Analyzing Critical Process Models through Behavior Model Synthesis

Christophe Damas[1], Bernard Lambeau[1], François Roucoux[2] and Axel van Lamsweerde[1]

[1]*Département d'Ingénierie Informatique*
Université catholique de Louvain (UCL)
{damas, blambeau, avl}@info.ucl.ac.be

[2]*Radiothérapie Expérimentale (IMRE)*
Faculté de Médecine (UCL)
francois.roucoux@uclouvain.be

Abstract

Process models capture tasks performed by agents together with their control flow. Building and analyzing such models is important but difficult in certain areas such as safety-critical healthcare processes. Tool-supported techniques are needed to find and correct flaws in such processes. On another hand, event-based formalisms such as Labeled Transition Systems (LTS) prove effective for analyzing agent behaviors.

The paper describes a blend of state-based and event-based techniques for analyzing task models involving decisions. The input models are specified as guarded high-level message sequence charts, a language allowing us to integrate material provided by stakeholders such as multi-agent scenarios, decision trees, and flowchart fragments. The input models are compiled into guarded LTS, where transition guards on fluents support the integration of state-based and event-based analysis. The techniques supported by our tool include model checking against process-specific properties, invariant generation, and the detection of incompleteness, unreachability, and undesirable non-determinism in process decisions. They are based on a trace semantics of process models, defined in terms of guarded LTS, which are in turn defined in terms of pure LTS. The techniques complement our previous palette for synthesizing behavior models from scenarios and goals.

The paper also describes our preliminary experience in analyzing cancer treatment processes using these techniques.

1. Introduction

For effective software support, real-world processes should be captured by adequate models [6]. Process and workflow modeling languages have therefore flourished, e.g., UML Activity Diagrams [20], BPMN [21], and Little-Jil [3], to cite a few languages in use.

Building adequate, complete, and consistent process models is not necessarily an easy task. Flawed models might not be a big concern in business workflow systems. They can however have dreadful consequences in safety-critical areas such as medical processes. Techniques should therefore be available for systematically detecting and fixing severe flaws.

Efforts were recently made to adapt verification technology [1] to process models. Typically, a state machine model is derived from the input model and then checked against properties. For example, structural consistency constraints on UML activity diagrams can be checked using the NuSMV model checker [7]. Similar constraints can be verified on Little-JIL process models [17], after task conversion into LTS, using LTSA [18]. LTSA was also used for deadlock analysis and model-checking of workflow schemas represented in FSP/LTS [14]. In those papers, decision nodes are not handled. The checked properties are event-based and refer to events associated with task performance. They are not process-specific in [7, 17].

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 (a) therapy *scenarios* involving interacting agents with multiple exceptions, (b) *decision* trees, (c) goals and properties on *state* variables about patients, and, (d) sequencing of *phases* composed of tasks. Our observations find confirmations in the literature on medical workflows, e.g., [13, 8, 10].

As we did not find any process language supporting those features together, we have extended the language of high-level Message Sequence Charts (hMSC) [12], satisfying requirements (a) and (d), with guards on fluents, to meet requirements (b) and (c). Beyond smaller conceptual distance from stakeholders, our choice was motivated by the prospect for a clear formal semantics and the availability of related formal techniques [9, 22], including our techniques and toolkit for synthesizing annotated behavior models from scenarios and goals [4, 5].

The paper introduces guarded hMSC as a process modeling language and guarded LTS (g-LTS) as an intermediate language, used for a variety of analyses, upwards the LTS language used for model-checking and animation. The integration of event-based and state-based specification styles is achieved by letting guards refer to fluents [9]. 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. An algorithm is provided for generating trace-equivalent g-LTS and LTS models from a guarded hMSC process model.

The paper then presents tool-supported analysis techniques applicable to our process models, namely, model checking against state-based properties from a goal model; state invariant generation; and guard analysis. The latter includes checking that the guards on alternatives at some decision point in a task flow cover all possibilities (no missing branch), do not overlap (deterministic decisions), and are all satisfiable (reachability of subsequent tasks).

Finally, we discuss our experience in applying those techniques to the analysis of a safety-critical medical process, the clinical pathway for rectal cancer.

The paper is organized as follows. Section 2 provides some necessary background on scenario specifications, LTS, and fluents. Section 3 introduces guarded hMSCs for process modeling whereas Section 4 defines the trace semantics of this language in terms of g-LTS. Section 5 describes our algorithm for generating the set of event traces admitted by a g-LTS, yielding the equivalent LTS. Section 6 describes the analyses we can perform on process models in guarded hMSC. Section 7 shows our approach in action for clinical pathway analysis.

2. Background

This section introduces some necessary rudiments on MSCs, hMSCs, LTS, and fluents.

2.1 High-level message sequence charts

Message sequence charts (MSCs) are commonly used for capturing multi-agent scenarios [12]. A MSC is composed of vertical timelines associated with agent instances and horizontal arrows representing interactions among them. *Agents* are active components of a system; they define the system's scope and control system behaviors. A timeline label declares the class of the corresponding agent instance. An arrow label indicates some interaction event among the source and target agent instances; the event is synchronously controlled by the source and monitored by the target.

