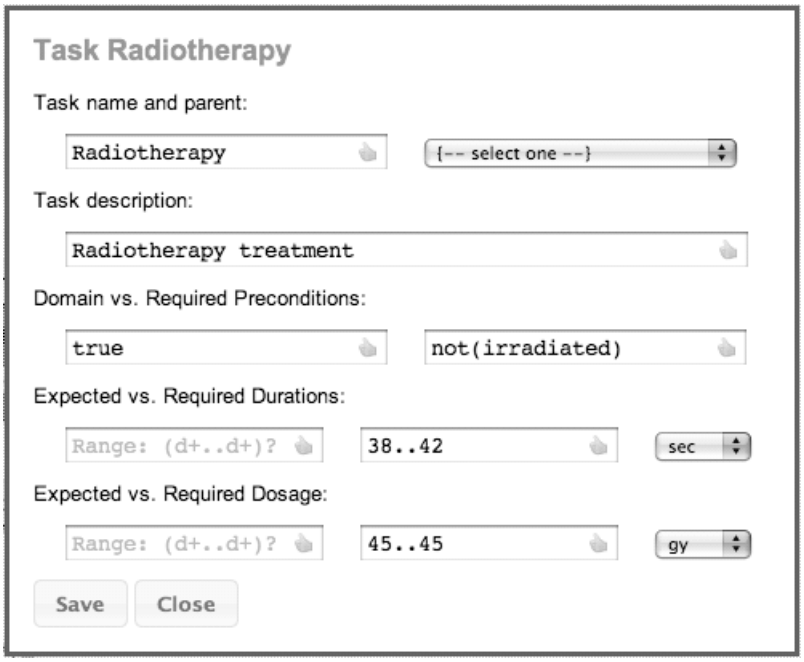


Figure 7-4 Window for annotating a task

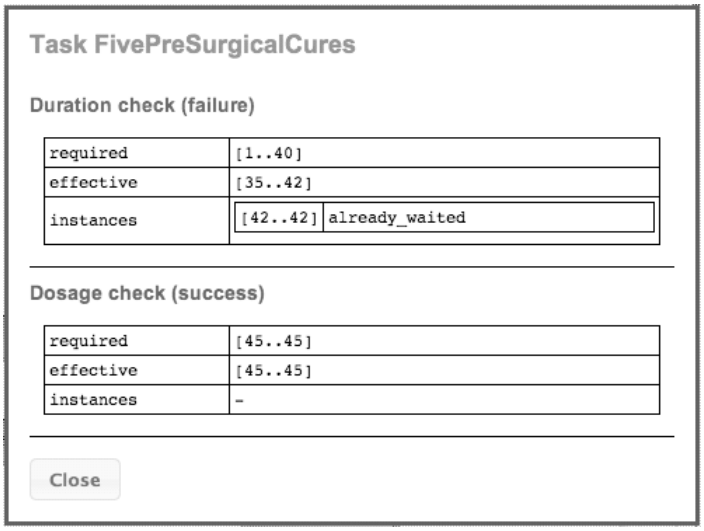


Figure 7-5 Pop-up window for a problematic task due to timing violation

7.3 Methodology followed for building decision-based process models

The construction of decision-based process models should be performed by analysts in collaboration with stakeholders. In our experience, the latter provide initial model fragments on paper, using box-and-arrow diagram pieces that exhibit tasks and decision nodes. Such informal sketches are sometimes also available from medical guidelines used by medical staff.

Such fragments are translated by analysts into an initial guarded hMSC model using the GISELE tool. The translation process triggers fruitful discussions between analysts and stakeholders. Issues about the process are raised and discussed, leading to early clarification of the preliminary paper version.

The analysts should first translate high-level fragments that show the overall, coarse-grained tasks of the process. While this high-level model is assembled, task preconditions, safety properties and non-functional requirements about timing and resources are elicited from the stakeholders.

Variables in decision nodes are formalized in fluents or in tracking variables. Tracking variables are suited for modeling variables tracking environment quantities, e.g., the variable *platelet_low* which is true if the last measure of the patient platelet level was low. Each tracking variable may lead to the identification of its associated accuracy meta-fluent that describes when the tracking variable is up-to-date. Variables not depending on the environment of the process are modeled as fluents, e.g., *diag_known* which is true if the diagnosis of the patient is known.

The GISELE tool is used to guide analysts and stakeholders in the incremental elaboration of the process model. At any step of the modeling process, the tool may highlight problematic nodes in the model and the analysts are invited to correct them. Error detections may stimulate discussions between stakeholders and analysts and therefore contribute to the model elicitation process.

The analyses can be made early on models that are still small and coarse-grained. Early analysis avoids the late fixing of anomalies disseminated at different places of a large and complex model.

When this high-level model is stabilized, because of no further error pointed out by the analyzers, the high-level tasks may be refined in new guarded hMSCs or MSCs. The analyses are then replayed on the finer-grained tasks. When a guarded hMSC is stabilized, it may be further refined.

If a task is simple enough and does not contain decision or loop, it should be refined in a MSC; otherwise it should be refined in a finer-grained guarded hMSC.

7.4 Model construction and analysis

From a high-level point of view, the treatment of rectal cancer can be described as follows:

"In general, a patient 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 evaluation is aimed at estimating cancer invasiveness and extension data:

- ***T** for local Tumor invasion,*
- ***N** for lymphatic Nodes invasion,*
- ***M** for distant Metastasis.*

If the rectal cancer is confirmed, the medical staff envisions some appropriate therapy strategy based on the evaluation. When the patient can undergo surgery, the main curative treatment will consist 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 prescribed directly."

7.4.1. Top-Level guarded hMSC for rectal cancer treatment

Fig. 7-6 shows a top-level guarded hMSC model draft for rectal cancer treatment based on the previous description. In the general case, the treatment consists of a sequence of *diagnosis*, *staff meeting*, *pre-treatment*, *surgery*, and *post-treatment* tasks. The task *StaffMeeting* is a meeting where all agents involved in process make clinical decisions about the future treatment of the patient.

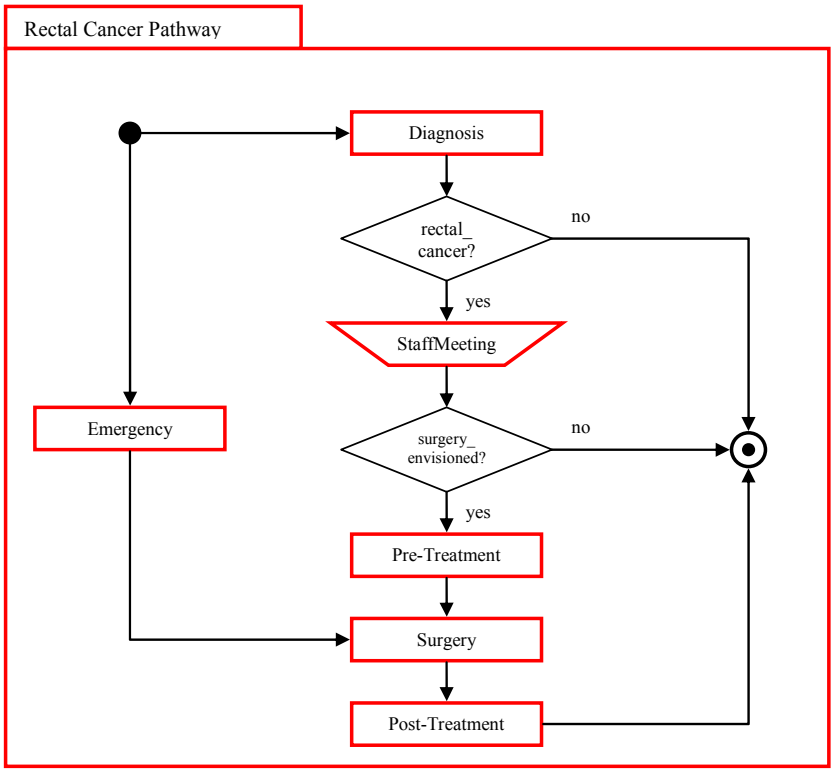


Figure 7-6 Top-level guarded hMSC model for the treatment of rectal cancer

Some of these tasks will be refined in other guarded 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 will follow another, separate palliative care process.

Guard formalization requires fluents and tracking variables to be identified and defined. Here, *rectal_cancer* and *surgery_envisioned* are tracking variables which are defined as follows:

tV *rectal_cancer* = $\{Diagnosis_{end}\}$

tV *surgery_envisioned* = $\{StaffMeeting_{end}\}$.

These definitions capture that *rectal_cancer* gets true if the diagnosis has revealed the presence of a cancer whereas *surgery_envisioned* is true if the medical staff has decided that surgery is the best plan to fight the patient's cancer.

Tasks may be annotated with preconditions on fluents and tracking variables (see Section 4.6.1). 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 the *pre-treatment*, *surgery*, and *post-treatment* tasks if the cancer is confirmed and the surgery is envisioned for the patient. The precondition of the task *StaffMeeting* is the logical expression *diag_known*, where *diag_known* is a fluent defined as follows:

$$fluent\ diag\_known = <\{Diagnosis_{end}\}, \{PostTreatment_{end}\}>.$$

This precondition means that the staff 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 few tasks and such tasks are coarse-grained.

Checking preconditions. Thanks to the generic approach in Section 5.4 instantiated to check preconditions (see Section 6.2), we can detect that the preconditions of *Surgery* and *PostTreatment* can be violated. Indeed, if the patient comes from an emergency service, the tracking variables *rectal_cancer* and *surgery_envisioned* will not necessarily be true.

To resolve these problems, different strategies may be proposed to the medical staff:

- The preconditions of *Surgery* and *PostTreatment* might be weakened as they turn to be too strong.

- Two decision nodes "*rectal_cancer*?" and "*surgery_envsioned?*" might be inserted just after the *Emergency* task, requiring these two conditions to be met before the task *Surgery*.
- A distinction between two different surgeries might be made, namely, *NormalSurgery* and *EmergencySurgery*. In that case, the tasks to be performed after *EmergencySurgery* should 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 these two cases.

All tasks are to be refined. In the next sections, we will refine some of them.

7.4.2. Refinement of the *Diagnosis* task

In this section, we will refine the *Diagnosis* task. This task consists in evaluating the patient tumor in order to select the right treatment for the patient. In order to accurately evaluate the tumor, an endoscopy should be performed followed by a biopsy and a spread evaluation. For some patients, such results may already be available from earlier evaluations in another hospital. In this case, the exams should not be repeated. A guarded hMSC refining the *Diagnosis* task is shown in Fig. 7-7.

In addition to the fluent *diag_known* already defined, the fluent *extension_known* is introduced:

$$fluent\ extension\_known = <\{SpreadEvaluation_{end}\}, \{Chemotherapy_{end}\}>$$

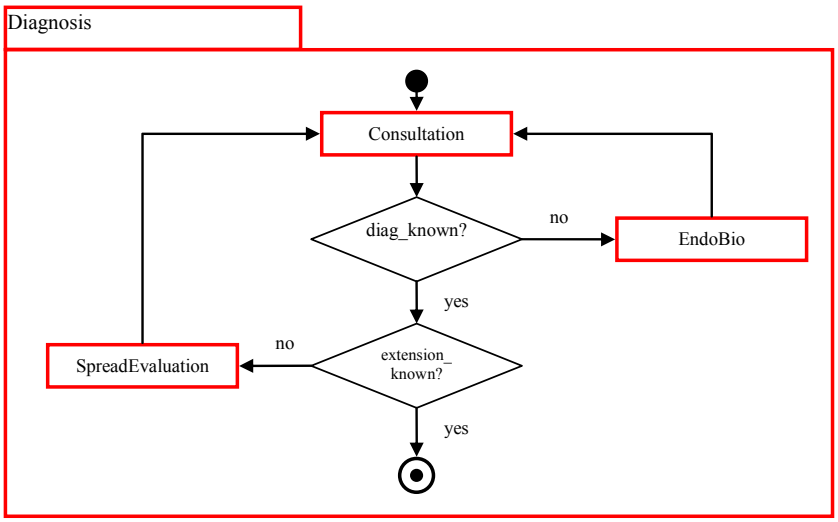


Figure 7-7 Refinement of the task *Diagnosis*

This fluent captures the fact that the result of spread evaluation makes cancer extension assessable. Such result is valid until a chemotherapy treatment; the latter will modify the status of the tumor. The constraint among initial values of fluents should be added to the initial condition C_0 of the global process. Here, we need to add:

$$extension\_known \supset diag\_known,$$

to reflect the following domain property elicited from clinicians: *"if spread evaluation is already performed in the initial state (possibly in another hospital), then the diagnosis is necessarily known (as possibly communicated by that other hospital)"*.

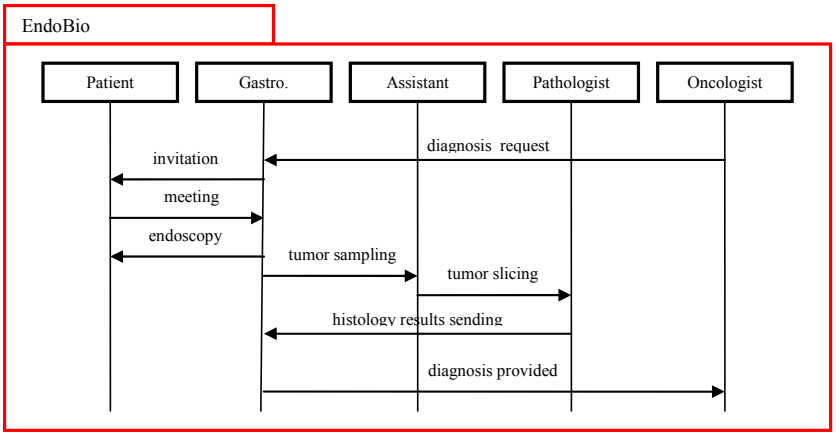


Figure 7-8 MSC for the leaf task *EndoBio*

The tasks *EndoBio* and *SpreadEvaluation* can be refined in a MSC as they appear fine-grained and contain no decision or loop. The refinement process is thereby completed for these tasks. Fig. 7-8 presents the MSC for *EndoBio*. This scenario is triggered by the oncologist in charge of the patient. A gastroenterologist collects a sample of the tumor, which is then prepared for microscopic examination by a pathologist. The results of the examination are finally communicated to the oncologist in charge of the patient.

