Representing and Exploring the Behavioral View of Goal Models: A Semi-Formal Approach

Project for the Course: "Automated Verification"

Sotirios Liaskos

Toronto
2003

Contents

1 Introduction

1.1 Goal Analysis and System Behavior

Goal analysis has been found to be an effective way for *early* requirements identification and modeling ([5, 26, 33]). It provides an intuitive way to capture information about *what* the stakeholders want to achieve, *why* and *how* they can do so. Especially to answer the last question, *goal decomposition* is employed as a way to divide the original problem into more specific sub-problems with a better defined solution: goal decomposition ends up to possible sets of tasks, the performance of which is assumed to be sufficient for the satisfaction of the root goals. The *goal decomposition model* we will discuss here, is a graph that illustrates this in a readable, *semi-formal* manner.

Semi-formal would mean that although the units and links of the representation are formally declared, their properties are informally asserted ([31]). For example Entity-Relationship diagrams are considered to be semi-formal because the entities, for instance, are not yet completely/specifically defined. In the same sense, in a semi-formal decomposition model, the goals and the tasks are described in an informal way (a title possibly accompanied with a comment in natural language). However the decomposition structure is declared formally.

The focus of goal analysis in this early/ semi-formal level has not been the description of the *temporal* characteristics of the system of discourse. Although a goal decomposition model says a lot about what needs to be done for the goals to be satisfied it does not answer *when* or in *what order*. This has been subject for several research efforts over the past years. We can divide the approaches into three major categories:

Goals and Scenarios. Approaches of this category use *scenarios* to represent temporal/behavioral characteristics of the system being modeled. A scenario constitutes a chunk of knowledge acquired and modeled separately from goals although these two end up being strongly related. Thus, in [30] scenarios are used as "test cases" for validating/steering the goal analysis process, whereas in [2] scenarios are introduced as operationalizations of goals. In [29] scenarios are first established as concretizations of high level goals, and then used for the definition of new goals, in the spirit that the founding element of the requirements model is the touple $\langle$Goal, Scenario$\rangle$ (an approach further elaborated in [22]). In [32] goal and scenario elaboration are preformed in parallel and a systematic way to infer goals from scenarios is given. In [24] *use-case maps* ([4]) are used to model scenarios while, in parallel, goal models are developed in $i*$ ([33]). What is interesting in this work though is that in some cases, the modeling constructors of the use-case maps are tasks that are also found in the goal model.

Common denominator of the above approaches is that an enumerable (in a practical sense) set of scenarios can be captured/defined and they constitute a separate body of knowledge, complementary to goals.

Formal Specification and Analysis. Independent on whether a scenario acquisition process is presumed, goal models obtain a temporal dimension when formalized. Both KAOS [10, 32, 23], the formal side of goal analysis, and Formal Tropos [14] employ linear temporal logic (LTL) for this purpose: each concept specified is associated with temporal formulae that serve as definitions, pre- or post-conditions, triggers etc. Then several sorts of formal analysis can be applied. In Formal Tropos, for instance the LTL formulae are converted into a Kirpke structure (aka an SMV model) so that the result can be verified against assertions. Of course, the assertions being verified are also given in LTL.

In KAOS the issue of defining the correct LTL specification from informal requirements is dealt through the use of the temporal logic specification pattern system ([11]). However, this approach does not allow the visualization of the big picture of the behavioral model: from the textual description the experts derive complicated temporal formulas without any intermediate results. We will return to this shortly.

Temporal Notation. The third approach is to directly embed temporal information in the semi-formal goal model. The focus here is to understand and communicate the behavioral aspect of the goal model while being developed. However simulation and other sorts of analysis and reasoning are still possible. We could not find examples that clearly follow such an approach other than [15]. In this work the low level tasks of a goal decomposition model are gathered in a separate diagram and connected through preconditions and means-ends links. The resulting diagram can be directly interpreted into ConGolog ([16]) and used for simulating the system being modeled. Thus, although there is no explicit specification of scenarios, the different ways (orders) with which tasks are preformed illustrate the intended temporal dimension of the decomposition model.

Let us also make two comments that serve as our motivation:

- Aiming at steering the development of highly *customizable software*, there is an increasing interest for requirements models that express *variability*. In [20], where this is examined in the context of product families, several types of variability are described. Thus, *functional variability* refers to the several alternative combinations of tasks with which the same set of requirements is fulfilled. *Behavioral variability*, on the other hand, focuses on alternative *courses* with which the same tasks can be performed.