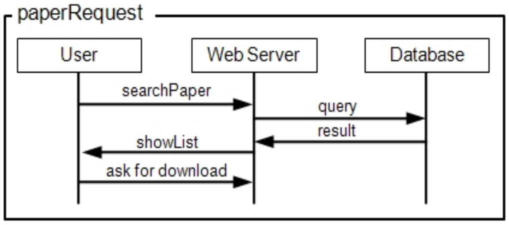

Figure 1: MSC scenario for paperRequest task

Fig. 1 shows a MSC scenario of users searching for papers on the web to ask for download. The system is composed of three agents: a user, a website server, and a database agent. (For clarity of presentation, our running example is kept simple and non-medical.)

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

2.2 Labeled transition systems

A *labeled transition system* (LTS) is an automaton defined by a structure (Q,Σ,δ,q_0), where Q is a finite set of states, Σ is a set of event labels, $\delta \subseteq Q\times\Sigma\times Q$ is a labeled transition relation, and q_0 is an initial state [18].

A system is behaviorally modeled by a parallel composition of LTS models – one for each agent. The LTS being composed behave asynchronously but synchronize on shared events.

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

The semantics of MSCs and hMSCs is defined in terms of LTS and parallel composition [22]. A MSC timeline defines a finite LTS trace capturing the behavior of the corresponding agent instance. The semantics of an entire MSC is similarly defined as a trace of the system LTS; MSCs yield traces of this LTS.

2.3 Fluents

A fluent Fl is a proposition defined by a set I_{Fl} of initiating events, a set T_{Fl} of terminating events, and an initial value $Initially_{Fl}$ that can be true or false [9]. The sets of initiating and terminating events must be disjoint. A fluent definition takes the form:

fluent Fl = < I_{Fl}, T_{Fl} > initially $Initially_{Fl}$

For example, we can define a fluent *LOGGED* to capture whether a user is logged:

fluent *LOGGED* = <*login, logout*> initially *false*

This fluent states that a user is logged (resp. not logged) after a *login* (resp. *logout*) event occurs and until a *logout* (resp. *login*) event occurs, and is initially not logged.

Given a set of fluent Φ and a LTS trace $<e_0,\ldots,e_n>$, a *state* can be defined after every event in the trace. This state is characterized by the value of every fluent at this point in the trace. In such *fluent value assignment,* a fluent gets *true* (resp. *false*) if either of the following conditions holds:

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

Process goals and properties can be defined in Fluent Linear Temporal Logic (FLTL), a LTL variant where atomic propositions are fluents. The FLTL assertions use standard operators for temporal referencing such as: ○ (at the next smallest time unit), ◊ (some time in the future), □ (always in the future), U (always in the future until), W (always in the future unless), → (implies in the current state), ⇒ (always implies) [19, 15]. For example, the goal "*the user should be logged to download a paper*" is formalized as:

LoggedToDownload: $\Box(\bigcirc \text{DOWNLOAD} \rightarrow \text{LOGGED})$

where *DOWNLOAD* is a fluent becoming true after the event *download* and false after any other event in the alphabet.

3. Process models as guarded hMSC

A *guarded hMSC* is a directed graph where each node is a MSC, a decision node, or a finer-grained guarded hMSC.

For example, the *paperRequest* box of the hMSC in Fig. 2 is the MSC shown in Fig. 1. (To keep our running example simple, the MSC nodes in the sequel will consist of a single event having the MSC name.)

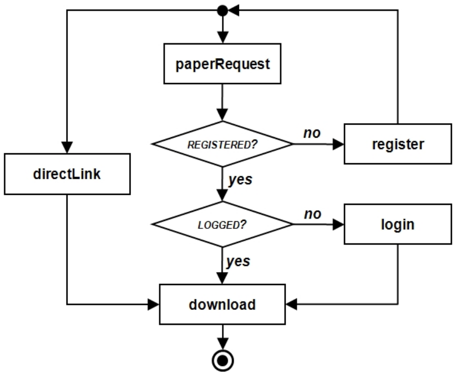

Figure 2: Guarded hMSC

Guarded hMSCs may capture parallel processes and decisions. Parallelism arises from MSC nodes. A *decision node* states specific conditions for the tasks along outgoing branches to be performed. A *guard* labels each outgoing branch; it must be evaluated to true for this branch to be followed. Guards are specified as Boolean expressions on fluents. 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 (as in Fig. 2).

Beyond the capturing of explicit decisions regulating subsequent tasks, guards prove convenient for modeling systems or processes where different instances can start in different states. In our example, the model should cover different processes dependent on the user's initial state. Different paths in the model will be followed dependent on whether a user is initially already registered, currently logged, etc.

In a fluent-based specification, we will just omit the initial value $Initially_{Fl}$ for the relevant fluents, meaning 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 *REGISTERED* hereafter specifies that some users are already registered when surfing the web site while others are not:

fluent *REGISTERED* = *<register, unregister>*

When initial values are left unspecified for some fluents, 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 [15]. For example, assuming that the initial values of fluents *REGISTERED* and *LOGGED* are not specified, we know that a user cannot be logged when not registered. Any initial state should meet this property. To support this, a guarded hMSC may be annotated with an *initial condition* constraining the acceptable initial values of unitialized fluents. In our example, the initial condition might be: $LOGGED \rightarrow REGISTERED$. Initial values in fluent definitions are no longer needed then.

The introduction of guards in hMSCs provides a source for interesting checks such as guard completeness and disjointness (see section 6.3).

4. 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 guards or events on transitions. It provides 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, interresting analyses may be performed at g-LTS level, see Section 6.

4.1 Guarded LTS

A *guarded LTS* (g-LTS) is defined by a structure (Q,Σ,Φ,δ,q_0,C_0) where Q is a finite set of states, Σ is a set of event labels, Φ is a set of fluents defined on Σ, $\delta \subseteq Q\text{x}(\Sigma\cup 2^{\Phi})\text{x}Q$ 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.