To be more accurate, the fluent definition of *diag_known* can now be made further precise as follows:

$$fluent\ diag\_known = \langle \{diagnosis\ provided\}, \{PostTreatment_{end}\} \rangle$$

Note the difference with respect to the earlier coarser-grained definition in Section 7.3.1; the event $Diagnosis_{end}$ has been replaced here by the event *diagnosis_provided.*

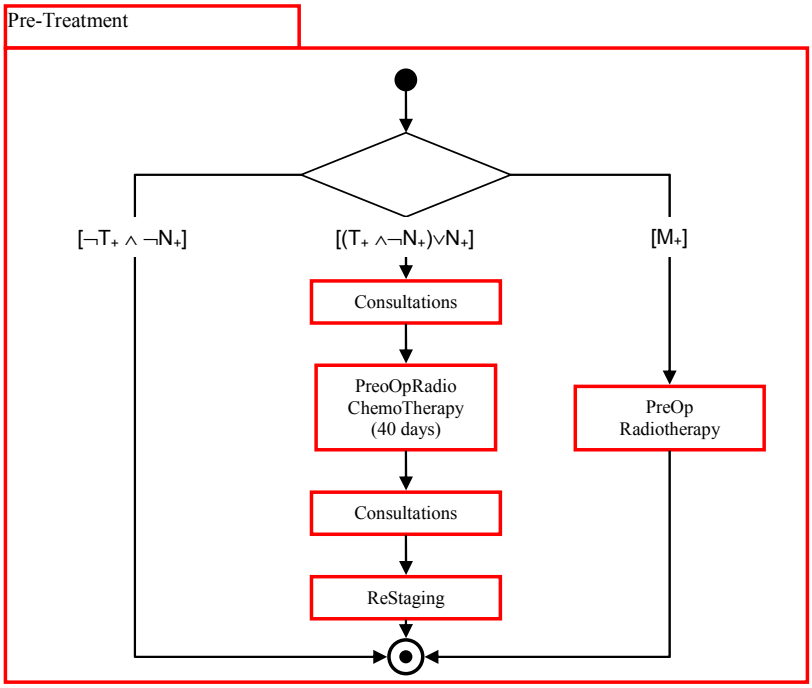


Figure 7-9 Guarded hMSC for the *Pre-Treatment* task

7.4.3. Refinement of the *Pre-Treatment* task

In this section, we refine the *Pre-Treatment* task introduced in Section 3. The refinement is shown in Fig. 7-9. During the staff meeting preceding *Pre-Treatment* in Fig. 7-6, a particular treatment will be decided for the patient. The decision is captured by the decision node in Fig. 7-9.

In Fig. 7-9, the three branches and corresponding guards are taken from a process documentation given to us by medical staff.

- 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 in intertwined radiotherapy and chemotherapy sessions.

- In presence of metastases, the treatment consists only in radiotherapy sessions.

Checking guards against completeness, disjointness, and satisfiability. We use the appropriate instantiation of our generic algorithm to perform the three types of guard analysis discussed in Section 6.1. Undesired non-determinism in decisions is automatically detected; the guards

$[\neg T \wedge \neg N]$, $[(T \wedge \neg N) \vee N]$, and $[M]$

are indeed overlapping.

The problem is easily fixed in this case by adding the conjunct $\neg M$ in the first two guards, as advised by medical staff. With regard to the two other properties, the guards are proved to be complete and all satisfiable.

Refining the task PreOpRadioChemoTherapy. The task *PreOpRadioChemoTherapy* in Fig 7-9, capturing intertwined presurgical radio and chemotherapy, can in turn be refined into another guarded hMSC.

This specific therapy consists of five radio-chemotherapy cures. 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 organizational reasons, the duration of this task should not exceed 40 days.

Fig. 7-10 shows a first refinement attempt where protocol excerpts were literally translated from the available 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, (D days) is used for [D days, D days] that is, the duration interval required to perform the corresponding task; a task box with a "*n*X" inside unfolds in a sequence of *n* occurrences of this task.

This figure also introduces the tracking variable *platelet_low* defined as follows:

$tV\ platelet\_low = \{BloodTest_{end}\}$

This tracking variable is known to be accurate until the end of a subsequent of a *radio-chemo treatment*; therefore, a corresponding accuracy meta-fluent is introduced:

$$fluent\ platelet\_low\_accurate = <\{BloodTest_{end}\}, \{RadioChemoTreatment_{end}\}> initially\ false$$

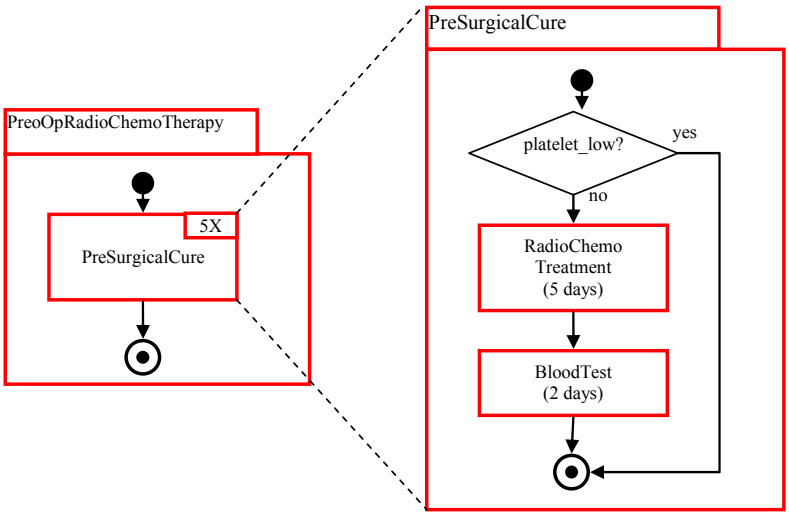


Figure 7-10 Refinement of the task *PreOpRadioChemoTherapy* (initial attempt)

Our techniques allow two problems to be detected at this stage.

Checking decision inadequacy. First, there is a task-dependent decision inadequacy (see Section 6.3); 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 fluent *platelet_low_accurate* may indeed be false when the decision node is evaluated.

The problem is fixed by moving the *BloodTest* task so that it appears right before the decision point (see Fig. 7-11.a).

Checking dose requirements. A second, more subtle problem being detected remains after rechecking the model of figure 7-11.a. If the

RadioChemoTreatment task is bypassed at least one time, due to low platelet level, the total dose administered in five cures will be less than 45 Gy.

To fix this problem, we should replace treatment cancelation by a waiting period of 5 days to allow normalization of the platelet level. The new model after the latter fix is shown in Fig. 7-11.b.

Checking time requirements. If we now recheck the revised model in Fig. 7-11.b, the tool highlights a problem of another type. This problem is detected by our algorithm instantiated to compute elapsed time (see Section 6.4). When the treatment is delayed too often, the 40-day duration constraint will be violated.

A model-based discussion with process stakeholders further reveals that a delay between radiotherapy cures is not advisable.

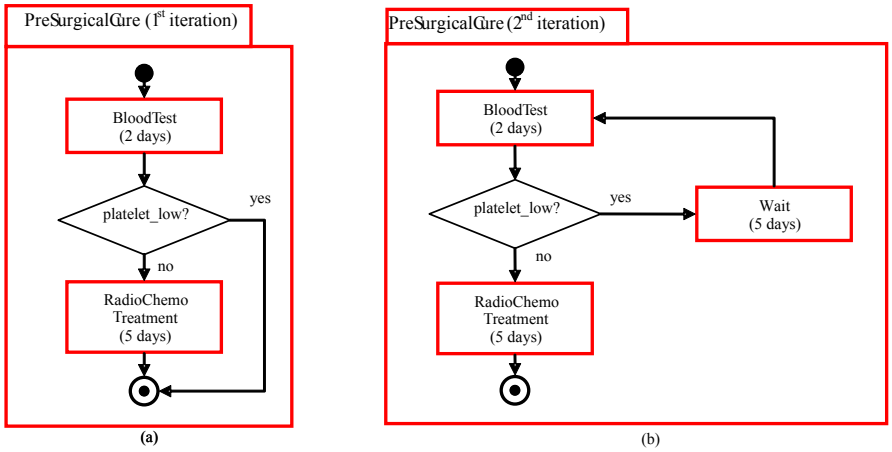


Figure 7-11 Check-driven refinement of the task *PreSurgical cure* (1st and 2nd iteration)

To resolve this problem, we may uncouple radiotherapy and chemotherapy treatments. Following medical advice, we introduce the possibility of delaying the treatment only on first occurrence of a platelet fall, see Fig. 7-

12. The variable *first_occurence* introduces there is a fluent defined as follows:

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

Re-checking time constraints. When rechecked again, the revised model still violates the 40-day duration constraint, as the longest possible execution may last 42 days. Such duration will be reached for all executions where the *Wait* task occurred in order to normalize the blood platelet.

Discussion with oncologist suggests that the 40-day duration constraint is a bit too strong, and we decide to move up the requirement to 42 days.

With the new requirement, the model revised after three iterations comply with the *time*, *dose* and *accuracy* requirements expressed in the protocol. Nevertheless, this version raises a new discussion with oncologists about the possibility of dose reduction rather than a mere skip of chemotherapy.

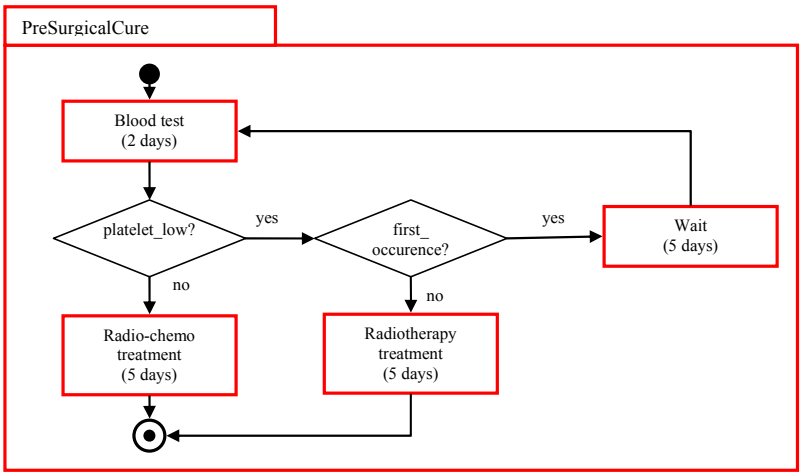


Figure 7-12 Refinement of the task *PreSurgicalCure* (3rd iteration)

7.4.4. Refinement of the *Post-Treatment* task

In this section, we will refine the *Post-Treatment* task introduced in Fig. 7-6. The guarded hMSC for post-treatment is shown in Fig. 7-13. *Post-treatment* begins with a staff meeting to evaluate the status of the cancer after surgery.

It will be followed by a specific treatment if the patient still wants to follow such treatment. This decision is based on the tracking variable *wished_treatment* formally defined as follows:

tV wished_treatment = {Consultation$_{end}$}.

This tracking variable needs to be reevaluated after each heavy treatment that may have an impact on the patient's motivation (radiotherapy, chemotherapy, surgery). This leads to the definition of the following accuracy meta-fluent:

fluent Wished_Treatment_Accurate =
<{Consultation$_{end}$},
{Surgery$_{end}$, PreOpRadioChemoTherapy$_{end}$, PreOpRadiotherapy$_{end}$}>
initially false

The specific treatment will depend on:

- the *T*, *N* and *M* parameters of the cancer,
- the fluent *radio_preop* capturing whether the patient has undergone radiotherapy sessions during pre-treatment,
- the tracking variable *contra_chimio* capturing whether the patient has contra-indications for chemotherapy.

The latter two variables are defined formally as follows:

fluent radio_preop =
<{PreOpRadiotherapy$_{end}$, PreOpRadioChemoTherapy$_{end}$}, {}>
tV contra_chimio = {StaffMeeting$_{end}$}

The introduction of these two variables leads to the elicitation of two properties, namely:

(a) "a patient may never be irradiated twice in the same anatomic zone"

(b) "a patient with contraindications for chemotherapy may not follow chemotherapy treatments"

In the following subsections, we will analyze the post-treatment phase through model-checking, precondition checking, and verification that all decision nodes are accurate, disjoint and complete.

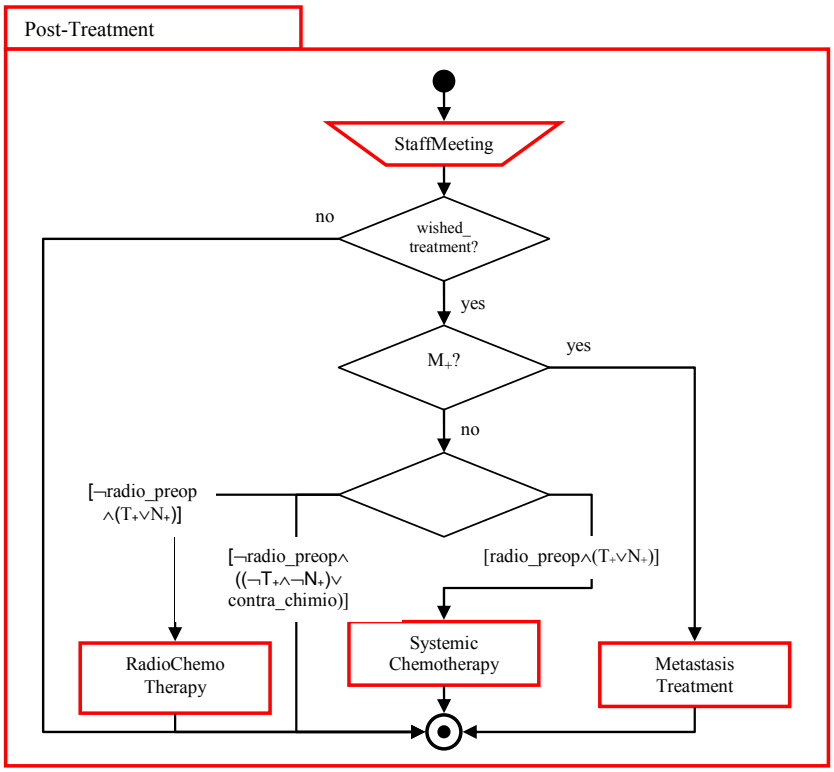


Figure 7-13 guarded hMSC for the task *Post-Treatment*

7.4.5. Guard Analysis

Checking guards against completeness, disjointness, and satisfiability. In Fig. 7-13, the checks on the two first decision nodes are fairly obvious. For the third decision, we must verify whether the guards

$[\neg radio\_preop \wedge (T_+ \vee N_+)]$,

$[\neg radio\_preop \wedge ((\neg T_+ \wedge \neg N_+) \vee contra\_chimio)]$,

$[(radio\_preop \wedge (T_+ \vee N_+))]$

overlap, are complete and are all satisfiable.

The two first guards are overlapping as the tool returns the following variable assignment that can make them both true:

$radio\_preop = false \wedge contra\_chimio = true \wedge T_+ = true \wedge N_+ = true.$

To resolve this problem, the condition $\neg\, contra\_chimio$ should be added to the first guard, yielding a new guard:

$[\neg\, radio\_preop \ \wedge \neg\, contra\_chimio \wedge (T_+ \vee N_+)].$

The resolution of those problems is shown in Fig.7-14.

On another hand, the set of guards is not complete; the following case is not covered:

$(radio\_preop = true \wedge T_+ = false \wedge N_+ = false).$

This assignment corresponds to the case of a patient already irradiated by another treatment with a small tumor without lymphatic nodes. The treatment to be given for such patient should be elicited from domain experts.

Note that those checks reveal increasingly subtle problems.

7.4.6. Model-checking a first property: *NeverIrradiatedTwiceInSameZone*

Model-checking guarded hMSC models is another useful technique for detecting flaws in process [Dam09, Lam11]. Model checking a FLTL property involves constructing a Büchi automata B that recognizes all infinite traces that violate the property. Next, the synchronous product of B with the trace-equivalent LTS of the guarded hMSC is verified to be empty. If not, a counter-example g-LTS trace is returned, that is, an initial variable assignment together with a sequence of event. For more details, the interested reader should refer to [Dam09, Lam11].

Let us verify the following property:

"A patient may never be irradiated twice in the same anatomic zone".