  In [21], it is argued that AND/OR decomposition graphs are capable of representing such functional variability in a compact, implicit manner, allowing, at the same time, the application of simple reasoning procedures about it.

  But what is the modeling framework for representing behavioral variability? Such a framework would use an intuitive, compact and readable meta-model to imply a great number of behavioral alternatives, just as an AND/OR tree may imply billions of functional alternatives with a few tens of nodes.

- There is a substantial difference between formalizing and verifying a goal model against bugs (Formal Tropos, KAOS) on one hand and describing and exploring its temporal behavior at an early phase on the other.

  The former is an arguably expensive process that involves a mature requirements model and several experts to perform the formalization and the verification correctly (see [12], for example, on outsourcing the formalization tasks). The motivation is to eliminate errors and defects in the specifications.

  The latter, which interests us here, aims at understanding the behavioral implications of the attempted decomposition and communicating them with the users. However, when the goal is such, sophisticated formulations in temporal logic cannot achieve what simple and intuitive diagrammatic models can. For example, think of activity diagrams in

UML [13] versus CTL (or CTL pattern descriptions on the opposite side) for initiating the behavioral modeling of the system of discourse.

1.2 Model Checking and Exploration Technologies

The term model checking ([9, 28]) has been widely used to refer to methods and algorithms for checking whether for a *finite-state* model ("program") M, and a *temporal formula* φ, $M \models \varphi$ holds. The motivation has been the inspection of software artifacts (focusing on concurrent ones) subject to bugs and undesired properties.

There have also been proposals to apply the same technology to "inspect" requirements models (e.g. [12, 14] and KAOS literature). Common denominator in these efforts is the fact that a formal specification of the requirements models is presumed, either in one of the goal-oriented languages mentioned above or in other popular formalisms (e.g. SCR [27]).

However, as we claimed in the previous section, formalization of requirements cannot be claimed to be an inexpensive process. On the other hand, in this early requirements phase, the corresponding semi-formal models (e.g. goal graphs) are constantly changing to reflect, for example, new information that is being gathered by the analysts or new goals and preferences that are being posed by the stakeholders. This multiplies the cost of the formalization effort, if such effort is at all possible due to the inherent vagueness of the early requirements models.

What can be possible however, is simple analysis and reasoning over the "partial" formality of these models. For example, it has been shown in [25] or in [17], that semi-formal goal graphs can already be subjects for algorithmic examination due to the diagrammatic language in which they are developed. Still, this work has been restricted to static analysis of goal satisfaction (more specifically: "satisficing") propagation and not how this dynamically evolves as the underlying operations are performed and events occur (with [15] being an exception, motivated, thought, towards simulation and not reasoning).

Therefore, we can investigate the possibility to apply model checking technologies before having a formal requirements model and while developing a semi-formal one on which some sort of qualitative reasoning is already possible. In other words, we can first examine how the state-transition model M can be defined at even earlier stages of the requirements process, than these currently proposed in the literature. Then, we can see whether this indeed serves any purpose, probably radically different than the one model checking technologies are currently used for.

A related technology that may prove very useful for such purposes is *Query Checking* ([6, 3, 19]). Given a state-transition model M and a temporal formula with free "variables" φ, query checking aims at finding formulae with which if we replace the variables, then $M \models \varphi$. In other words we do not ask whether a temporal formula is entailed by the model or not, but which instantiations of a template temporal formula are true for a model.

Thus, for our purposes, apart form NuSMV ([7]), a popular symbolic model checker, we will also consider the query checking tool *TQSolver* ([19]), developed in the University of Toronto.

1.3 In This Report

We will work as follows: we will first extend the language for constructing Tropos goal models, i^*, so that it can capture the dynamic view of goal and task satisfaction (Section 2). We will use a constraint-based approach, in order to have a large number of implicit behavioral examples, which will be the subject of our exploration.

Then we will translate this into a state-transition model (Section 3). Using model checking (NuSMV) and query checking (TQSolver) we will use two example goal trees to explore what kinds of temporal formulas/queries can be useful for understanding our model and the behaviors it allows (Section 4). We will summarize our findings in Section 5.

2 Adding Temporal Information to Tropos Goal Graphs

2.1 Goal Modeling in Tropos: Example

The Tropos methodology for requirements-driven development of information systems is presented in ([5]). Tropos distinguishes between *early* and *late* requirements. In the early requirements phase the *actors* that participate in the social system that will potentially relate to the system-to-be are first identified.