In a guarded LTS, transitions are labeled either by a guard or by an event. A *guard* is a conjunction of literals where a literal is a fluent or its negation. Intuitively, the guard must be evaluated to *true* for its transition to be activated. Note that every defined fluent or its negation must appear in every guard. 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 {[*REGISTERED*]} from state 1 to state 2 in Fig. 3 actually covers two transitions guarded with [*REGISTERED* ∧ *LOGGED*] and [*REGISTERED* ∧ ¬*LOGGED*], respectively.

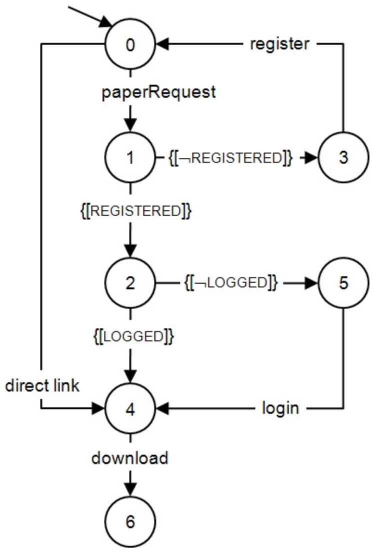

Figure 3. g- LTS for the guarded hMSC in Fig. 2

4.2 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,Σ,Φ,δ,q_0,C_0). A *trace* of G from q_0 is a pair (*Init*, $<l_0,\ldots>$) where *Init* is an initial fluent value assignment, mapping every fluent in Φ to *true* or *false*, and $<l_0,\ldots>$ is an infinite sequence of labels $l_i \in \Sigma\cup 2^{\Phi}$, some being events and others being guards. Such trace is accepted by G from q_0 iff the following *acceptance conditions* are met for every i:

trace inclusion: $\exists\, q_{i+1} \in Q$ 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 fluent value 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 fluent value 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 a trace 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.

4.3 Guarded hMSCs as g-LTS

A guarded hMSC can be rewritten as a g-LTS having the same traces. The rewriting algorithm extends [23] to take a guarded hMSC as input and a guarded LTS for the global system as output. The latter abstracts from the agents and captures the set of global behaviors covered by the hMSC. Our algorithm may be outlined as follows.

Handling nodes. Every hMSC node yields a behaviorally equivalent sub-LTS.

- A MSC node is rewritten as a sub-LTS collecting the linear event sequences from the scenario.
- A decision node is rewritten as a sub-LTS having only one state and no event.
- For a node expanded into a finer-grained hMSC, the procedure is applied recursively to obtain the corresponding sub-LTS.

In each case, initial and terminal states are added to the corresponding sub-LTS to connect transitions created in the next step.

Handling edges. The edges in a guarded hMSC yield transitions between the terminal and initial states of the sub-LTS corresponding to their source and target nodes, respectively.

- An outgoing edge of a decision node is labeled by a guard. As this guard may be any Boolean expression on fluents, we first compute all conjunctions of fluent literals satisfying it. Each of these yields a guarded transition in the g-LTS. A guarded hMSC edge is thus rewritten as a set of guarded transitions in the g-LTS.
- Any other edge is simply converted as LTS transition labeled with an unobservable event (*tau*), which can be removed while preserving

trace equivalence using standard automata algorithms.

This extended algorithm yields the g-LTS in Fig. 3 from the guarded hMSC in Fig. 2.

5. From g-LTS to LTS

The set of traces accepted by a g-LTS is determined by building a trace-equivalent LTS. The latter is a parallel composition of LTS ensuring the various acceptance conditions in Section 4.2. The first LTS in this composition is a "super LTS" meeting the *trace inclusion* condition. To meet the *admissible start* condition, an initializer LTS is added in the composition; this LTS forces initial value assignments on fluents to satisfy C_0. To meet the *guard satisfaction* condition, the set of traces of the super LTS is pruned further by composing it with fluent automata. Let us make each LTS in the composition further precise.

Super LTS. The LTS (Q, Σ, δ, q_0) meeting the *trace inclusion* condition is built from the g-LTS (Q_g, Σ_g, Φ, δ_g, q_{0g},C_0) as follows:

$Q = Q_g \cup \{q_{start}\}$

$\Sigma = \Sigma_g \cup 2^\Phi$

$\delta = \{(q_i, l, q_j) \mid (q_i, l, q_j) \in \delta_g\} \cup \{(q_{start}, start, q_{0g})\}$

$q_0 = q_{start}$,

where q_{start} is a new specific initial state for the super LTS and *start* is a transition from it on which the *Initializer* will synchronize for their composition to guarantee the *admissible start* condition (see below). The *start* event will enforce two phases: (a) fluent value assignments before it, to define an admissible initial state, and (b) system run after it.

In this super LTS, some transition labels l in the transition function δ denote original events whereas others are event encodings of the guards they are replacing. We will call them *guard-events*. The LTS alphabet is thus extended accordingly. To avoid confusing a guard and its guard-event, we will denote the latter by dropping the brackets and expressing the logical connectors in natural language (AND, OR, NOT). For example the guard-event for the guard *[¬ LOGGED ∧ REGISTERED]* is denoted by *NOT LOGGED AND REGISTERED*.

The super LTS defines a superset of traces; it meets the *trace inclusion* condition by construction. We now need to restrict this set so as to meet the other acceptance conditions.