Its parent goal is to avoid the potentially serious side-effects of iterated radiation. The property implies in particular that a patient may not be irradiated during post-treatment if she has already been irradiated during pre-treatment. This can be formalized as follows in FLTL:

$$\Box\,(\bigcirc RadioChemoTherapy_{start} \rightarrow \neg\, radio\_preop)$$

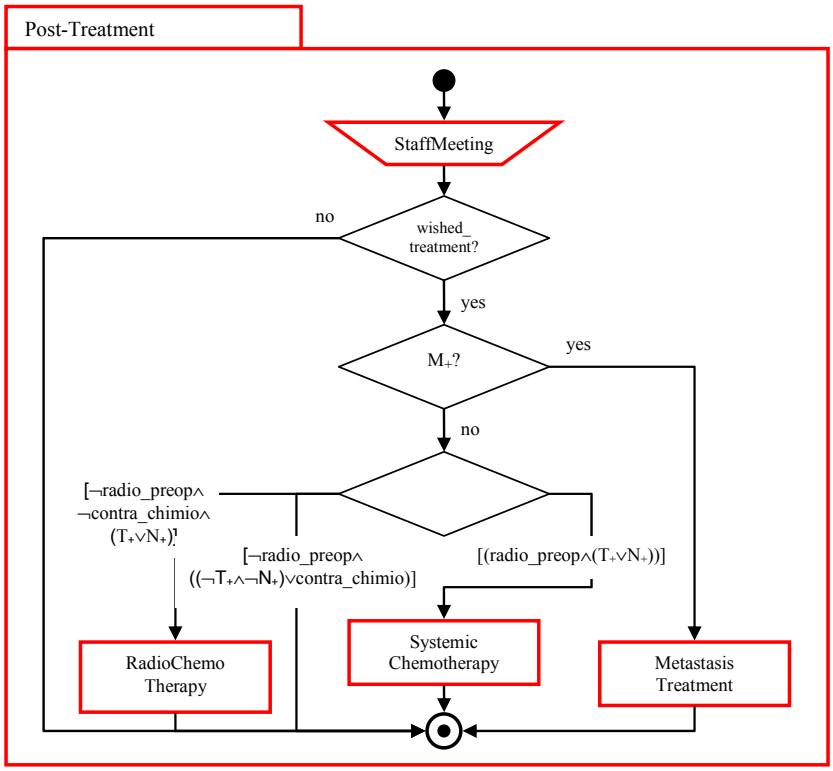


Figure 7-14 Revised version of task *Post-Treatment* after guard analysis

This property turns to be verified by the model-checker on the current model version. However, oncologists suggest that some patients might have been already irradiated prior to their treatment of rectal cancer.

This results in a new property that strengthens the previous one:

$$\Box((\bigcirc PreOpRadiotherapy_{start} \vee \bigcirc PreOpRadioChemoTherapy_{start} \vee \bigcirc RadioChemoTherapy_{start} \rightarrow \neg irradiated),$$

where irradiated is defined as follows:

$$fluent\ irradiated = \langle\{\ PreOpRadiotherapy_{end}, PreOpRadioChemoTherapy_{end,} RadioChemoTherapy_{end}\ \},\{\ \}\rangle.$$

Note the difference with the fluent *radiopreop*; the value of *irradiated* may be true for some patients in the initial state. When verifying the new property, our model-checker returns the following violating trace:

IRRADIATED AND M+ AND …

$Diagnosis_{start}$
$Consultation_{start}$
$Consultation_{end}$
$Diagnosis_{end}$
$StaffMeeting_{start}$
$StaffMeeting_{end}$
$Pre\_treatment_{start}$
$PreOpRadiotherapy_{start}$

This trace shows that the case of a patient already irradiated was overlooked in the original process fragments supplied by medical staff. The problem appears in the refinement of the *Pre-Treatment* task.

This problem may be fixed by putting further guards on all tasks involving radiotherapy in the pre-treatment (Fig. 7-9), e.g., the guard

$$[M_{+} \wedge \neg irradiated].$$

Such fix in the pre-treatment phase, however, breaks guard completeness on the decision node, which raises the following new question:

"What happens in case of already irradiated patient, where irradiation care would otherwise have been provided?".

After re-checking the same property on the modified model (Fig. 7-15), we now see that there is another problem with post-treatment. Here is the counterexample trace returned by the model-checker:

IRRADIATED AND NOT T+ AND NOT N+ AND …

$\texttt{Diagnosis}_{\texttt{start}}$
$\texttt{Consultation}_{\texttt{start}}$
$\texttt{Consultation}_{\texttt{end}}$
$\texttt{Diagnosis}_{\texttt{end}}$
$\texttt{StaffMeeting}_{\texttt{start}}$
$\texttt{StaffMeeting}_{\texttt{end}}$
$\texttt{PreTreatment}_{\texttt{start}}$
$\texttt{PreTreatment}_{\texttt{end}}$
$\texttt{Surgery}_{\texttt{start}}$
$\texttt{Surgery}_{\texttt{end}}$
$\texttt{StaffMeeting}_{\texttt{start}}$
$\texttt{StaffMeeting}_{\texttt{end}}$
$\texttt{RadioChemoTherapy}_{\texttt{start}}$

This trace reveals that the problem already discovered in the pre-treatment tasks appears in the post-treatment task as well.

To fix the problem, the medical staff might decide to replace every occurrence of the fluent *radio_preop* in Fig. 7-15 by the fluent *irradiated.* With those fixes in place, the property is now successfully verified by our model-checker.

7.4.7. Model-Checking a second property: *NoChemoIfContraIndications*

Let us now verify the following property

> *"a patient with contra-indications for chemotherapy may not follow chemotherapy treatments".*

This property can be formalized as follows

$$\Box((\bigcirc SystemicChemotherapy_{start} \vee \bigcirc RadioChemoTherapy_{start} \vee \bigcirc \text{PreOpRadioChemoTherapy}_{\text{start}}) \rightarrow \neg \text{ contra\_chimio}).$$

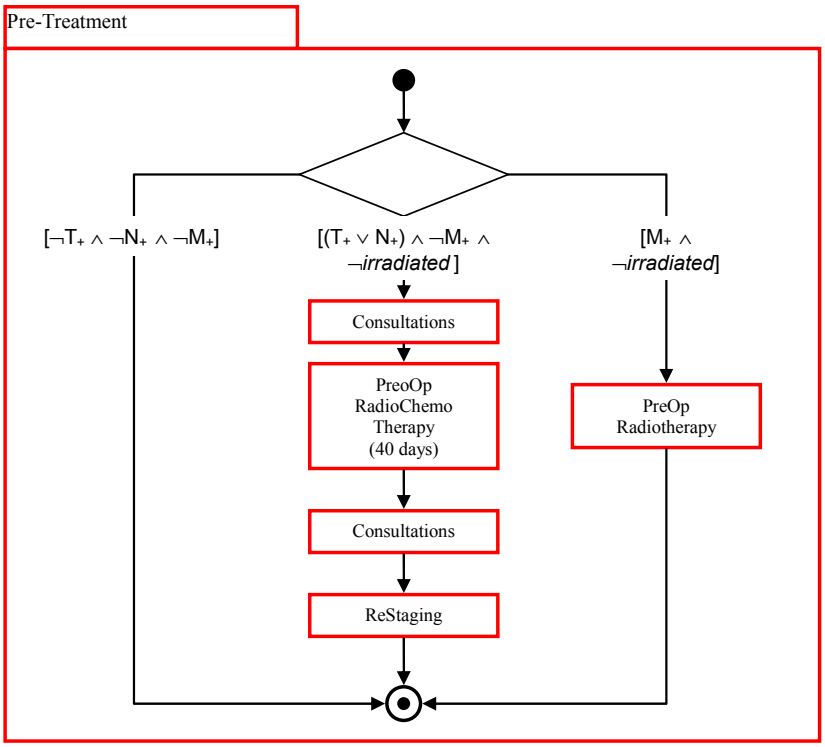


Figure 7-15 Revised version of Task *Pre-Treatment* after model-checking

The model-checker now returns the following counterexample trace:

```
CONTRA_CHIMIO AND T+ AND NOT M+ AND ...

    Diagnosis_start
    Consultation_start
    Consultation_end
    Diagnosis_end
    StaffMeeting_start
    StaffMeeting_end
    PreTreatment_start
    Consultations_start
    Consultations_end
    PreOpRadioChemoTherapy_start
```

The trace shows that a patient with contra-indication against chemotherapy may however follow chemotherapy treatment in the *pre-treatment* phase.

Similarly to the previous property, the problem is detected in another model portion than the one where the property was elicited.

To fix this problem, the *pre-treatment* phase involving chemotherapy should be restricted to patients with no contra-indications against chemo-therapy.

The following question then arises:

> *"What happens in case of patients with contra-indications for radiotherapy, where chemotherapy care would otherwise have been provided?".*

7.4.8. Further Precondition Checking

As reported in Section 7.4, the precondition of any *StaffMeeting* task is *diag_known*. For the specific *StaffMeeting* task within post-treatment, the tool detect that this precondition is violated for a patient coming from an emergency service. Indeed, such a patient did not undergo the *Diagnosis* task including *endoscopy*, *biopsy* and *spread evaluation*.

To fix this problem, we might add a *restaging* task for such patient; it consists of *endobio* and *spread evaluation* subtasks, to take place between the *Surgery* and the *StaffMeeting* tasks within *PostTreatment*. This *restaging* task might be added to the top-level guarded hMSC (Fig 7-6) or in the guarded hMSC for post-treatment (Fig. 7-14).

7.4.9. Further checking of decision adequacy

We can systematically check whether all decisions taken throughout the process model are adequate, that is, whether they rely on fresh, accurate information about the environment.

For example, there is a problem with the first decision node in the *Post-Treatment* task (Fig. 7-14). The fluent *wished_treatment_accurate* is always false before the first decision node. *Surgery* happens just before the *Post-*

Treatment task. It is a mandatory task for any patient; surgery is a terminating event and therefore the fluent is false for any patient in the initial state of the post-operation task (Fig. 7-14); the *StaffMeeting* task is not an initiating task of the fluent, and therefore the fluent is still false right before the decision node.

There is indeed a missing *Consultation* task during which the patient should express his agreement. To fix the problem, a consultation is therefore added just before the *StaffMeeting* task in order to ask the patient if she still wants to follow the treatment (Fig. 7-16).

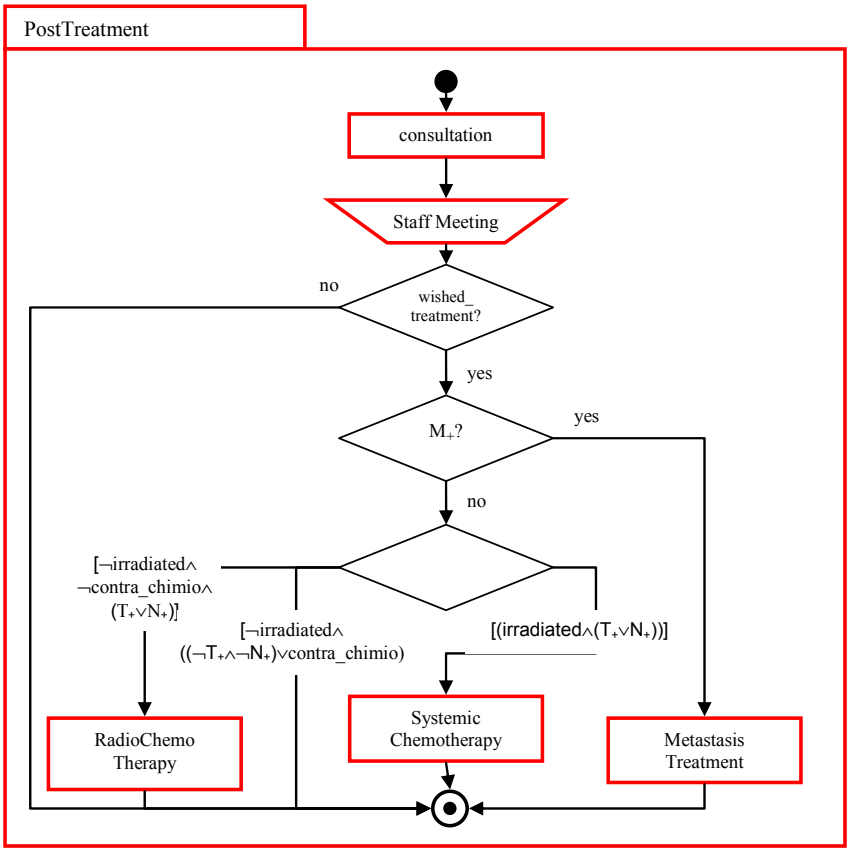


Figure 7-16 Revised version of task *Post-Treatment* after decision adequacy checking

7.5 Discussion

There are obstacles to the application of verification techniques to complex processes in the medical domain [Che08]. A correct model accurately describing the real process to be followed turns out to be very difficult to build. The amount of effort to be invested by medical professionals and the analysts involved may be considerable. The analysts have to learn enough of the medical domain to understand the medical experts. The latter need to learn how to abstract from multiple details and focus on key aspects of the treatment. They should also think about all possible situations that may arise. Another obstacle is the specification of required properties to be model-checked. Such specification may be hard to write. Domain experts often have problems being precise about the properties they want their processes to obey.

The errors found by our tool were mostly in the process models rather than in the *actual* processes followed by medical staff during their daily practice. Analysis is nevertheless important for a correct model to be used in a variety of contexts, e.g., to highlight potentially hazardous situations in real processes, enact daily work processes, advise less experienced people, deliver explanations and tutorial material, and so forth.

7.6 Conclusion

In this chapter, a process model for rectal cancer treatment was elaborated stepwise using the GISELE toolset. Some of the tools there implement the techniques described in Chapters 3, 5 and 6. They allowed us to perform different kinds of analysis on guarded hMSC models, namely,

- the analysis of guards decision nodes against completeness, disjointness, and satisfiability,
- the checking that pre-conditions are satisfied,

- the detection of inaccurate decisions,
- the verification of timing requirements and other non-functional resource-related requirements,
- the model-checking of safety properties.

The incremental model elaboration intertwined with those complementary analyses, followed by model revision and refactoring provides multiple benefits:

- We can systematically detect errors that frequently arise from novice modelers, such as non-satisfiable or overlapping guards, or that result from the inherent complexity of guards in the medical domain.
- Model analysis may reveal ambiguities in the model and suggest ways of fixing them.
- Error detection can be made early, thereby avoiding the difficulties of late fixing of anomalies disseminated at different places of a large, complex model.
- Early analysis contributes to the model elicitation process. Implicit information gets clarified.

The elaboration shown in this chapter is not complete - some tasks may be further refined, new properties or time requirements may be elicited, and further fixes for detected problems need be made.

Chapter 8 Related Work

This section compares our results with existing work along two directions. Section 8.1 reviews behavior modeling languages and the techniques available to analyze them. Section 8.2 does the same for process modeling languages. Section 8.3 reviews techniques that generates invariants on programs and compares them with our analyses.

8.1 Behavior modeling and analysis

Section 8.1.1 compares guarded LTS with other behavior models. Section 8.1.2 reviews different techniques for analyzing behavior models.

8.1.1. Behavior modeling

Numerous state machine formalisms have been proposed in the literature. There are different ways of classifying them.

We may distinguish *flat* and *hierarchical* state machines. Statecharts [Har87] was the first language to support hierarchical structuring of states. The language is gaining widespread usage since a variant has become part of the UML [OMG03]. The RSML language is a formal extension of Statecharts with interface descriptions and direct communication among parallel state machines [Lev94]. LTS and guarded LTS are flat languages; model structuring is achieved through the parallel composition operator Our guarded hMSC modeling language supports hierarchical refinement as well.