Then, using the i^* modeling language we represent the relationships that are formed between the actors. These relationships are *dependencies* between actors, subject to *goals*, *resources*, *tasks*. Here we will focus on goal dependencies: an actor (*depender*) depends on another actor (*dependee*) in order to achieve a goal (*dependum*). The process results to the *strategic dependency* (SD) diagram where the actors are represented together with their between dependencies.

In Figure (1) a very simple SD diagram is depicted. It is between a customer of a travel agency (*Traveler*) and the travel agency itself (*Agent*). The former depends on the latter in order to `[Buy an Airplane Ticket]`. The latter depends on the former to `[Sell an Airplane Ticket]`. From this bilateral goal dependency a relationship between the two actors emerges.

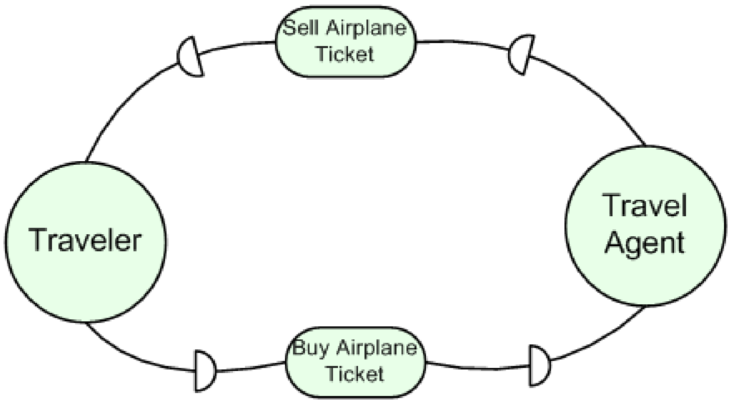

Figure 1: A Simple Pair of Goal Dependencies

From here we need to perform *AND/OR reduction* of the goal dependencies, in order to understand the courses of *tasks* (in this text also: *operations*) that are going to be required for the dependency goals to be fulfilled. AND/OR reduction (also: goal decomposition) is a gradual, recursive refinement of goals into subgoals that are less generic. A single decomposition step might be: 1) an AND-decomposition of a goal into subgoals which all need to be satisfied for the original goal to also be satisfied or 2) an OR-decomposition into subgoals from which the satisfaction of exactly[1] one is sufficient for the satisfaction of the original goal.

When the result of a decomposition step is a set of specific tasks rather than goals, the decomposition process will stop. The result of the process

[1]Thus, in practice and at least in this discussion it is a XOR decomposition.

is an AND/OR tree showing with what tasks (leafs of the tree) the original dependency goal (root of the tree) can be satisfied. In Figure (2) the decomposition of the dependencies between the Traveler and the Agent are depicted.

Notice that due to the existence of OR-decompositions there are several ways for achieving the root goals. Namely, as many as the solutions of the AND/OR tree. Let T_G be the set of all leaf level tasks of the decomposition tree for the goal G. $A_G \subseteq T_G$ is a minimal set of tasks that satisfy the goal G. Minimal in a sense that if we remove any of the tasks from the set, then the set will no longer satisfy the goal. We will refer to A_G with the term *alternative*. Then $\mathcal{A}_G$ is the set of all alternatives implied by G's decomposition tree.

So far, the system-to-be is absent. It is because we are still in the early requirements phase. From here we can assume a third actor (*The System*) can be introduced in order to take over the responsibility of performing any of the leaf level tasks. This leads to late requirements analysis, which we will not discuss here.

2.2 Precedence

Let us stay at the airplane ticket example we saw in the previous section (Figure 2). By taking a close look at the leaf level tasks we can identify some dependencies of the form: "t_1 cannot(/should not) be satisfied before t_2 is performed first". For example, the Traveler will not `[Learn More About Her Selection]` unless the Agent `[Says More About A Selection]`. Or, the Agent cannot `[Receive a Cheque (by mail)]` unless the Traveler `[Mails A Cheque]` first. Thus we can define a simple precedence relation between tasks.