Initializer LTS. This automaton enforces the *admissible start* condition by letting its events, encoding fluent value assignments, meet the initial condition C_0 before synchronizing with the super LTS on the *start* shared event. A self-transition labeled {*TRUE*} on its final state is added to avoid undue restrictions on the occurrence of guards from the super LTS. This self-transition represents the set of 2^Φ transitions corresponding to all possible combinations of fluent literal values.

Fig. 4 shows the *Initializer* LTS for our running example. The transitions between the two first states are labeled by guard-events. They capture fluent value assignments meeting the initial condition *LOGGED → REGISTERED*.

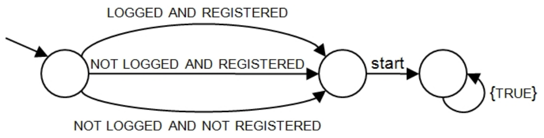

Figure 4: Initializer LTS

Fluent LTS. The *guard satisfaction* condition is enforced by pruning all traces violating guards in the super LTS. For this we compose the super LTS with fluent automata. The latter keep track of the current fluent values; guard-events are constrained to happen only when the corresponding guard is true.

For example, the fluent LTS for fluent *LOGGED* is shown in Fig. 5. The states q_u, q_f, q_t correspond to the states where the fluent is not assigned yet, is *false*, and is *true*, respectively. The fluent's initiating and terminating events synchronize with the super LTS to keep track of the current fluent value at each step. Transitions from the *unassigned* state are introduced to synchronize with the first transitions of the *Initializer* LTS (before event *start*). A transition labeled by {LOGGED} corresponds to two transitions, labeled by LOGGED AND REGISTERED and LOGGED AND NOT REGISTERED, and similarly for {NOT LOGGED}. When the fluent is *false* (resp. *true*), the fluent automaton prevents the occurrence of any transition with LOGGED (resp. NOT LOGGED). Those transitions thus prevent activation of guard-events when the corresponding guards are not satisfied.

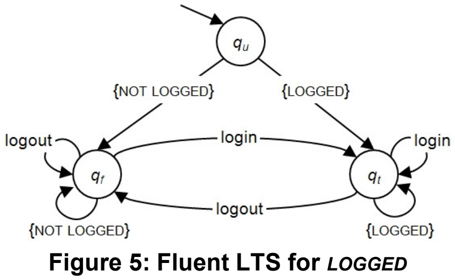

Figure 5: Fluent LTS for *LOGGED*

A fluent automaton is more precisely defined as follows:

$Q = \{q_u, q_t, q_f\}$

$\Sigma = I_{Fl} \cup T_{Fl} \cup 2^\Phi$

$$\begin{aligned}\delta = \ & \{(q_f, e, q_t) \mid e \in \mathrm{I}_{\mathrm{Fl}}\} \cup \{(q_t, e, q_t) \mid e \in \mathrm{I}_{\mathrm{Fl}}\} \\ & \cup \{(q_t, e, q_f) \mid e \in \mathrm{T}_{\mathrm{Fl}}\} \cup \{(q_f, e, q_f) \mid e \in \mathrm{T}_{\mathrm{Fl}}\} \\ & \cup \{(q_f, x, q_f) \mid x \in 2^{\Phi}, \mathrm{Fl} \notin x\} \\ & \cup \{(q_u, x, q_f) \mid x \in 2^{\Phi}, \mathrm{Fl} \notin x\} \\ & \cup \{(q_t, x, q_t) \mid x \in 2^{\Phi}, \mathrm{Fl} \in x\} \\ & \cup \{(q_u, x, q_t) \mid x \in 2^{\Phi}, \mathrm{Fl} \in x\} \\ q_0 = \ & q_u\end{aligned}$$

Synthesized LTS. Putting all pieces together, the trace-equivalent LTS of a g-LTS is obtained through the following parallel composition:

$$(\mathit{Super\ LTS} \ || \ \mathit{Initializer} \ || \ F_{Fl1} \ || \ldots || \ F_{Fln}) \setminus \{2^{\Phi}, \mathit{start}\},$$

where || is the standard LTS composition operator and `\{events}` is the hiding operator replacing all specified events by unobservable *tau* events. In this case, we hide all events that are not in the system alphabet, i.e., the guard events and the *start* event. The resulting LTS may be further minimized. It provides, by construction, the set of event traces accepted by the g-LTS.

In practice, we may want to keep the *start* event as well as the initial fluent value assignments before it, e.g., for annotating counterexample traces produced by the model checker or for documentation purpose. The hiding procedure can be adapted to hide guards only.

Fig. 6 shows the LTS generated from the g-LTS in Fig. 3. Note that the fluent values are assigned before starting, to determine the initial state.

6. Analyzing guarded hMSCs

This section presents different kinds of analysis our tool performs on processes modeled by guarded hMSCs, including model checking, state invariant generation, and guard analysis.

6.1 Model checking

We may want to verify that LTL properties on fluents are satisfied by our process model and, if not, see a trace counter-example showing the violation. For example, if the property:

$$\mathit{LoggedToDownload} : \ \Box \ (\bigcirc \ \mathit{DOWNLOAD} \rightarrow \mathit{LOGGED})$$

is violated, we would like to see a sequence of tasks allowed by the model where a user downloads a paper without being logged.