We may also distinguish *event-based* and *state-based* formalisms. In event-based formalism such as LTS, the target system is modeled in terms of sequence of relevant events. In state-based formalisms such as Statecharts, SCR [Par95, Hei96] or RSML, the target system is modeled in terms of relevant, application-specific states separated by events. Guarded LTS

cannot be classified in one of these classes only. Their transitions are labeled by events or by guards referring to system states. This formalism is closer to Doubly Labeled Transition Systems (L^2TS) [Den95]. Transitions there are also labeled by events. In addition, states are also labeled by sets of atomic propositions. Unlike guarded LTS, propositions are not defined in terms of events in L^2TS. To be consistent, a L^2TS model must meet certain conditions on state and event labeling.

State machines can also be extended to support specific concerns. For example, timed-automata [Alu90] are finite-state machines extended with clock variables.

8.1.2. Analyzing behavior models

Behavioral models may be manipulated by automated tools for a wide variety of purposes:

- to produce animations of the model in order to check its adequacy [Har90, Hol97, Lar97];
- to infer temporal specification inductively from these models [Lam98a, Cha00];
- to generate higher-level specifications such as invariants [Jef98];
- to check specific forms of model consistency/completeness efficiently [Hei96, Par98].
- to confirm that a behavior model satisfies a specification and if not, generate a counterexample [Que82, Cla86]. Model-checking techniques are available for different formalisms, e.g., for LTS [Mag06], for SCR [Atl93] or for timed automata using model-checkers such as UPPAAL [Lar97] or KRONOS [Daw94];

The last four categories here above are based on techniques that are closely related to those of the thesis. Let us discuss each of them.

Inference of temporal properties

The work closest to our goal inference algorithm is the one described by van Lamsweerde and Willemet [Lam98a]. They proposed an inductive learning technique for generating LTL goal specifications from positive and negative scenarios. In addition to scenarios, this technique requires a specification of pre- and postconditions of the operations whose applications define the interaction events in MSC scenarios. Our approach use fluents rather than pre- and post-conditions on event occurrences for the following reasons.

- Invariant generation requires fewer input information. Only the counterpart of postconditions has to be provided in fluent definitions; preconditions are derivable from the state machine.
- Fluents turn to be simpler to use; they allow the analyst to focus on one single state variable at a time; we can thereby annotate LTS incrementally.

Our work is also related to temporal-logic queries [Cha00], a technique for inferring temporal properties from transition systems. A temporal-logic query is a temporal-logic formula with a placeholder. Given a state-based transition model, a solution to a query is a set of assignments of propositional formulas to the placeholder; replacing the placeholder with any of these assignments results in a temporal logic formula that holds in the model. Various extensions were proposed. For example, [Gar02] extended temporal logic-query checking to allow for queries with multiple placeholders. Compared to our technique, temporal-logic queries can infer more property patterns. There are two main differences with our approach. Firstly, the properties are generated from state machines models and not from scenarios. Secondly, the properties are inferred from state-based

models rather then event-based ones. Such techniques could be used to infer properties after having decorated LTS with fluents.

Invariant generation

Our algorithm for generating state invariants along a LTS somewhat corresponds to a fluent-based counterpart of the algorithm for generating condition lists along scenario timelines in [Lam98a]; this algorithm was subsequently used in [Whi00] for statecharts synthesis. The specificities of our algorithm are the following:

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

Jeffords and Heitmeyer proposed an algorithm for generating state invariants for SCR state machines [Jef98]. Their algorithm is a fixpoint one like ours; it computes one invariant per mode. This algorithm cannot be used to generate invariants on LTS states as SCR state machines are quite different from LTS. For example, SCR state machines are state-based specifications over monitored and controlled variables while LTS are event-based ones.

Consistency/completeness checks on behavior model

Our checks on guards are close in spirit to those supported in [Hei96] for SCR tables. Beyond different formalisms there is a notable difference, however. The checks here must account for the contextual condition holding at the point in the process model where the decision node is reached. This contextual condition is the state invariant at that point, generated by our decoration algorithm.

[Par98] proposed a technique for checking whether a RSML specification is consistent. A RSML specification is *consistent* if two outgoing transitions from the same state that are triggered by the same event should have mutually exclusive guarding conditions. This technique is somewhat similar to our technique for detecting overlapping conditions as it also first generates an invariant on the source of the outgoing transitions. However, guards and invariants are quite different. In RSML, they are conjunctions of predicates; each predicate captures whether a given state in a parallel state machine is active or not.

Model-checking

Some of the analyses presented in Chapter 6 might in principle be performed through model-checking. As fluents and tracking variables may themselves be represented by state machines, some analyses are equivalent to model-check the parallel composition of the system extended with the state machines of all variables [Lam11]. Model-checking has the benefits of giving a counterexample trace when the given property is not valid.

For example, the check that a precondition P of a task T is never violated through a model can be done by verifying the following property

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

Also, checking that a decision may be inadequate may be done by model-checking a property of the form

$$\Box (guard \rightarrow acctV)$$

where *guard* is a fluent which is true when a decision node involving a tracking variable *tV* is evaluated and false otherwise, and *acctV* is the accuracy fluent associated when *tV*.

However, analyses such as check that a set of decision nodes are disjoint cannot be performed straightforwardly by model-checkers. Such checks are difficult to translate to meaningful temporal properties. In case of overlap,

our tool returns all variable assignments that meet several guards. Such information is typically not given by model-checkers.

In addition, model-checking techniques present two major drawbacks. Firstly, the properties to be checked may not be easily given by the user. Secondly, when multiple checks are performed, our decoration algorithm is more efficient than model-checking; the decoration generation cost may be distributed among *multiple* checks. One decoration computation may be used for several analyses.

8.2 Process modeling and analysis

This section reviews major process modeling languages and compares them with guarded hMSC in terms of their semantics, expressiveness and ease of use by domain experts. The available analysis techniques on such languages are also reviewed and compared.

8.2.1. Activity Diagrams and BPMN

Activity Diagrams [OMG03] and the Business Process Modeling Notation (BPMN) [OMG08] are standard graphical representations for business process models. Activity Diagrams were standardized in the Unified Modeling Language (UML) effort. Like guarded hMSC, they are flowchart-style process languages. Their objective is to support business process management by providing notations that are sufficiently intuitive. However, the notation specifications in [OMG03, OMG08] do not provide a precise semantics; this means that such diagrams are ambiguous and cannot be verified for behavioral correctness [Won11].

For this reason, different efforts were made in order to provide a formal semantics for these languages. The two most notable works in that direction are the following ones.

- A semantics for a subset of UML 1.5 activity diagrams was given by [Esh06] to check structural consistency constraints on them. The semantics is inspired by the Statemate semantics of Statecharts [Har87, Har90]. The NuSMV model-checker [Cim02] is used to verify consistency properties between the activity diagram and a set of class diagrams.
- Two semantics for a subset of BPMN in the CSP process algebra are provided in [Won01a]: an untimed and a relative timed semantics. Using those semantics, behavioral properties of BPMN models can be mechanically verified through model-checking. In addition, compatibility between interacting BPMN processes can be checked [Won01b]; this consists in verifying that the composition of the interacting processes is deadlock free.

UML Activity diagrams and BPMN are more expressive in several respects - e.g. loops, parallelism among sub-process, data flow between agents forming the process, transaction support. However, they do not support two basic features found in guarded hMSC.

- They do not support decisions based on variables of the process; they only support decisions based on non-deterministic choices.
- The checked properties are purely event-based and refer to events associated with task performance; state-based properties may not be checked.

8.2.2. YAWL

YAWL (*Yet Another Workflow Language*) [Van05] is a process language based on Petri-nets. Its objective is to support a wide variety of patterns called workflow patterns [Van03]. Related to design patterns [Gam95], workflow patterns refer to recurrent problems and proven solutions for developing of workflow applications.

Compared to g-hMSC, the expressive power of YAWL is higher:

- YAWL is based on Petri-nets and Petri-nets are more expressive than state machines.
- YAWL supports a lot of features. For example, YAWL can express workflows involving multiple instances or complex synchronization. Also, YAWL can also represent data flows and resource allocations.

YAWL models are much more complex and lower-level; they might be quite difficult to understand for domain experts. Moreover, guards in decision nodes are not formalized.

YAWL process models support different kinds of analysis based on their underlying Petri-net semantics. For example, [Van97] proposes a technique to verify that a YAWL model always terminates. Such check is based on standard Petri-net tools. For a guarded hMSC, termination can be proved by verifying that the time distance between the start node and each node in the corresponding LTS is always finite. Other analyses are performed at runtime; for example, [Roz08] proposes an incremental approach to check the conformance of a process model and its execution.

8.2.3. Little-Jil

Little-Jil is a process language with a visual syntax[Wis00]. Processes are described by task trees. This proves convenient for interaction with domain experts. In addition to task composition constructs, such as sequencing, parallelism and non-deterministic choice, Little-Jil provides an exception mechanism inspired from programming languages; this turns to be very useful when modeling medical processes, as exceptions turn to be very frequent there [Han06]. Tasks may also be annotated informally by pre- and post-conditions. However, guards in decision nodes are not supported.

Little-Jil has a formal semantics in terms of finite state-machines; various types of analysis can therefore be performed. A Little-Jil model-checker is described in [Ler04]. This model-checker may verify that event properties are satisfied and that the process is deadlock-free. However, unlike the properties we consider on fluents and tracking variables, the properties they can check are purely event-based and refer to events associated with task performance.

A heuristic technique has also been proposed for deriving fault trees from processes specified in the Little-JIL language [Che06]; the aim is to suggest that tasks might be wrongly executed, that communications might fail, etc. Along this direction, we might use the techniques for building obstacle trees in [Lam00], to be applied to the *Achieve* goals associated with each task.

8.2.4. Other process language formalisms

There are also several process modeling languages whose main objective is process enactment, e.g. SPADE [Ban94]. Some of these languages are thoroughly reviewed in [Fin94]. Those efforts do not consider the formal analysis of process models.

Various time analysis techniques were also proposed for specific languages [Bau06, Dem06]. Such techniques typically proceed in two steps; a state machine model is first derived from the input model and then checked against properties. In [Dem06], workflow models are encoded in timed automata [Alu94] and then model-checked. No high-level process language is available to modelers; process decisions are not supported. In [Bau06], medical guidelines are written in ASBRU and then rewritten in SMV for model checking. False negatives may however appear due too coarse time abstractions.

As in the other process formalisms, state/event-based guards in decision nodes are not supported in any of those languages.

8.3 Generating invariants on programs

Our analysis techniques are somewhat counterparts of algorithms that generate invariants over programs. Invariant generation consists in discovering inductive assertions on programs. Rather than generating invariants at each state of a state machine, such techniques generate invariants at specific program points. The main difference is that our techniques are applied on higher-level, more abstract models.

Our approach builds on good-old-times principles of symbolic execution [Kin76]. The principle of symbolic execution is to execute the program with symbolic values rather than actual ones. A symbolic execution may represent a large, usually infinite class of normal executions. Symbolic execution can proceed forwards through the program [Flo67] or backwards according to Hoare's backward substitution mechanism for program proving [Hoa69].

Abstract interpretation [Cou77, LeC94] combines symbolic execution with abstraction in order to ensure termination of the invariant generation process.

In abstract interpretation, the symbolic execution is not performed with the concrete semantics; the latter is the semantics describing the actual execution of the program. The symbolic execution is performed with an abstract semantics, which is an approximation of the concrete semantics, in order to make the analysis more tractable.

If the abstraction is sound, conclusions derived from the abstract semantics are also valid for the program concrete semantics and specification. In general, there is a trade-off to be made between the precision of the analysis and its efficiency.

As discussed in Section 5.7, the use of abstract interpretation might significantly improve the efficiency of our algorithms. However, abstract interpretation techniques require proving that each abstraction is sound with respect to their concrete one. Such proof is very unlikely to be available from end-users.

Chapter 9 Conclusion

Models may be used for a variety of purposes. Model building, however, is a complex and error-prone task. The construction and analysis of high-quality models should be supported by systematic methods and tools. The thesis provides a set of tool-supported analysis techniques that can be used in order to build more adequate, complete and consistent models.

The techniques in the thesis support incremental modeling, where models are built, validated and refined through successive iterations. The overall strengths of performing systematic model analysis with our approach are the following:

- Analysis can be performed on high-level models that are close to the material provided by stakeholders. A common language among stakeholders, analysts and supporting tools is provided. The flaws detected by the tools may be easily understood by stakeholders. Such flaws are then more easily corrected in the model by analysts and stakeholders. This may result in better model adequacy and completeness.
- Completeness is also improved by performing analysis early and on different aspects of the system. Our tool-supported analyses may be performed on partial models. Early detection of errors thereby contributes to the model elicitation process which in turn contributes to completeness. In addition, early detection avoids the difficulties of late fixing of anomalies disseminated at different places of a large, complex model.
- Both functional and non-functional aspects of the system can be analyzed with our techniques. For example, our techniques support the verification of timing, accuracy and cost constraints on process

models. Our checks are also customizable to dedicated analyses in order to handle concerns that are specific to certain application domains. Our analysis toolset might be further customized or extended in order to handle other non-functional aspects such as adequate resource usage.

- The systematic detection and fixing of errors during model elaboration enforces the consistency between the behavioral, operational and intentional views in process and system modeling. The thesis also shows how techniques used for model analysis may also enable model synthesis. Goal inference from scenarios could for example be adapted to infer goals from process models. Such joint use of model synthesis and model analysis techniques opens interesting perspectives for effective guidance during modeling.

More specifically, the thesis makes the following contributions.

(i) An algorithm for systematically decorate LTS state machines with state invariants on fluents, together with its correctness proof.

(ii) A high-level language for modeling decision-based processes, that combines the event-based and state-based paradigms, together with a lower-level formalism for defining its operational semantics.

(iii) A fully generic algorithm for decorating each state of a g-LTS model with generic placeholders, together with its correctness proof.

(iv) A rich set of complementary techniques for analyzing decision-based process models. These techniques are all instantiations of the generic g-LTS decoration algorithm. Those instantiations require minimal effort on the analyst who just needs to provide the placeholder domain together with a function that describes how placeholder values are propagated through events.

(v) An assessment of the proposed analysis techniques on a real case study of significant size for the elaboration of a complex, safety-critical process model for cancer therapy.

These contributions are further detailed in Section 9.1; limitations and open issues deserving future work are discussed in Section 9.2.

9.1 Contributions

This section follows the structure of the thesis.

9.1.1. Generating state invariants on LTS models

A fixpoint algorithm was described for generating state invariants on labeled transition systems so as to make such models easier to understand and validate. The generated invariants are defined over fluents

They can be used for more efficient analysis. For example, they were used in another technique for generating goals and domain properties from a set of positive scenarios (see Section 3.3).

Moreover, the decoration of each LTS node with its generated invariant makes the LTS much more comprehensible while exhibiting a state-based "look".

9.1.2. Modeling decision-based processes by guarded hMSC

Guarded hMSCs were introduced to model multi-agent processes involving decisions. In our experience, this flowchart-like formalism proved to be understandable by stakeholders and convenient for modeling decomposable tasks, interaction scenarios, associated goals, and decision trees combining event-based and state-based conditions. The originality of this language lies in its support of formal decision nodes. To the best of our knowledge, no

other process language provides similar analyses involving formal decision nodes.

Decisions refer to variables defined over process events. In addition to the fluent variables introduced before [Gia03], a new type of variable was introduced for higher expressiveness of decisions. Tracking variables are state variables tracking environment quantities that are not directly observable by the agents involved in the process. This introduction resulted, in particular, in a new type of analysis leading to the detection of tasks that are missing for decision to be taken on up-to-date information.