FLTL properties can be verified on LTS models using a model checking procedure described in [9] and implemented in LTSA [18]. This procedure cannot be used directly on our generated LTS. It checks a property for a specific initial state; in our case we want to check it for any initial state satisfying the initial condition. In case of violation, we would like a counterexample trace with a specific initial state leading to violation. The procedure in [9] is therefore extended accordingly.

In [9], the checked LTS is composed with (a) a Büchi automaton encoding the negation of the verified property *P*, (b) fluent automata, and (c) a synchronizer forcing the transition on the Büchi automaton after every system event. The property *P* is violated if an accepting state of the Büchi automaton can be reached in this composition. Our extension is outlined as follows.

- The checked LTS is the one generated from the process model using the algorithm from the previous section. In this LTS, the initial fluent value assignments and the *start* event are kept.
- The Büchi automaton is generated for the property `¬([](start->P)`, instead of `¬ P;` the user's property *P* must be verified only after the occurrence of a *start* event.
- The fluent automata are the ones defined in the previous section. Those in [9] do not contain the *unassigned* state, which is required here.
- The synchronizer LTS is slightly modified so as to first synchronize with initial fluent value assignments and the *start* event.

When verifying the property *LoggedToDownload* on the process model in Fig. 2, the extended model checker returns the following counterexample trace:

```
NOT LOGGED AND NOT REGISTERED
start
directLink
download
```

The counterexample showing the violation path in the process model always provides an initial fluent value assignement, followed by the event *start*, followed by an event trace leading to violation. In the

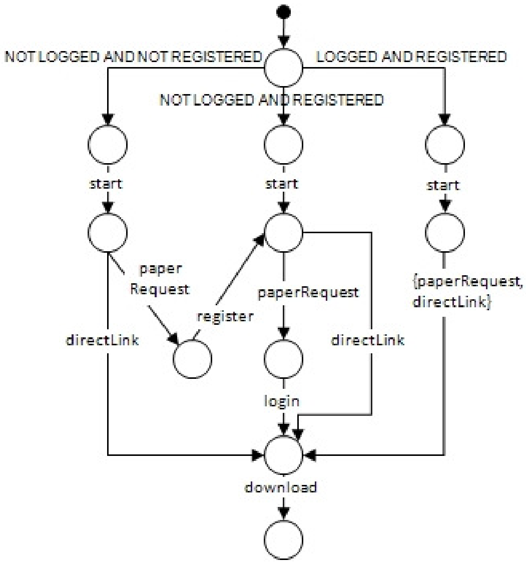

Figure 6. LTS generated from the guarded hMSC in Fig. 2

example, the counterexample shows that the use of the *directLink* feature may lead to a violation of the property if the user is initially not logged and not registered. In this case, the trace suggests that the definition of fluents LOGGED and REGISTERED should be adapted so that the *directLink* feature encapsulates user authentication, ensuring her to be registered and logged.

6.2 State invariant generation

A *state invariant* of a state machine is an assertion on some specific state which holds every time this state is visited. Annotating LTS state machines with state invariants has multiple benefits: the understandability and documentation of the state machine is improved; the invariants can be used for validation and error detection; and code generators may use them for improving code quality. Moreover, state invariants are required for the guard analyses in Section 6.3 hereafter.

An algorithm for generating state invariants on LTS from fluent definitions was described in [4]. Each state is decorated by a conjunction of fluent literals. A fluent or its negation appears in the conjunction decorating some state if it is *true* or *false* for all LTS executions reaching this state. It does not appear in the conjunction if it is *true* for some LTS executions reaching the state and *false* for others. For example, if $(A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge C)$ is the weakest invariant at some state, the algorithm in [4] will return *C,* as *A* and *B* are *true* for some executions and *false* for others.

The algorithm presented here extends our previous one along two directions. The computed state invariants are more accurate; they capture all possible literal combinations satisfied by their associated state through disjunctions of conjunctions. Moreover, the new algorithm handles guards and initial conditions (while applicable to guard-free LTS as well.)

The algorithm 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 initial condition C_0. The algorithm propagates fluent literals through the state machine according to fluent definitions and outcomes of guards. 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.

Fig. 7 shows the invariant generation algorithm. The set *ToExpl* collects the states whose decoration changed and to which propagation of literals should still be applied. The algorithm terminates when this set is empty. It will eventually be empty as a state can change its decoration at most 2^Φ times. In the worst case, a state will be decorated by the disjunction of all possible literal combinations, that is, *true*.

```
Input:  A guarded LTS $(Q, \Sigma, \Phi, \delta, q_0, C_0)$,
        where $\Phi$ is a set of fluents $Fl_i$
Output: decor : $Q \rightarrow P(2^\Phi)$

/* initial decorations */
for each $q \in Q$ do
   decor(q) $\leftarrow$ false ;
decor($q_0$) $\leftarrow$ $C_0$ ;

/* fixpoint loop, starting with the initial state */
ToExpl $\leftarrow \{q_0\}$
while (ToExpl $\neq \emptyset$) do
   source $\leftarrow$ getOne(ToExpl);
   ToExpl $\leftarrow$ ToExpl\{source};
   for each (target,label) such that
        (source,label,target) $\in \delta$ do
      /* propagate source decoration (by case) */
      if (label $\in 2^\Phi$) then
         pDecor = decor(source) $\wedge$ label
      else
         pDecor = decor(source)$|_{event}$

      /* compute disjunction with old decoration */
      decor'(target) $\leftarrow$ pDecor $\vee$ decor(target);

      /* mark as "to explore" if changed */
      if (decor'(target) $\neq$ decor(target)) then
         ToExpl $\leftarrow$ ToExpl $\cup$ {target};
         decor(target) $\leftarrow$ decor'(target)
return decor
```

Figure 7. Fixpoint generation of invariants

The expression *decor(source)*$|_{event}$ in Fig. 7 denotes the decoration of state *source* after the corresponding *event* has been applied. If *event* belongs to the initiating (resp. terminating) events of a fluent *F*, this expression is calculated by replacing all occurrences of $\neg F$ (resp. *F*) by *F* (resp. $\neg F$). As an event may belong to initiating/terminating events of several fluents, this must be done for each fluent.