The language has a clear formal semantics defined in terms of LTS.

Tasks may be annotated by preconditions, time constraints and other non-functional attributes referring to costs, resource usage, etc.

Guarded LTS were proposed as an intermediate event-based formalism allowing state-based guards to be defined as conjunctions of fluents and tracking variables. In this thesis, the different analyses are performed at this level to keep state enumeration implicit.

9.1.3. A generic algorithm for generating abstract decorations on guarded LTS

An algorithm was presented for decorating each g-LTS state with generic placeholders. The latter are abstract quantities that should be instantiated for the specific analysis considered; an example of placeholder is the time elapsed from the initial state. Our generic algorithm is easy to instantiate. For each different type of placeholder, the analyst only needs to instantiate the rule for propagating placeholders through a single transition. There is no need to specify the lattice structure to be manipulated by the fixpoint order

(in particular, the supremum operator and underlying order). No instantiation-specific proofs are required. Its correctness was proved.

9.1.4. Analysis techniques for decision-based process models

Various instantiations of the same generic algorithm were discussed to analyze a guarded hMSC model under different, complementary perspectives. The instantiations allowed us to highlight a variety of modeling errors early and incrementally on partial models, including,

- inadequate decisions resulting from inaccurate or outdated information about the environment;
- incompleteness of decisions;
- unreachability of specific tasks along certain process paths;
- undesirable non-determinism in process decisions;
- violation of temporal constraints;
- violation of other non-functional constraints, e.g. about costs or resource usage.

The analyses were applied in combination to build a model for a real safety-critical process for cancer treatment. Errors in the specification fragments supplied to us by medical stakeholders were successively detected and corrected while building the model.

9.2 Limitations and Future Work

The work presented in the thesis has limitations that call for extensions in several directions.

9.2.1. Improving the expressiveness of guarded hMSCs

Compared with complex notations such as BPMN [OMG08], the language of guarded hMSC is intuitive and simple enough for understanding by process stakeholders. As a price to pay, the language may be seen to be not very expressive. Here are some desirable features that could be added to the language.

- *Variables over bigger domains.* Currently, the variables used in guarded hMSC must be Boolean variables. This obviously is a limitation of our formalism. For example, integer variables would be useful for defining finite loops.
- *Parallel constructs.* Parallelism is currently captured in our guarded hMSC within MSC nodes, at the lowest level of refinement. This might not be sufficient for processes involving coarser-grained tasks to be done in parallel. Higher-level fork/join constructs should be made available as in many other process languages. The semantics of such constructs should however be defined carefully. For example, issues such as the following should be addressed: *May a sub-process change the value of a variable being used in a guard of another sub-process concurrent with it? May two parallel sub-processes use the same event? If so, do the two sub-processes synchronize on shared events?*
- *Exception handling mechanisms.* Exceptional behaviors are currently captured in guarded hMSC through decision nodes. However, the number of exceptional conditions may be quite large in real processes. In some situations, exceptions may happen during any task (in the medical domain, for example, the exception

"anaphylactic shock occurs" can happen during any drug administration). Using decision nodes to cover large numbers of exceptions, some fairly infrequent, clutters the model and makes it become unreadable. Exception handling mechanisms similar to those provided by other process languages [Ler10] should be made available.

New modeling construct requires updating the analyses in order to handle a richer language. A language with a lot of features is however less easy to use and to understand as exemplified by languages such as YAWL. Some good tradeoff should thus be established here.

9.2.2. Improving decision-based process analysis

Our analysis techniques raise further issues.

If a check detects an error in the process, the feedback given by our technique and supporting tool highlights a problematic state and provides a state invariant for this state. As an additional feedback, a counter-example trace should be provided to suggest the root causes of the problem in the process model.

Another improvement would be the provision of domain properties in the model. Such properties could be used, in particular, for simplifying derived information such as preconditions in order to make them more compact and more understandable.

As discussed in Section 5.7, the algorithm to generate placeholder decorations might not be efficient enough in some cases. The use of abstract interpretation techniques might then produce more efficient analyses [Cou77, LeC94].

In addition, the convergence of the decoration algorithms can be sped up through a good strategy during the selection of a node in the set *ToExpl*[1]; *ToExpl* is the set of nodes to which propagation should be further applied. The strategy for selecting a node in *ToExpl* is important as it has an impact on the number of iterations needed to reach a fixpoint. Some propagations have no impact as the new computed decorations will be completely overridden by future propagations. It is preferable to select nodes that have a durable impact on decorations of successor nodes. A good strategy could consist in recording the elements of *ToExpl* in a priority queue. In this priority queue, elements of *ToExpl* are sorted topologically; a node will be selected in *ToExpl* after the nodes leading to it.

9.2.3. Using goals jointly with operational workflows

Our operational process models should be enriched with other declarative information, such as the goals underlying tasks, for complementary analyses [Lam09].

Goals prove to be more stable as the tasks operationalizing them can change regularly. Further to providing the rationale for tasks, the use of goals could improve the process model by pointing out missing paths. Goal refinements may be checked against completeness. Incomplete goal refinements may lead to the identification of new goals and therefore new tasks operationalizing these goals [Lam09].

Obstacle analysis [Lam00] could further improve the process model by anticipating unexpected problems and exploring resolutions to these.

[1] I am indebted to Renaud De Landtsheer for giving me this idea.

Moreover, some tasks appear difficult to refine operationally (see, e.g. the task *StaffMeeting* in Chapter 7); a goal tree might be associated to such tasks to explain the task objectives together with their interrelationship.

9.2.4. Detecting interaction among multiple processes

The integration of multiple process models raises interference issues. For example, a patient following both colonorectal cancer and diabetes therapies is exposed to feature interaction problems, e.g., 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 to detect conflicts between the processes operationalizing them [Lam98b]. There might be different stages for detecting and resolving such problems: at modeling time or during the enactment of the processes. This should be investigated further in order to know which approach appears the best.

References

[Alp86] B. Alpern and F. B. Schneider, *Recognising Safety and Liveness,* Department of Computer Science, Cornell University, TR 86-727, January 1986.

[Alu90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In Proc. of Int. Colloquium on Algorithms, Languages, and Programming, volume 443 of LNCS, pages 322–335, 1990.

[Atl93] Joanne M. Atlee, John D. Gannon: State-Based Model Checking of Event-Driven System Requirements. IEEE Trans. Software Eng. 19(1): 24-40 (1993)

[Ban94] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Luigi Lavazza. 1994. SPADE: an environment for software process analysis, design, and enactment. In *Software process modelling and technology*, Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh (Eds.). Research Studies Press Ltd., Taunton, UK, UK 223-247.

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

[Cha00] William Chan, Temporal-Logic Queries, In *Proceedings of the 12th International Conference on Computer Aided Verification* (CAV '00), E. Allen Emerson and A. Prasad Sistla (Eds.). Springer-Verlag, London, UK, 450-463.

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

[Che08] Bin Chen, George S. Avrunin, Elizabeth A. Henneman, Lori A. Clarke, Leon J. Osterweil, Philip L. Henneman: Analyzing medical processes. ICSE 2008: 623-632

[Cim02] 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",* in Proceeding of International Conference on Computer-Aided Verification (CAV 2002). Copenhagen, Denmark, July 27-31, 2002

[Cla86] 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. 2,1986, 244-263.

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

[Cou75a] P. Cousot & R. Cousot. Vérification statique de la cohérence dynamique des programmes. In Rapport du contrat IRIA SESORI No 75-035, Laboratoire IMAG, University of Grenoble, France. 125 pages. 23 September 1975.

[Cou75b] P. Cousot & R. Cousot. Static Verification of Dynamic Type Properties of Variables. Research Report R.R. 25, Laboratoire IMAG, University of Grenoble, France. 18 pages. November 1975.

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

[Dam05]C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde, “Generating Annotated Behavior Models From End-User Scenarios*”*, *IEEE Trans. on Software Engineering,* Special Issue on Interaction and State-Based Modeling*,* Vol. 31 No.12, Dec. 2005, 1056-1073.

[Dam06]C. Damas, B. Lambeau, and A. van Lamsweerde, “Scenarios, goals, and state machines: a win-win partnership for model synthesis”, *Proc. 14th ACM SIGSOFT Symp. on Foundations of Software Engineering*, Portland, Oregon, 2006.

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

[Dam10]C. Damas, B. Lambeau, F. Roucoux, A. van Lamsweerde, « Abstractions for Analyzing Decision-Based Process Models”, Technical Report, 2010.

[Daw94]Conrado Daws, Alfredo Olivero, Sergio Yovine: Verifying ET-LOTOS programmes with KRONOS. FORTE 1994: 227-242

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

[Den95] R. De Nicola and F. W. Vaandrager, *Three Logics forbranching bisimulation*, Journal of the ACM, Vol. 42, No.2, pp. 458-487, 1995.

[Dum05]M. Dumas, W. van der Aalst, A. ter Hofstede, *Process-Aware Information Systems*. Wiley, 2005.

[Esh06] R. Eshuis, “Symbolic model checking of UML activity diagrams”, *ACM Trans. on Software Eng. and Methodology* Vol. 15, No. 1, Jan. 2006, 1-38.

[Fea97] M.S. Feather, S. Fickas, A. Finkelstein, A. van Lamsweerde, *Requirements and Specification Exemplars* Automated Software Engineering, Kluwer Pubs, Vol. 4, No. 4 (1997)

[Fin94] Anthony Finkelstein, Jeff Kramer and Bashar Nuseibeh*, Software Process Modelling and Technology*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[Flo67] R.W. Floyd, "Assigning Meaning to Programs", in Proceedings of Symposium on Applied Mathematics, Vol. 19, J.T. Schwartz (Ed.), A.M.S., 1967, pp. 19–32

[Fox98] J. Fox, N. Johns, A. Rahmanzadeh, "Disseminating Medical Knowledge – The ProForma Approach", *Artif. Intell. Med.* Vol. 14, 1998, 157-181.

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Professional Computing Series, Addison Wesley, Reading, MA, USA, 1995.

[Gar02] Arie Gurfinkel, Benet Devereux, and Marsha Chechik. 2002. Model exploration with temporal logic query checking. *SIGSOFT Softw. Eng. Notes* 27, 6 (November 2002), 139-148.

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

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

[Har87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, Vol. 8, 231-274.

[Har90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, Mark B. Trakhtenbrot: STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Trans. Software Eng. 16(4): 403-414 (1990)

[Hei96] C.L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications", *ACM Trans. on Software Eng. and Methodology* Vol. 5 No. 3, July 1996, 231-261.

[Hoa69] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (October 1969), 576-580.

[Hoa85] C.A.R. Hoare, "Communicating Sequential Processes", Prentice-Hall International Series in Computer Science, 1985.

[Hol97] Gerard J. Holzmann: The Model Checker SPIN. IEEE Trans. Software Eng. 23(5): 279-295 (1997)

[IOM01] Institute of Medicine. Crossing the quality chasm: a new health system for the 21st century. Washington, D.C.: National Academy Press, 2001.

[ITU96] ITU, *Message Sequence Charts*. Recommendation Z.120, Intl. Telecom Union, Telecom. Standardization Sector, 1996.

[Jef98] R. Jeffords and C. Heitmeyer, "Automatic Generation Of State Invariants From Requirements Specifications", *6th ACM Symp. Foundations of Software Engineering*, Los Alamitos, California, 1998.

[Kai05] S.Kaiser and S. Miksch. *Modeling Computer-Supported Clinical Guidelines and Protocols: A Survey.* Vienna Univ. Technology, Rep. Asgaard-TR-2005-2, 2005.

[Kar00] Karamanolis, C. T., Giannakopoulou, D., Magee, J., and Wheater, S. M., "Model Checking of Workflow Schemas", *Proc. 4th Intl. Conf. on Enterprise Distributed Object Computing*, IEEE, 2000, 170-181.

[Kin76] James C. King: Symbolic Execution and Program Testing. Commun. ACM 19(7): 385-394 (1976)

[Koh99] 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

[Lam11]B. Lambeau, "Synthesizing Multi-View Models of Software Systems", Ph. D. Thesis, Université catholique de Louvain.

[Lam94]L. Lamport, "The Temporal Logic of Actions", *ACM Transactions on Programming Languages and Systems*, 16(3), 1994, 872-923.

[Lam98a]A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios*"*, *IEEE Trans. on Software. Engineering*, Vol. 24 No. 12, December 1998.

[Lam98b]A. van Lamsweerde, R. Darimont, E. Letier, *Managing Conflicts in Goal-Driven Requirements Engineering*, IEEE Transactions on Software Engineering, *Special Issue on Managing Inconsistency in Software Development,* November 1998.

[Lam00]A. van Lamsweerde, E. Letier, *Handling Obstacles in Goal-Oriented Requirements Engineering*, IEEE Transactions on Software Engineering, *Special Issue on Exception Handling*, Vol. 26 No. 10, October 2000, 978-1005.

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

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

[LeC94] Baudouin Le Charlier: Abstract Interpretation and Finite Domain Symbolic Constraints Constraint Programming 1994: 147-170

[LeC94b]Baudouin Le Charlier, *Interprétation abstraite*, Notes de Cours, Namur, 1994.

[Ler04] B. S. Lerner, “Verifying process models built using parameterised state machines”, *Proc. ACM SIGSOFT Symp. Software Testing and Analysis*, 2004, 274–284.

[Ler10] 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, Issue 2, March/April 2010, pages 162 - 183.

[Let02a] E. Letier and A. van Lamsweerde, “Agent-Based Tactics for Goal-Oriented Requirements Elaboration”, *Proc. ICSE’02: 24th Intl. Conf. on Soft. Engineering*, Orlando, May 2002.

[Let02b] E. Letier and A. van Lamsweerde, Deriving Operational Software Specifications from System Goals, Proceedings FSE 2002 - 10th International Symposium on the Foundation of Software Engineering, ACM Press, Charleston, November 2002, pp. 200-211.

[Let05] E. Letier, Jeff Kramer, Jeff Magee and Sebastian Uchitel, Monitoring and Control in Scenario-Based Requirements Analysis, Proceedings ICSE 2005 - 27th International Conference on Software Engineering, ACM Press, St. Louis, Missouri, USA, May 2005.

[Lev94] N. Leveson, M. Heimdahl and H. Hildtreth, “Requirements Specification for Process-Control Systems”, IEEE Transactions on Software Engineering, Vol. 20, No. 9, September 1994, 684-706.

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

[Mak01] E. Mäkinen and T. Systä, “MAS – An Interactive Synthesizer to Support Behavioral Modelling in UML”, *Proc. ICSE’01 – Intl. Conf. Soft. Engineering*,, Toronto, Canada, May 2001.

[Man92] Z. Manna and A. Pnueli, “The Temporal Logic of Reactive and Concurrent Systems”, Springer-Verlag, 1992.

[McM93] K.L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, Kluwer, 1993.

[Mil89] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[Mil99] R. Miller and M. Shanahan, “The Event Calculus in Classical Logic – Alternative Axiomatisations”, Linkoping *Electronic Articles in Computer and Information Science*, Vol. 4, No. 16, 1999, pp. 1-27.

[Myl92] J. Mylopoulos, L. Chung and B. Nixon, “Representing and Using Nonfunctional Requirements: A Process-Oriented Approach”, IEEE Transactions on Software Engineering, Vol. 18, No 3/4, 127-155, 1992.