Let us see how the guarded LTS in Fig. 3 gets decorated. Table 1 shows decorations at each step until a fixpoint is reached.

Step 0. We initialize all decorations to *false* except the initial state, which gets the initial condition ¬LOGGED ∨ REGISTERED.

Step 1. Only state 0 is in *ToExpl*. We propagate its decoration to states 1 and 4. Events *paperRequest* and *directLink* do not belong to initiating or terminating events of fluents; the propagated decoration *pDecor* is simply the decoration of state 1, i.e. ¬LOGGED ∨ REGISTERED. State 1 and state 4 are decorated by the disjunction of *pDecor* with their old decoration: (¬LOGGED ∨ REGISTERED) ∨ *false*, yielding ¬LOGGED ∨ REGISTERED. States 1 and 4 are added to *ToExpl* as their decoration has changed.

Step 2. Let us assume state 1 is chosen in *ToExpl.* Its decoration is propagated to its successors (states 2 and 3). For each of these, we compute `pDecor` by taking the conjunction of the decoration of state 1 with their respective guard. The decoration of the states is obtained as the disjunction of the propagated decoration with their old one; both are added to *ToExpl*.

Step 3. State 3 is choosen in *ToExpl*. It has an outgoing transition to state 0; `pDecor` is computed as $(\neg\text{LOGGED} \wedge \neg\text{REGISTERED})|_{register}$. After event *register*, the fluent *registered* becomes *true;* `pDecor` is therefore equal to $\neg\text{LOGGED} \wedge \text{REGISTERED}$. The disjunction of `pDecor` with the old decoration of state 0 keeps it unchanged; state 0 is thus not added to *ToExpl*.

The process is continued until *ToExpl* is empty.

6.3 Guard analysis

Three kinds of guard-based 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.

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 in the same situation. In clinical processes, for example, every patient with the same state must be treated identically (see Section 7).

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

In our framework, those three types of checks are applied to guarded hMSCs. Incomplete, overlapping, or unsatisfiable guards are reported to stakeholders for fixing the problematic decision nodes.

Such checks are close in spirit to those supported in [11] 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 the decoration algorithm in Fig. 7.

The checks could be implemented at two different levels: on the g-LTS or on the trace-equivalent LTS. Our tool implements them on the g-LTS for several reasons. The alternative of taking the trace-equivalent LTS for checking incompleteness by deadlock analysis and overlaps by non-determinism detection would prevent us from determining whether the problem is specifically due to guards or not. Moreover, unreachable paths would not be present in the target LTS. It would be quite difficult to point out the problematic guards.

Once contextual conditions are generated as state invariants, the checks are automated through satisfiability checking. For a set of guards g_i at a decision node with contextual condition C holding right before, we need to check the following:

$$C \models \bigvee_i g_i \, ,$$

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

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

In case of incompleteness, our tool returns all fluent value assignments that are not covered by the guards. In case of overlap, it returns all fluent value assignments that meet several guards. In case of unsatisfiability, it indicates outgoing transitions that are unreachable.

7. Analyzing cancer therapy processes

Our techniques were applied on a real medical process for treatment of rectal cancer, with medical staff at the UCL university hospital as stakeholders. Our work is motivated by a joint project aimed at building and analyzing models of clinical pathways [2]. A *clinical pathway* is a well-defined process, based on medical protocols, guidelines and recommendations, centered on a specific patient class with similar needs, involving a multi-disciplinary team, and addressing clear clinical

Step	ToExpl	State0	State1	State2	State3	State4	State5	State6
0		$\neg L \vee R$	false	false	false	false	false	false
1	{**0**}	$\neg L \vee R$	$\neg L \vee R$	false	false	$\neg L \vee R$	false	false
2	{**1**,4}	$\neg L \vee R$	$\neg L \vee R$	R	$\neg L \wedge \neg R$	$\neg L \vee R$	false	false
3	{**3**,2,4}	$\neg L \vee R$	$\neg L \vee R$	R	$\neg L \wedge \neg R$	$\neg L \vee R$	false	false
4	{**2**,4}	$\neg L \vee R$	$\neg L \vee R$	R	$\neg L \wedge \neg R$	$\neg L \vee R$	$\neg L \wedge R$	false
5	{**4**,5}	$\neg L \vee R$	$\neg L \vee R$	R	$\neg L \wedge \neg R$	$\neg L \vee R$	$\neg L \wedge R$	$\neg L \vee R$

Table 1. Decoration steps for the g-LTS in Fig. 3. *L*, *R* stand for fluents *LOGGED*, *REGISTERED*, respectively.

goals. One project objective is to help medical staff detect and correct flaws in clinical pathways early in their elaboration using software engineering technology.

Preliminary model elaboration. Initial model fragments for the rectal cancer pathway were sketched on paper by clinicians using box-and-arrow diagram pieces showing tasks and decision nodes. These fragments were translated next into a guarded hMSC using our tool. Such translation already stimulated fruitful discussions with medical staff. Issues about the process were raised and discussed, leading to early clarification of the preliminary paper version.