[Nis89] Celso Niskier, T. S. E. Maibaum, Daniel Schwabe: A Pluralistic Knowledge-Based Approach to Software Specification. ESEC 1989: 411-423

[Nus94] Bashar Nuseibeh, Jeff Kramer, Anthony Finkelstein: A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. IEEE Trans. Software Eng. 20(10): 760-773 (1994)

[OMG03]OMG, *UML 2.0 Superstructure Specification*, 2003.

[OMG08]OMG, Business Process Modeling Notation, v1.1, 2008.

[Par95] David Lorge Parnas, Jan Madey: Functional Documents for Computer Systems. Sci. Comput. Program. 25(1): 41-61 (1995)

[Par98] David Y. W. Park, Jens U. Skakkebæk, David L. Dill: Static Analysis to Identify Invariants in RSML Specifications. FTRTFT 1998: 133-142

[Que82] J. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CAESAR", *Proc. 5th International Symposium on Programming*, LNCS 137, 1982.

[Roz08] Anne Rozinat, Wil M. P. van der Aalst: Conformance checking of processes based on monitoring real behavior. Inf. Syst. 33(1): 64-95 (2008).

[Rum91]James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, William E. Lorensen: Object-Oriented Modeling and Design Prentice-Hall 1991.

[Rus95] Stuart J. Russell, Peter Norvig: Artificial intelligence - a modern approach: the intelligent agent book. Prentice Hall 1995: I-XXVIII, 1-932

[Tur49] A. Turing, "Checking a Large Routine", in Report of a Conference on High-Speed Automatic Calculating Machines, University Mathematical Laboratory, 67-69.

[Uch03] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of Behavioral Models from Scenarios", *IEEE Trans. Softw. Engineering*, 29(2), 2003, 99-1.

[Van97] Wil M. P. van der Aalst: Verification of Workflow Nets. ICATPN 1997: 407-426

[Van03] W. M. P. van der Aalst and A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros, "Workflow Patterns", Distributed and Parallel Databases, 14(1), 2003, 5-51.

[Van05] W. M. P. van der Aalst and A. H. M. ter Hofstede. 2005. YAWL: yet another workflow language. *Inf. Syst.* 30, 4 (June 2005), 245-275.

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

[Wis00] A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton, Jr.,"Using Little-JIL to Coordinate Agents in Software Engineering", Automated Software Engineering Conference (ASE 2000), Grenoble, France, pp. 155-163, September 2000.

[Won11a]Peter Y. H. Wong, Jeremy Gibbons: Formalisations and applications of BPMN. Sci. Comput. Program. 76(8): 633-650 (2011)

[Won11b]Peter Y. H. Wong, Jeremy Gibbons: Property specifications for workflow modelling. Sci. Comput. Program. 76(10): 942-967 (2011)

[Yu93] E.S.K. Yu, "Modelling Organizations for Information Systems Requirements Engineering", Proc. RE'93 - 1st Intl Symp. on Requirements Engineering, IEEE, 1993, 34-41.

[Zav97] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering", ACM Transactions on Software Engineering and Methodology, 1997, 1-30.

Annex Proof of Theorems

In Annex A, we will prove the correctness of algorithms presented in Section 3. In Annex B, we will prove the correctness of the algorithm presented in Section 5. The structure of the proof correctness is inspired from [LeC94b]

A. Correctness proof for the LTS decoration algorithms

Fig. 3-4 and Fig 3-7 presented two algorithms for decorating each state of a LTS with state invariants. We will first prove the correctness of the second algorithm in Section A.1. In Section A.2, we will show that the algorithm of Fig. 3-4 is a special case of the latter where the number of fluents is reduced to one.

A.1 Correctness proof for the LTS decoration algorithm for computing most specific invariants

As introduced in Section 3.2.3, the specification of the algorithm is given in Fig. A-1. In the specification, *TraceCond(σ)* denotes the fluent assignment after a trace σ.

GIVEN *a LTS model* (Q, Σ, δ, q_0)
A set Φ *of fluents* $Fl_i = <Init_{Fli}, Term_{Fli}>$ *initially* $Initially_{Fli}$

FIND *a node decoration function*
$deco: Q \rightarrow P(2^{\Phi})$
such that for any state q:

(INV) For any LTS trace σ *reaching q:*
$PathCond(\sigma) \supset deco(q)$

(MSI) For any fluent assignment $v \in 2^{\Phi}$ *such that*
$v \supset deco(q)$,
there exists a LTS trace σ *reaching q such that*
$PathCond(\sigma) = v$

Figure A-1 Specification of the algorithm with disjunctive invariants

As seen in Section 3.2.3, the specification ensures two conditions. The first one states that the decorations are state invariants and the second one states that they are the most specific ones.

The algorithm is recalled in Fig. A-2.

To prove that the algorithm satisfies the above specification, we need to prove two theorems that may be directly derived from the two conditions of the specification, namely,

- the computed invariants are state invariants (Theorem 1.1)
- the computed invariants are the most specific ones (Theorem 1.2).

We need also to prove that the algorithm terminates (Theorem 1.3).

Input: A LTS (Q,Σ,δ,q_0)
A set Φ of fluents $Fl_i = \langle Init_{Fli}, Term_{Fli} \rangle$ initially $Initially_{Fli}$
Output: deco: $Q \rightarrow P(2^{\Phi})$ as defined in Fig. 3-5

$deco(q_0) := \curlywedge_i InitiallyFl_i$
forall $q \in Q \setminus \{q_0\}$ **do** $deco(q) := false$
ToExpl := $\{q_0\}$
while ToExpl $\neq \emptyset$ **do**
    source := getOne(ToExpl)
    ToExpl := ToExpl \ {source}
    forall (target,event) such that target $\in \delta$(source, event) **do**
        newVal := Effect(deco(source), event)
        if not (newVal $\supset$ deco(target)) **then**
            deco(target) := newVal $\vee$ deco(target)
            ToExpl := ToExpl $\cup$ {target}
return deco

Figure A-2 Decoration algorithm with most specific invariants

<u>Theorem 1.1</u>: When the algorithm terminates, for any state *q*, we have that

for any LTS trace σ reaching *q*, $TraceCond(\sigma) \supset deco(q)$

Theorem 1.2: When the algorithm terminates, for any state *q* and for any fluent assignment $v \in 2^{\Phi}$ such that $v \supset deco(q)$,

there exists a LTS trace σ reaching q such that $TraceCond(\sigma) = v$

Theorem 1.3 The algorithm always terminates

For easier understanding, the proofs of the two first theorems will be based on a number of lemmas and corollaries.

Theorem 1.1 states that the decoration of a node should imply every fluent assignment yield by all traces leading to this node. This theorem will be proved by induction on LTS traces. The induction step will rely on Lemma 1.2 and Corollary 1.2, stating that each decoration implies the propagations of their predecessor state decoration through the events between them.

Both theorems will also rely on Lemma 1.1 and Corollary 1.1 that state basic properties of the *Effect* function.

Section A is organized as follows. Lemma 1.1 and Corollary 1.1 are proved in Section A.1.1, Lemma 1.2 and Corollary 1.2 in Section A.1.2, Theorem 1.1 in Section A.1.3, and Theorem 1.2 in Section A.1.4. Finally, the termination of the algorithm will be proved in Section A.1.6.

A.1.1 Proofs of Lemma 1.1 and Corollary 1.1

At each step of the algorithm in Fig. A-2, every state has a decoration belonging to $P(2^{\Phi})$. This decoration implies fluent assignments that are valid in that state. To be more efficient, the propagation rule is applied only to the decoration rather than to the elements it implies.

For example, consider a system with two fluents A and B. Let us assume that the decoration of state q is A, meaning that $A \wedge B$ and $A \wedge \neg B$ are

assignments that are possible at q. If we propagate the decoration of q, the propagation rule will be applied to A rather than to $A \wedge B$ and $A \wedge \neg B$.
We should therefore prove that decoration propagation through an event is equivalent to propagating each fluent assignment implied through that event.

Lemma 1.1 For any x and $val_j \in P(2^{\Phi})$ such that $x = \vee_j val_j$, and for any event e, we have

$$Effect(x, e) = \vee_j Effect(val_j, e).$$

Proof:

By the definition of the propagation of a Boolean expression on fluents through an event, we have (See Section 3.2.1)

$Effect(x, e) = (x[\backslash(\{v_i\} \cup \{v_t\})]) \wedge (\wedge_i v_i) \wedge (\wedge_t \neg v_t)$

(where $\{v_i\}$ is the set of al fluents v_i such that e is among their initiating event, and $\{v_t\}$ is the set of al fluents v_t such that e is among their terminating event.)

By replacing x by $\vee_j val_j$, we get:

$$= (\vee_j val_j[\backslash(\{v_i\} \cup \{v_t\})]) \wedge (\wedge_i v_i) \wedge (\wedge_t \neg v_t)$$

By distribution of the *Drop* operator over disjunctions:

$$= \vee_j (val_j[\backslash(\{v_i\} \cup \{v_t\})]) \wedge (\wedge_i v_i) \wedge (\wedge_t \neg v_t)$$

By distribution of the conjunctions over disjunctions:

$$= \vee_j (val_j[\backslash(\{v_i\} \cup \{v_t\})] \wedge (\wedge_i v_i) \wedge (\wedge_t \neg v_t))$$

$$= \vee_j Effect(val_j, e).$$

From this lemma, we can infer that if an element is implied by a decoration, the propagation of this element through an event is still implied by the propagation of the decoration through that event. In other words, the *Effect* function is monotonous.

Corollary 1.1 For any $x, y \in P(2^{\Phi})$, if $x \supset y$, then for any event e,

$$Effect(x, e) \supset Effect(y, e)$$

Proof

As $x \supset y$, there exists $z \in P(2^{\Phi})$ such that $x \vee z = y$. From Lemma 1.1, we can deduce that:

$$Effect(x, e) \vee Effect(z, e) = Effect(y, e)$$

from which we conclude that $Effect(x, e) \supset Effect(y, e)$.

A.1.2 Proofs of Lemma 1.2 and Corollary 1.2

As explained in the introduction of Annex A, we need to prove that each state decoration implies the propagations of their predecessor state decoration through the events between them. First, we prove that this property is valid at each step of the algorithm for all the states that do not belong to *ToExpl* (Lemma 1.2). Next, the property is proved for all states by noticing that *ToExpl* is empty at the end of the algorithm (Corollary 1.2).

Lemma 1.2: The following loop invariant is preserved by every iteration of the main loop of algorithm A.2:

For every triple (q, q', e) such that

$$q' \in \delta(q, e),\ deco(q) \neq false \text{ and } q \notin ToExpl,$$

we have:

$$Effect((deco(q)), e) \supset deco(q') \qquad (A.1)$$

Proof: by computational induction on iterations of the main loop.

Base

The loop invariant is satisfied for all the transitions from the initial node as the initial state is in *ToExpl*.

It is also satisfied for all the other transitions as their source node is decorated by *false*. The propagation of *false* through any event is always

false, and thus, the formula (A.1) is trivially true because its antecedent is *false.*

Inductive step

Let us assume that a node *q* is selected. It is removed from the *ToExpl* set and its decoration is propagated to its successors *q'*. The loop invariant remains satisfied for the transitions from *q* to all *q'* as

$$deco(q') = Effect\ (deco(q),e) \vee deco(q')$$

after the application of propagation rule in Fig. 3-6.

States whose decoration have changed are added in *ToExpl*; the loop invariant remains satisfied for all outgoing transitions from such states. The other transitions are not affected by such changes; by induction hypothesis, the invariant assertion remains satisfied for these transitions as well.

From this lemma, the following corollary is derived. It states that each state decoration is necessarily higher in the lattice than the propagations of their predecessor state decorations through the events between them.

Corollary 1.2 When the algorithm terminates, we have that:
for every triple *(q, q', e)* such that $q' \in \delta(q, e)$ and $deco(q) \neq false$,

$$Effect(deco(q)),e) \supset deco(q')$$

Proof

Consequence of *Lemma 1.2* and the fact that the set *ToExpl* is empty when the algorithm terminates.

A.1.3 Proof of Theorem 1.1

Now that every lemma necessary for the demonstration are given, let us prove Theorem 1.1.

Theorem 1.1: When the algorithm terminates, for any state q, we have that

for any LTS trace σ reaching q, $TraceCond(\sigma) \supset deco(q)$

Proof: by structural induction on σ.

Base: $\sigma = \varnothing$

$TraceCond(\varnothing) = \curlywedge_i Initially_{Fli}$ with $\curlywedge_i Initially_{Fli} \supset deco(q_0)$

because q_0 was initially decorated by $\curlywedge_i Initially_{Fli}$ and the decoration of q_0 may only go upwards in the lattice.

Induction step: Let us consider $\sigma' = <\sigma, e>$ with σ reaching q and σ' reaching q'

We have that $TraceCond(\sigma) \supset deco(q)$ by induction hypothesis.

From this and using *Corollary 1.1*, we infer that

$Effect(TraceCond(\sigma), e) \supset Effect(deco(q), e)$

We also have $Effect(deco(q), e) \supset deco(q')$ from *Corollary 1.2*.

As $Effect(TraceCond(\sigma), e) = TraceCond(\sigma')$, we have shown that

$TraceCond(\sigma') \supset deco(q')$

A.1.4 Proof of Theorem 1.2

Theorem 1.2.

When the algorithm terminates, for any state q and for any fluent assignment $v \in 2^{\Phi}$ such that $v \supset deco(q)$,

there exists a LTS trace σ reaching q such that $TraceCond(\sigma) = v$

Proof: by induction on iterations on the main loop.

Base

The decoration of the initial state is

$deco(q_0) = \curlywedge_i Initially_{Fli}$.

The only element of 2^{Φ} implied is $\curlywedge_i Initially_{Fli}$ which is the decoration of the empty trace:

$TraceCond(\varnothing) = \curlywedge_i Initially_{Fli}$.

The other states are decorated by the least element (which implies no element in 2^{Φ}).

Induction step

They are two kinds of states:

- The states whose decoration has not changed. By induction hypothesis, the loop invariant remains satisfied for these states.
- The states whose decoration has changed. Let us denote by q the node chosen in *ToExpl*, and e_i the events of its outgoing transitions towards the states q_i.

  The new decorations of states q_i are

  $$prevdeco(q_i) \vee Effect(deco(q), e_i),$$

  where $prevdeco(q_i)$ is the decoration of q_i at the previous iteration.

  Our induction hypothesis states that, for all q_i, we have that for all $v \in 2^{\Phi}$ such that

  $v \supset prevdeco(q_i)$, there exists a LTS trace σ reaching q_i such that $TraceCond(\sigma) = v$.

  We need to verify that this is true for $Effect(deco(q), e_i)$ as well.

  Let us denote by val_j the elements of 2^{Φ} implied by $deco(q)$.

  By induction hypothesis, we have that for all val_j, there exists a LTS trace σ reaching q such that $TraceCond(\sigma) = val_j$.

  Therefore, for each q_i and val_j, there exists a LTS trace

  $$\sigma' = <\sigma, e_i> \text{ (with } \sigma \text{ reaching state } q)$$

  reaching q_i, such that $TraceCond(\sigma') = Effect(val_j, e_i)$.

  As $Effect(deco(q), e_i) = \vee_j Effect(val_j, e_i)$ by Lemma 1.1, the loop invariant is proved.

A.1.6 Proof of Termination

Theorem 1.3 The algorithm terminates

Proof

The algorithm terminates when *ToExpl* is empty. This set will eventually be empty as a state can change its decoration at most 2^{Φ} times. Decorations can only go upwards in the lattice. In the worst case, a state will be decorated by the disjunction of all possible literal combinations, that is, *true*.

A.2 Correctness proof for the LTS decoration algorithm with one fluent only

The specification and algorithm for one fluent only are recalled in Fig. A-3 and Fig A-4 respectively.

GIVEN a LTS model (Q, Σ, δ, q_0)
a *fluent Fl* = < $Init_{Fl}$, $Term_{Fl}$ > *initially* $Initially_{Fl}$,

FIND *a node decoration function*
$Q \rightarrow Bool_{abs}$
defined as follows:
deco (q) =
'true' iff *Fl* is true for all LTS traces reaching state *q*
'false' iff *Fl* is false for all LTS traces reaching state *q*
'top' iff *Fl* is true for some LTS traces reaching state *q* and false for others
'bottom' iff state *q* is not reachable from the initial state

Figure A-3 Specification of algorithm with one fluent only

By comparing the specification with the specification in Fig. A-1, we can see that the specifications are equivalent in the particular case where the number of fluents is reduced to one:

- In both cases, even if the vocabulary is different, the state decorations may have four possible values. In the algorithm of Fig A-2, these values are *{false, Fl, ¬Fl, true}* which correspond to those of Fig. A-4, namely *{bottom, true, false, top}*.The lattice structure is thus the same in both cases.

- In Fig. A-3, the postcondition corresponds to the postcondition of Fig A-1. The specification of Fig A-3 states that the decoration should be a state invariant and the most specific one, e.g., the value 'top' may only be given if there exists a trace reaching q rendering the fluent true and a trace reaching q rendering the fluent false.

By comparing the two algorithms, we can see that (in case of one single fluent) they amount to each other even though the propagation rules seem different. We need to show that the *Effect* function with one fluent only has the same result as the propagation rule in Fig. 3-2. This appears trivial by considering the possible types of event - an initiating event, a terminating event or an event that does not belong to the initiating or terminating events.

Input: A LTS (Q,Σ,δ,q_0)
A Fluent Fl = $<Init_{Fl}, Term_{Fl}>$ initially $Initially_{Fl}$
Output: deco: $Q \rightarrow Bool_{abs}$ as defined in Fig 3.1.

$deco(q_0) \leftarrow Initially_{Fl}$
forall $q \in Q \setminus \{q_0\}$ **do** $deco(q) \leftarrow bottom$
$ToExpl \leftarrow \{q_0\}$
while $ToExpl \neq \emptyset$ **do**
    source ← getOne(ToExpl)
    ToExpl ← ToExpl \ {source}
    forall (target,event) such that target ∈ δ(source, event) **do**
        if event ∈ $Init_{Fl}$ **then**
            newVal ← true
        else
            if event ∈ $Term_{Fl}$ **then**
                newVal ← false
            else
                newVal ← deco(source)
        if not (newVal ≤ deco(target)) **then**
            deco(target) ← sup(newVal, deco(target))
            ToExpl ← ToExpl ∪ {target}
return deco

Figure A-4 Algorithm with one fluent only

B. Correctness proof for the g-LTS decoration algorithm

As introduced in Section 5.4, the specification of the algorithm is given in Fig. B-1. In the specification, *PlValue(γ)* denotes the placeholder value after a g-LTS execution γ.

GIVEN *a guarded LTS* $(Q, \Sigma, \Phi, \Theta, \delta, q_0, C_0)$
a placeholder domain PlDomain
a finite placeholder domain $PlFiniteDomain \subseteq PlDomain$
a propagate function $PlDomain \times \Sigma \rightarrow P(PlDomain)$ *such that*
for any $Pl \in PlDomain \setminus PlFiniteDomain$ *and* $e \in \Sigma$*:*
$propagate(Pl,e) \cap PlFiniteDomain = \varnothing$.
the value of the placeholder after an empty execution $p_0 \in PlFiniteDomain$

FIND *a node decoration function*
$deco: Q \rightarrow (2^{\Phi_+ \Theta} \rightarrow P(FinitePlDomain \cup \{Out\_Of\_Bonds\}))$
such that for any state q:
(INV1) *for any g-LTS execution γ reaching q,*
if $PlValue(\gamma) \in PlFiniteDomain$ *then* $PlValue(\gamma) \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$
(INV2) *for any g-LTS execution γ reaching q,*
if $PlValue(\gamma) \notin PlFiniteDomain$ *then* $Out\_Of\_Bounds \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$
(MSM1) *for any* $Pl \in PlDomain$ *such that* $Pl \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$,
there exists a g-LTS execution γ reaching q such that: $PlValue(\gamma) = Pl$
(MSM2) *if* $Out\_Of\_Bounds \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$,
then there exists an execution γ reaching q such that
$PlValue(\gamma) \notin PlFiniteDomain$

Figure B-1 Specification of the g-LTS decoration algorithm

At each state, the algorithm computes placeholder values for each variable assignment. These placeholder values must meet the following conditions:

- Each placeholder value yield by a g-LTS execution reaching state q should belong to the computed decoration of q, i.e. for all executions γ reaching a state q,
    - (INV1) if *PlValue(γ)* belongs to the finite domain of placeholders then it belongs to the codomain of *deco(q)*.
    - (INV2) otherwise *Out_Of_Bonds* belongs to the codomain of *deco(q)*.
- The computed placeholder values for a state q should be produced by at least one execution reaching q, i.e.,

- (MSM1) Each placeholder value in the codomain of *deco(q)* should be a placeholder value produced by at least one execution reaching *q*.
- (MSM2) If *Out_Of_Bounds* belongs to the codomain of *deco(q)*, there exists at least one execution yielding a placeholder value outside the placeholder finite domain.

The algorithm is recalled in Fig. B-2. To prove that the algorithm satisfies the above specification, we need to prove four theorems that may be directly derived from the four conditions of the specification.

We need also to prove that the algorithm terminates (Theorem 2.5).

Theorem 2.1: When the algorithm terminates, for any state *q*, we have that for any g-LTS execution γ reaching *q*,

if $PlValue(\gamma) \in PlFiniteDomain$

then $PlValue(\gamma) \in \cup\{valueSet \mid valueSet \in ran_{deco(q)}\}$

Theorem 2.2: When the algorithm terminates, for any state *q*, we have that for any g-LTS execution γ reaching *q*,

if $PlValue(\gamma) \notin PlFiniteDomain$

then $Out\_Of\_Bounds \in \cup\{valueSet \mid valueSet \in ran_{deco(q)}\}$

Theorem 2.3: When the algorithm terminates, for any state *q* and for any $Pl \in PlDomain$ such that $Pl \in \cup\{valueSet \mid valueSet \in ran_{deco(q)}\}$,

there exists a g-LTS execution γ reaching *q* such that:

$$PlValue(\gamma) = Pl$$

Theorem 2.4*:* When the algorithm terminates, for any state *q*,

if *Out_Of_Bounds* $\in \cup\{valueSet \mid valueSet \in ran_{deco(q)}\}$,

then there exists an execution γ reaching *q* such that

PlValue*(γ) $\notin$ *PlFiniteDomain

Theorem 2.5: The algorithm always terminates

Input: A g-LTS $(Q,\Sigma,\Phi,\Theta,\delta,q_0,C_0)$
A placeholder domain PlDomain
A finite placeholder subdomain PlFiniteDomain $\subseteq$ PlDomain
A propagate function PlDomain x $\Sigma \rightarrow$ P(PlDomain)
The value for the placeholder after an empty path p_0 ($p_0 \in$ PlFiniteDomain)
Output: deco: $Q \rightarrow (2^{\Phi+\Theta} \rightarrow (P(PlFiniteDomain) \cup \{Out\_Of\_Bonds\}))$

```
deco(q0) := { x ↦ {p0} | x ∈ C0};
forall q ∈ Q \ {q0} do
        deco(q) := ∅;
ToExpl := {q0};
while ToExpl ≠ ∅ do
        source := getOne(ToExpl);
        ToExpl := ToExpl \ {source};
        forall (target,label) such that target ∈ δ do
                if (label ∈ 2^(Φ+Θ))
                        newVal := {label} ◁ deco(source) ;
                else
                        newVal := ⋁{(x) ↦ finite_propagate(deco(source)(y),label))
                                        | y ∈ dom_deco(source) and x |= Effect(y,label) and x ∈ 2^(Φ+Θ)};
                if not (newVal ≤ deco(target)) then
                        deco(target):= newVal ∨ deco(target)
                        ToExpl := ToExpl ∪ {target};
return deco;

finite_propagate(set, e)=
        toreturn := ∅;
        forall x ∈ set
                if (x == Out_Of_Bonds) toreturn := toreturn ∪ {Out_Of_Bonds};
                else    val := propagate(x,e);
                        if (val \ PlFiniteDomain ≠ ∅)
                                val := (val ∩ PlFiniteDomain) ∪ {Out_Of_Bonds};
                toreturn := toreturn ∪ {val};
        return toreturn;
```

Figure B-2 Generic algorithm for g-LTS decoration

For easier understanding, the proofs of the first four theorems will be based on a number of lemmas and corollaries.

The structure of the proofs will be similar to the ones of the Annex A. However they are much more complex as:

- the transitions of a guarded hMSC are labeled by guards or events. The propagation rules in presence of events or guards are completely different. When dealing with propagation through a transition, the proofs must tackle these two different cases.
- the placeholder value for an execution may be inside or outside *PlFiniteDomain*. These two cases must be considered separately as well.
- the decoration type is much more complex. Rather than decorating the transition system with Boolean expression on variables, the algorithm decorates it with mappings associating assignment on variables to set of placeholder values.

The theorems do not mention explicitly the domain of the decoration function. As we have seen in Chapter 5, this domain (assignment of variables) is important in order to accurately propagate decorations through guards. For this reason, we will first prove lemmas stronger than the introduced theorems that state properties about the mapping rather than only properties about placeholder values. Lemma 2.8 states a stronger property than Theorem 2.1, Lemma 2.9 than Theorem 2.2, and Lemma 2.10 than Theorem 2.3 and Theorem 2.4.

Lemma 2.8 and 2.9 will be proved by induction on g-LTS executions. These theorems will rely on a set of lemmas and corollaries (Lemma 2.4, Corollary 2.5, Lemma 2.6, Corollary 2.7) stating that each decoration implies the propagations of their predecessor state decoration through the events and guards between them.

All theorems will also rely on basic properties of the propagation through an event (Lemma 2.1) and of the *finite_propagate* function (Lemma 2.2 and 2.3).

B.1 Proofs of Lemma 2.1

For simplicity, in the following, we define the function

$$propagateThroughEvent : deco \times \Sigma \rightarrow deco$$

as follows

$$propagateThroughEvent(deco, label) =$$
$$\vee\{(x) \mapsto finite\_propagate(deco(y), label)) \mid y \in dom_{deco} \text{ and } x \models Effect(y, label) \text{ and } x \in 2^{\Phi+\Theta}\}$$

The first lemma we need to prove states that the *propagateThroughEvent* function is monotonous.

Lemma 2.1 For any $f, g: 2^{\Phi+\Theta} \rightarrow P(PlDomain)$, if $f \leq g$ then for any event e,

$$propagateThroughEvent(f, e) \leq propagateThroughEvent(g, e)$$

Proof

By definition, f ≤ g means that $dom_f \subseteq dom_g$ and $\forall x \in dom_f, f(x) \subseteq g(x)$

We must therefore prove two properties

$$\cup_{y \in domf} \{x \mid x \models Effect(y,e)\} \subseteq \cup_{y \in domg} \{x \mid x \models Effect(y,e)\} \quad (1)$$

$$\text{for all } x \in dom_f, finite\_propagate(f(x)) \subseteq finite\_propagate(g(x)) \quad (2)$$

(1) is satisfied as $dom_f \subseteq dom_g$

(2) is satisfied as $f(x) \subseteq g(x)$

B.2 Proofs of Lemma 2.2 and Lemma 2.3

The two following lemmas describe the links between the *propagate* function given by the user and the *finite_propagate* function used in the algorithm. The proofs are not given but are quite trivial.

Lemma 2.2

For all $valueSet \subseteq P(PlFiniteDomain)$ and $e \in \Sigma$,

If for each $x \in valueSet$, $propagate(x,e) \subseteq PlFiniteDomain$ then

$$\cup_{x \in valueSet} propagate(x, e) = finite\_propagate(set, e)$$

Lemma 2.3

For all $valueSet \subseteq P(PlFiniteDomain)$ and $e \in \Sigma$,

If there exists x ∈ valueSet such that $propagate(x,e) \not\subset PlFiniteDomain$ then

$$finite\_propagate(set, e) =$$

$$((\cup_{x \in valueSet} propagate(x, e)) \cap PlFiniteDomain) \cup \{Out\_Of\_Bounds\}$$

B.3 Proofs of Lemma 2.4 and Corollary 2.5

As explained in the introduction of Annex B, we need to prove that each state decoration is higher in the lattice than the propagations of their predecessor state decoration through the events between them.

First, we prove that this property is valid at each step of the algorithm for all states that do not belong to *ToExpl* (Lemma 2.4). Next, the property is proved for all states by noticing that *ToExpl* is empty at the end of the algorithm (Corollary 2.5).

Lemma 2.4

The following loop invariant is preserved by every iteration of the main loop of algorithm A.2:

For every triple (*q, q', label*) such that

$q' \in \delta(q, label), label \in \Sigma, deco(q) \neq \{\}$ and $q \notin ToExpl,$

we have:

$propagateThroughEvent\ (deco(q), label) \leq deco(q')$

Proof: by computational induction on iterations of the main loop.

Base

The loop invariant is satisfied for all the transitions from the initial node as the initial state is in *ToExpl*.

It is also satisfied for all the other transitions as their source node is decorated by *{}*.

Inductive step

Let us assume that a node *q* is selected. It is removed from the *ToExpl* set and its decoration is propagated to its successors *q'*. *newVal* gets the value *propagateThroughEvent (deco(source), label)*.

deco(q') gets the following value :

$deco(q') := newVal \vee deco(q');$

The new value of deco(q') is equal to the supremum of its old value and newVal; its new value is therefore higher in the lattice than *newVal*. The loop invariant remains therefore satisfied for the transitions from *q* to all *q'*.

States whose decoration have changed are added in *ToExpl*; the loop invariant remains satisfied for all outgoing transitions from such states. The other transitions are not affected by such changes; by induction hypothesis, the invariant assertion remains satisfied for these transitions as well.

Corollary 2.5 When the algorithm terminates, we have that:

for every triple *(q, q', label)* such that

$q' \in \delta(q, label), label \in \Sigma, deco(q) \neq \{\}$

$propagateThroughEvent\ (deco(q), label) \leq deco(q')$

Proof

Consequence of Lemma 2.4 and the fact that the set *ToExpl* is empty when the algorithm terminates.

B.4 Proofs of Lemma 2.6 and Corollary 2.7

As explained in the introduction of Annex B, we need to prove that each state decoration implies the propagations of their predecessor state decoration through the guards between them.

First, we prove that this property is valid at each step of the algorithm for all the states that do not belong to *ToExpl* (Lemma 2.6). Next, the property is proved for all states by noticing that *ToExpl* is empty at the end of the algorithm (Corollary 2.7).

Lemma 2.6

The following loop invariant is preserved by every iteration of the main loop of algorithm B.2:

For every triple (*q, q', label*) such that

$q' \in \delta(q, label), label \in 2^{\Phi+\Theta}, deco(q) \neq \{\}$ and $q \notin ToExpl,$

we have:

$(\{label\} \triangleleft deco(q)) \leq deco(q')$

Proof: by computational induction on iterations of the main loop.