Fig. 8 shows the emerging guarded hMSC for a pathway fragment, simplified for explanatory purpose. It consists of a sequence of *pre-treatment*, *treatment*, and *follow-up* tasks, cycling in case of cancer recurrence. The refinement of the *pre-treatment* task is shown on the right part of Fig. 8. It can be rephrased as follows.

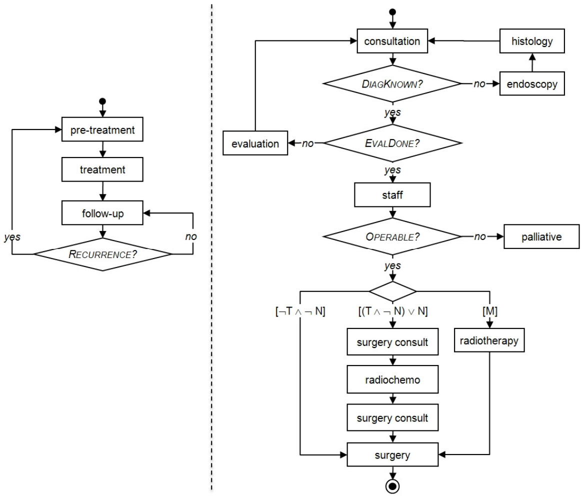

Figure 8. Treatment process for rectal cancer, with refinement of *pre-treatment* task

> "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). Based on this, the medical staff envisions some appropriate therapy strategy. When the patient is operable, the treatment may consist in simple surgery, or chemotherapy preceding surgery, or a combination of radiotherapy and chemotherapy preceding surgery. When the patient is not operable, palliative care is provided."

As model elaboration proceeds, tasks are further refined into MSCs or finer-grained hMSCs. (Unrefined tasks are encoded here as single events named by the task name.)

Fluent identification. Guard formalization requires fluents to be identified and defined. Some fluents were easily identified by medical staff. For example, the fluent *DIAGKNOWN* in Fig. 8 states that a diagnosis is known about the actual patient's cancer status when *histology* has been performed, and remains known until *follow-up* has been done. The fluent *EVALDONE* gets true when the *evaluation* task has been performed; the evaluation results remain valid until *chemotherapy* is performed. Hence the definitions:

fluent *DIAGKNOWN* = < *histology, follow-up* >
fluent *EVALDONE* = < *evaluation, chemotherapy* >

Other fluent definitions raise the need for refining tasks further. For example, the fluent *OPERABLE* refers to specific events identified by refining the inter-disciplinary decision meeting task named *staff* in Fig. 8. Similarly, the fluents *T*, *N* and *M* are obtained from the refinement of tasks *consultation*, *endoscopy* and *histology*.

Ten fluents were defined from the informal guards in Fig. 8. Most of them do not specify an initial value as the latter may differ from one patient to the other. The initial condition C_0 reflects a domain property elicited from clinicians, namely, "if the spread evaluation is already done in the initial state (possibly by another hospital), then the diagnosis is known (as possibly provided by that other hospital)". This condition is easily formalized:

$$\mathit{EVALDONE} \rightarrow \mathit{DIAGKNOWN}.$$

Guard analysis. Our tool was used to perform the three types of guard analysis discussed in Section 6.3. Undesired non-determinism in decisions was automatically detected in the clinical pathway fragment shown in Fig. 8; the guards

$$[\neg\, \mathrm{T} \land \neg\, \mathrm{N}],\ [(\mathrm{T} \land \neg\, \mathrm{N}) \lor \mathrm{N}],\ \text{and}\ [\mathrm{M}]$$

do overlap. The problem was easily fixed in this case by adding $\neg\, M$ in the first two guards, as advised by medical staff. Similar problems involving guards were identified in other pathway fragments supplied by medical staff, and similarly fixed, including missing guarded branches at other decision nodes.

Model checking. While building a companion goal model [15], a number of descriptive and prescriptive properties were identified with medical staff and model-checked using our tool.

For example, one *Avoid* goal states that "a patient may never get irradiated twice in the same anatomic zone". Its parent goal is to avoid the potentially serious side-effects of iterative radiation. In simpler form, the goal *Avoid* [PatientIrradiatedTwice] states that the *radiotherapy* and *radiochemo* tasks may not be performed when the patient has already been irradiated (we omit the "same anatomic zone" restriction to simplify the presentation.) The formal specification of this goal is:

$$\Box\, ((\bigcirc\, \text{radiotherapy} \lor \bigcirc\, \text{radiochemo}) \rightarrow \neg\, \text{IRRADIATED})$$

This specification yields an additional fluent definition:

fluent *IRRADIATED* = < *{radiotherapy, radiochemo}, { }* >
initially false

When verifying this property on the process model fragment in Fig. 8, our model-checker returns the following violating trace:

```
NOT IRRADIATED AND M AND …
start
consultation
[…]
radiotherapy
[…]
follow-up
consultation
[…]
radiotherapy
```

This trace shows that cancer recurrence, illustrated in the trace by the *follow-up* task followed by a new *consultation* task, was overlooked in the original pathway fragments supplied by medical staff. The problem may be fixed by guarding the *radiotherapy* task further, e.g., with $[\text{M} \wedge \neg \text{IRRADIATED}]$.

This fix, however, breaks guard completeness on the decision node, which raises the following new question: "What happens in case of cancer recurrence, when irradiation care would otherwise have been provided?".

A number of other goals and domain properties were verified, among which few were satisfied by the preliminary model.

- Some of the checked properties were originally too strong, being representative of a subset of patients only (e.g., those without metastasis). Such problem must be fixed by correct reformulation of the property in the goal model, possibly together with missing domain properties.
- Other properties were violated because of exceptions not correctly covered by the model (e.g., a patient coming from emergencies). In this case, further iterations with medical staff are required, possibly resulting in refactoring and refinement of the process model.

Preliminary lessons learned. Such iteration of stepwise process model elaboration, analysis, and refactoring appears quite promising.

- We can systematically detect errors arising from novice modelers, such as superfluous or overlapping guards, or resulting from the complexity of guards in the medical domain.
- Model analysis may reveal ambiguities in the model and suggest ways of fixing them –e.g., the addition of "$...\wedge\neg M$" in the first two guards as discussed before.
- Error detection can be made early, avoiding the difficulties of late fixing of anomalies disseminated throughout a large, complex model.
- Early analysis contributes to the model elicitation process. Implicit information gets clarified. In case of overlapping guards, we may highlight some modeling error or some desired non-determinism to be explicitly documented.

As in [3], we mostly found errors in the process *models* rather than in the *actual* processes currently followed by medical staff. Analysis is nevertheless important for a correct model to be used, e.g., to highlight potentially hazardous situations in real processes, enact daily work processes, advise less experienced people, deliver explanations or tutorial material, etc.

8. Conclusion

Model-driven development requires adequate model development. Model building is a complex and error-prone task. It should be supported, especially in case of critical systems or processes, through techniques and tools for stepwise model elaboration, analysis, and refinement.

Guarded hMSCs were introduced in this paper to model multi-agent processes involving decisions. This formalism proves understandable by stakeholders and convenient for modelling decomposable tasks, interaction scenarios, goals, and decision trees. The language has a clear formal semantics defined in terms of LTS.

Guarded LTS were proposed as an intermediate event-based formalism allowing state-based guards to be defined as conjunctions of fluents. Their trace semantics is defined in terms of LTS.

Three complementary techniques were proposed for analyzing process models as guarded hMSCs. The three of them are implemented by a tool. The model checker verifies safety properties that may refer to states and events. The tool also generates state invariants which are, in particular, used for analyzing decision alternatives against completeness, disjointness, and satisfiability.

Our approach was applied to a real safety-critical process for cancer treatment. Errors in the model fragments supplied to us were successively detected and corrected. Although preliminary, this experience appears promising for other clinical pathway models we are currently working on.

The techniques described in the paper were implemented and integrated in our ISIS tool for scenario-based and goal-based synthesis of behavior models [4, 5]. Guards are encoded as binary decision diagrams (BDDs), which allows for a compact representation of Boolean formulas. This avoids exploding guards through multiple transitions (up to 2^{Φ} in the worst case). Standard operators from BDD libraries also make it easier to implement the algorithms for fluent decoration and guard analysis. Our model checker is fully integrated and relies on an algorithm specifically developed to handle BDDs during LTS composition. Note that a worst-case computation of the trace LTS may lead to identifying up to 2^{Φ} states immediately reachable from the initial one. The provision of initial conditions is intended to prune these significantly. The exponential blow of the trace LTS naturally results from the ability of models with guards to cover numerous behaviors in an implicit, compact way.

Our approach raises a number of challenging issues. Firstly, some paths may be missing from the preliminary process model fragments provided by stakeholders. The technique in [4] for generalizing accepted behaviors of a MSC specification might be extended to handle guarded hMSCs as input. Secondly, the introduction of guards in scenarios and LTS raises the issue of guard monitorability; the agents involved in the process must be able to evaluate "their" guards in order to meet the process model. Thirdly, obstacle analysis should be applied to process models [16]. In particular, guards often refer to external quantities; they might be wrongly evaluated if such quantities are not accurately reflected in the model. Finally, the integration of multiple process models raises conflict management issues [15]. For example, a patient following both rectal cancer and diabetis pathways is exposed to feature interaction problems. All such problems need to be detected and resolved during process modeling through dedicated validation techniques.

Acknowledgments.

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

References

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

[2] H. Campbell, R. Hotchkiss, N. Bradshaw, and M. Porteous, "Integrated care pathways", *British Medical Journal (BMJ)*, 1998, pp 133-137.

[3] 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.

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

[5] 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.

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

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

[8] J. Fox, N. Johns, A. Rahmanzadeh, "Disseminating Medical Knowledge – The ProForma Approach", *Artif. Intell. Med.* Vol. 14, 1998, 157-181.

[9] D. Giannakopoulou and J. Magee, "Fluent model checking for event-based systems", *Proc. ESEC/FSE 2003*, Helsinki, 2003.

[10] M. Han, T. Thiery, X. Song, "Managing Exceptions in Medical Workflow Systems", *Proc. ICSE'06, 28th Intl. Conf. on Software Engineering*, Shanghai, May. 2006.

[11] 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.

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

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

[14] 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.

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

[16] A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering", *IEEE Trans. Softw. Eng.*, Vol. 26 No. 10, 2000, 978-1005.

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

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

[19] Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

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

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

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

[23] S. Uchitel, J. Kramer, and J. Magee, "Incremental elaboration of scenario-based specifications and behavior models using implied scenarios, *ACM Trans. Softw. Eng. and Methodol.*, Vol. 13 No. 1, 2004, 37-85.