Base

The loop invariant is satisfied for all the transitions from the initial node as the initial state is in *ToExpl*.

It is also satisfied for all the other transitions as their source node is decorated by *{}*.

Inductive step

Let us assume that a node *q* is selected. It is removed from the *ToExpl* set and its decoration is propagated to its successors *q'*. *newVal* gets the value

$newVal := \{label\} \lhd deco(source)$.

deco(q') gets the following value:

$deco(q') := newVal \vee deco(q')$.

The new value of deco(q') is equal to the supremum of its old value and newVal; its new value is therefore higher in the lattice than *newVal*. The loop invariant remains therefore satisfied for the transitions from *q* to all *q'*.

States whose decoration have changed are added in *ToExpl*; the loop invariant remains satisfied for all outgoing transitions from such states. The other transitions are not affected by such changes; by induction hypothesis, the invariant assertion remains satisfied for these transitions as well.

Corollary 2.7 When the algorithm terminates, we have that:

for every triple *(q, q', label)* such that

$q' \in \delta(q, label), label \in 2^{\Phi+\Theta}, deco(q) \neq \{\}$

$(\{label\} \lhd deco(q)) \leq deco(q')$

Proof

Consequence of Lemma 2.6 and the fact that the set *ToExpl* is empty when the algorithm terminates.

B.5 Proofs of Lemma 2.8

Lemma 2.8 states a property stronger than Theorem 2.1. The lemma states a property about the mapping rather than about placeholder values.

Lemma 2.8

When the algorithm terminates, for any state q, we have that

for any LTS trace γ reaching q,

if $PlValue(\gamma) \in PlFiniteDomain$,

then $(ExecGCond\ (\gamma) \mapsto PlValue(\gamma)) \leq deco(q)$

Proof: by structural induction on γ.

Base: γ is an empty g-LTS execution.

$ExecGCond\ (\gamma) \supset C_0$ and $PlValue(\gamma) = p_0$.

We have:

$$\{ x \mapsto \{p_0\} \mid x \in C_0\} \leq deco(q_0)$$

because q_0 was initially decorated by $\{ x \mapsto \{p_0\} \mid x \in C_0\}$ and the decoration of q_0 may only go upwards in the lattice.

Induction step: Let us consider $\gamma' = (Init, \langle \sigma, l \rangle)$ where σ was the label sequence of γ: $\gamma = (Init, \sigma)$. γ reaches q and γ' reaches q'.

If l is an event

We have that $(ExecGCond\ (\gamma) \mapsto PlValue(\gamma)) \leq deco(q)$ by induction hypothesis.

By Lemma 2.1, we have

$$propagateThroughEvent\ (ExecGCond\ (\gamma) \mapsto PlValue(\gamma), l) \leq propagateThroughEvent\ (deco(q), l)$$

By Corollary 2.5, we have also

$$propagateThroughEvent\ (deco(q), l) \leq deco(q')$$

As defined in Section 5.2, we have also that:

$$ExecGCond\ (\gamma') = x$$

where $x \models Effect(ExecGCond(\gamma, l)$ and $x \in 2^{\Phi+\Theta}$ *(B.1)*

and as defined in Section 5.1:

$PlValue(\gamma') = p$

where $p \in propagate(PlValue(\gamma), l)$ *(B.2)*

By Lemma 2.2, we have also that as

$PlValue(\gamma') \in PlFiniteDomain,$

we have

$PlValue(\gamma') \in finite\_propagate(\{PlValue(\gamma)\}, l).$ *(B.3)*

By definition of the partial order operator (see Section 5.3) and by (B.1, B.2, B.3), for all x and p satisfying (B.1, B.2), we have

$(ExecGCond(\gamma') \mapsto PlValue(\gamma')) \leq$

$propagateThroughEvent(ExecGCond(\gamma) \mapsto PlValue(\gamma), l)$

We have therefore shown that

$(ExecGCond(\gamma') \mapsto PlValue(\gamma')) \leq deco(q')$

If l is a guard

We have that

$(ExecGCond(\gamma) \mapsto PlValue(\gamma)) \leq deco(q)$

by induction hypothesis.

As defined in Section 5.1 and 5.2, we have that

$ExecGCond(Init, <\sigma, l>) = ExecGCond(Init, \sigma)$ and

$PlValue(Init, <\sigma, l>) = PlValue(Init, \sigma).$

As $(Init, <\sigma, l>)$ is a valid g-LTS execution, we know that

$ExecGCond(Init, \sigma) \models l$ (see Section 4.2) .

Therefore

$(ExecGCond(\gamma) \mapsto PlValue(\gamma)) \leq \{l\} \lhd deco(q).$

Using Corollary 2.7, we have shown that

$(ExecGCond(\gamma') \mapsto PlValue(\gamma')) \leq deco(q').$

B.6 Proof of Theorem 2.1

Theorem 2.1: When the algorithm terminates, for any state q, we have that for any g-LTS execution γ reaching q,

if $PlValue(\gamma) \in PlFiniteDomain$

then $PlValue(\gamma) \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$

Proof

Consequence of Lemma 2.8

B.7 Proof of Lemma 2.9

Lemma 2.9 states a property stronger than Theorem 2.2. The lemma states a property about the mapping rather than about placeholder values.

Lemma 2.9 : When the algorithm terminates, for any state q, we have that for any g-LTS execution γ reaching q,

if $PlValue(\gamma) \notin PlFiniteDomain$

then $(ExecGCond(\gamma) \rightarrow \{Out\_Of\_Bounds\}) \leq deco(q)$

Proof by structural induction on γ.

Base

The base case considers g-LTS executions $\gamma' = (Init, <l_1, \ldots, l_{i-1}, l_i>)$ such that

$PlValue(Init, < l_1, \ldots, l_{i-1} >) \in PlFiniteDomain$ and

$PlValue(\gamma') \notin PlFiniteDomain$.

Let us denote by γ the g-LTS execution $(Init, < l_1, \ldots, l_{i-1} >)$; γ reaches q, while γ' reaches q'.

We know by Lemma 2.8 that

$$(ExecGCond(\gamma) \mapsto PlValue(\gamma)) \leq deco(q).$$

l_i is an event as a guard does not change the value of the placeholder.

By Lemma 2.1,

propagateThroughEvent ((ExecGCond (γ)↦PlValue(γ)),l_i)

≤ propagateThroughEvent (deco(q), l_i)

where *propagateThroughEvent (deco, label) =*
∨{(x) ↦ finite_propagate(deco(y), label))
| y ∈ dom_{deco} and x |=Effect (y, label) and x ∈ $2^{\Phi+\Theta}$};

As *propagate(PlValue(γ), l_i)) ⊄ PlFiniteDomain*, we know by Lemma 1.3 that

Out_Of_Bounds ∈ finite_propagate(PlValue(γ'), l_i)).

Using Lemma 2.5 and as

ExecGCond(γ') |=Effect(ExecGCond(γ), l_i) (see Section 5.2),

we have shown that

(ExecGCond(γ')→{Out_Of_Bounds}) ≤deco(q').

Induction Step

Lets us denote by *γ'= (Init,<l_1,..., l_i>)* such that *PlValue(γ') ∉ PlFiniteDomain. (γ'* reaches *q')*

By induction hypothesis, we have that

(ExecGCond(γ')→{Out_Of_Bounds}) ≤deco(q')

Let *γ''= (Init,<l_1,..., l_{i+1}>)* reaching *q''*. We need to check that

(ExecGCond(γ'')→{Out_Of_Bounds}) ≤deco(q'').

There are two cases to be considered: l_{i+1} can be an event or a guard.

l_{i+1} is an event

By Lemma 2.1, we have

propagateThroughEvent ((ExecGCond (γ')↦{Out_Of_Bounds}, l_{i+1})

≤ propagateThroughEvent (deco(q'), l_{i+1})

As *propagate(Out_Of_Bounds, l_{i+1}))⊄ PlFiniteDomain,* we know by Lemma 2.3 that

Out_Of_Bounds ∈ finite_propagate(PlValue(γ'), l_{i+1})).

Using Lemma 2.5 and as

ExecGCond(γ'') |=Effect(ExecGCond(γ'), l_{i+1}) (see Section 5.2),

we have that

$(ExecGCond(\gamma') \rightarrow \{Out\_Of\_Bounds\}) \leq deco(q'')$

l_{i+1} *is a guard*

We have that $(ExecGCond(\gamma') \rightarrow \{Out\_Of\_Bounds\}) \leq deco(q')$ by induction hypothesis.

We have that

$ExecGCond(\gamma') = ExecGCond(\gamma')$ and

$PlValue(\gamma') = PlValue(\gamma')$.

As $\gamma' = (Init, <l_1, ..., l_{i+1}>)$ is a valid g-LTS execution, we know that

$ExecGCond(\gamma') \models l_{i+1}$ (see Section 4.2) .

Therefore,

$(ExecGCond(\gamma') \mapsto PlValue(\gamma')) \leq \{l\} \lhd deco(q')$

Using Corollary 2.7, we have shown that

$(ExecGCond(\gamma') \mapsto \{Out\_Of\_Bounds\}) \leq deco(q'')$

B.8 Proof of theorem 2.2

Theorem 2.2: When the algorithm terminates, for any state q, we have that for any g-LTS execution γ reaching q,

if $PlValue(\gamma) \notin PlFiniteDomain$

then $Out\_Of\_Bounds \in \cup \{valueSet \mid valueSet \in ran_{deco(q)}\}$

Proof

Consequence of Lemma 2.9

B.9 Proof of Lemma 2.10

Lemma 2.10 states a property stronger that the Theorem 2.3 and Theorem 2.4. The lemma states a property about the mapping rather than about placeholder values.

Lemma 2.10 *:* When the algorithm terminates, for any state q and for any assignment $Assi \in 2^{\Phi+\Theta}$ and any Pl such that

$$(Assi \rightarrow \{Pl\}) \leq deco(q)$$

there exists a g-LTS execution γ reaching q such that:

$ExecGCond(\gamma)=Assi$ and $PlValue(\gamma) = Pl$ if $Pl \neq Out\_Of\_Bounds$

$ExecGCond(\gamma)=Assi$ and $PlValue(\gamma) \notin PlFiniteDomain$

if $Pl = Out\_Of\_Bounds$

Proof: by induction on iterations on the main loop.

Base

The decoration of the initial state is

$$deco(q_0) = \{ x \mapsto \{p_0\} \mid x \in C_0\}$$

There exists executions for each mappings lower in the lattice than $deco(q_0)$. These executions are empty executions (with no labels) with all different initial variable assignments meeting the initial condition C_0.

The other states are decorated by the empty set which has no least element in the lattice.

Induction step

They are two kinds of states:

- The states whose decoration has not changed. By induction hypothesis, the loop invariant remains satisfied for these states.
- The states whose decoration has changed. Let us denote by q the node chosen in *ToExpl*, and l_i the events of its outgoing transitions towards the states q_i.

  The new decorations of states q_i are

  $$prevdeco(q_i) \vee newVal$$

  where $prevdeco(q_i)$ is the decoration of q_i at the previous iteration.

Our induction hypothesis implies that there exists a g-LTS execution for each mapping lower in the lattice than $prevdeco(q_i)$.
We need to verify that this is true for *newVal* as well.

If l_i *is an event*

$newVal = propagateThroughEvent\ (deco, label) =$
$\vee\{(x) \mapsto finite\_propagate(deco(y), label))$
$\quad | \ y \in dom_{deco}\ and\ x \models Effect\ (y, label)\ and\ x \in 2^{\Phi+\Theta}\};$

By induction hypothesis, we have that for all val_j and Pl_i such that

$(val_j \rightarrow Pl_i) \leq deco(q)$,

there exists a g-LTS execution γ reaching q such that $ExecGCond(\gamma) = val_j$ and

$PlValue(\gamma) = Pl_i$ if $Pl_i \neq Out\_Of\_Bounds$

$PlValue(\gamma) \notin PlFiniteDomain$ if $Pl_i = Out\_Of\_Bounds$

Case 1: Let us consider first the mappings such that $Pl_i \neq Out\_Of\_Bounds$.
Therefore, for each Pl_i and val_j, there exists a set of g-LTS execution $\{\gamma'\}$

$\gamma' = (Init, <\sigma, l_i>)$ (with σ reaching state q)

reaching q_i, such that

$ExecGCond\ (\gamma') \in Effect\ (val_j, l_i)$ and

$PlValue(\gamma') \in finite\_propagate\ (val_j, l_i)$.

if $propagate(val_j, l_i) \in PlFiniteDomain$:

$\quad finite\_propagate(val_j, l_i) = propagate(val_j, l_i)$.

if $propagate(val_j, l_i) \notin PlFiniteDomain$:

$\quad Out\_Of\_Bounds \in finite\_propagate(val_j, l_i)$.

The loop invariant is therefore proved(when l_i is an event in the first case).

Case 2: Let us consider now the mappings such that $Pl_i = Out\_Of\_Bounds$.
By induction hypothesis, we have that for all val_j such that

$(val_j \rightarrow \{Out\_Of\_Bounds\}) \leq deco(q)$,

there exists a g-LTS execution γ reaching q such that $ExecGCond(\gamma) = val_j$ and $PlValue(\gamma) \notin PlFiniteDomain$.

Therefore, for each val_j, there exists a set of g-LTS execution $\{\gamma'\}$

$\gamma' = (Init, <\sigma, l_i>)$ (with σ reaching state q)

reaching q_i, such that

$ExecGCond(\gamma') \in Effect(val_j, l_i)$ and

$PlValue(\gamma') \in finite\_propagate(\{Out\_Of\_Bounds\}, l_i)$.

As $finite\_propagate(Out\_Of\_Bounds, l_i) = Out\_Of\_Bounds$, the loop invariant is proved (when l_i is an event)

if l_i is a guard

$newVal := \{l_i\} \lhd deco(source)$;

case1 : $Pl_i \neq Out\_Of\_Bounds$

By induction hypothesis, we have that for all val_j and Pl_i,

$(val_j \rightarrow Pl_i) \leq deco(source)$ with $Pl_i \neq Out\_Of\_Bounds$,

there exists a LTS trace γ reaching q such that

$ExecGCond(\gamma) = val_j$ and $PlValue(\gamma) = Pl_i$.

Therefore, for each $val_j | = l_i$ and Pl_i, there exists a g-LTS execution

$\gamma' = (Init, <\sigma, l_i>)$ (with σ reaching state q)

reaching q_i, such that $ExecGCond(\gamma') = val_j$ and $PlValue(\gamma') = Pl_i$.

The loop invariant is therefore proved.

Case 2 : $Pl_j = Out\_Of\_Bounds$

As the guards doe not have impact on placeholder values, the second case can be proved in a similar way as the first one.

B.10 Proof of Theorem 2.3

Theorem 2.3: When the algorithm terminates, for any state q and for any $Pl \in PlDomain$ such that $Pl \in \cup\{valueSet \mid valueSet \in ran_{deco(q)}\}$,

there exists a g-LTS execution γ reaching q such that:

$$PlValue(\gamma) = Pl$$

Proof

Consequence of Lemma 2.10

B.11 Proof of Theorem 2.4

Theorem 2.4: When the algorithm terminates, for any state q,

if $Out\_Of\_Bounds \in \cup\{valueSet \mid valueSet \in ran_{deco(q)}\}$,

then there exists an execution γ reaching q such that

$$PlValue(\gamma) \notin PlFiniteDomain$$

Proof

Consequence of Lemma 2.10

B.12 Proof of Termination

Theorem 2.5 The algorithm terminates

Proof

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