Before we do so, it is important to underline the difference between *task performance* and *task satisfaction*. In particular we will say that *a task is satisfied* when the expected result of this task is realized. This might mean that the task was performed at some point in the past. Or that the same result has already been produced by a different task. Or that the task's purpose does not exist anymore. That is, when a task has *just* been performed it has *just* been satisfied, but not vice versa. Moreover, the fact that a task has been performed cannot be canceled as opposed to its satisfaction. For instance, if the task `[Wash one's Car]` was performed in the morning, that annoying afternoon rain can only cancel the satisfaction of the task and not

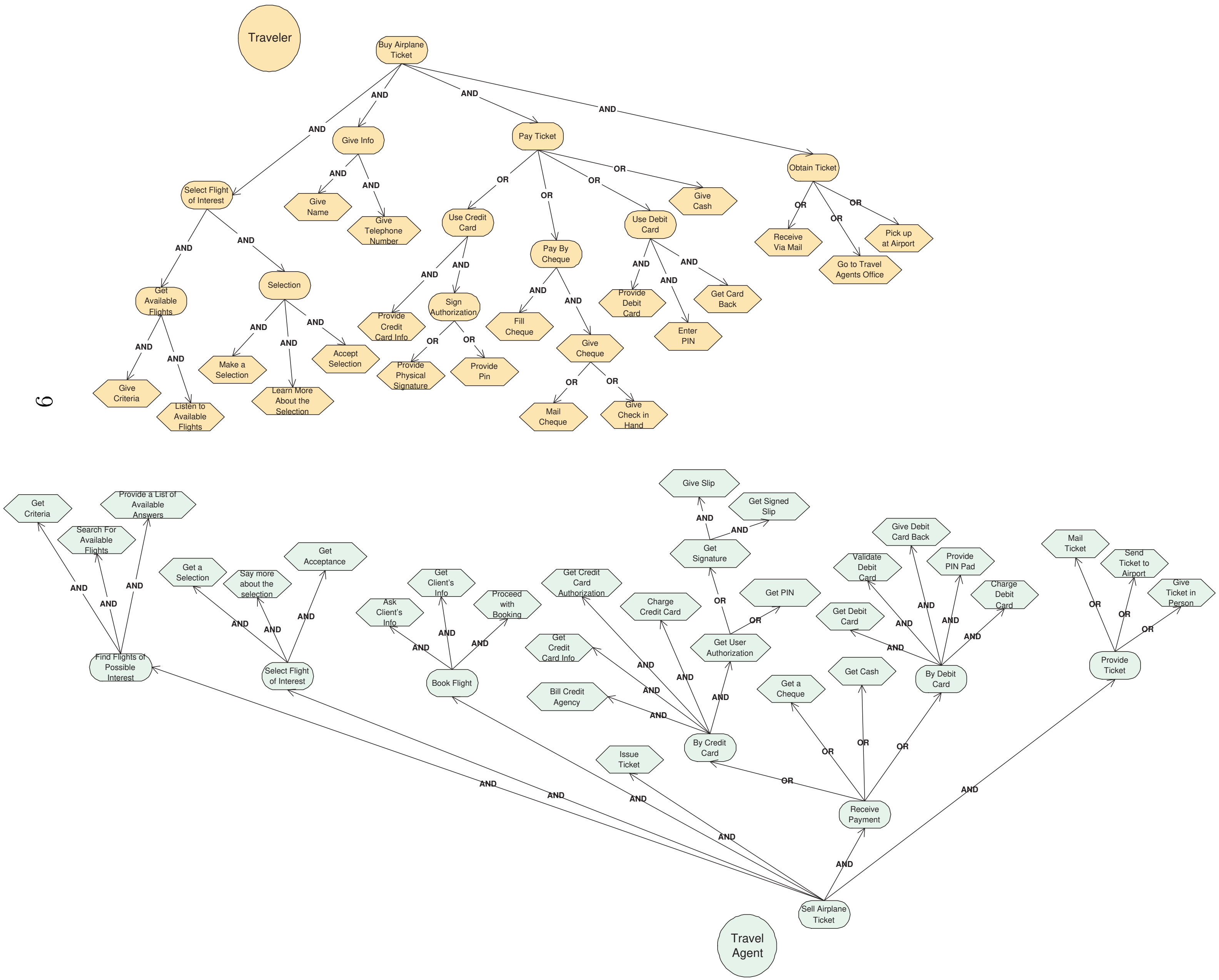

Figure 2: The Goal Decomposition Model

the fact that the task was performed. Satisfaction can be re-established by e.g. performing the task again.

This duality is a direct result of the goal reduction process: leaf-level tasks can be viewed both as (operationalized) goals that need to be satisfied and as operations that appear (together with their results) as historical facts in the timeline. For our purposes we will indeed need both interpretations.

Definition 2.1 We say that a task t_1 *precedes* a task t_2 when t_2 cannot be performed unless t_1 is satisfied. We write $t_1 \succ t_2$.

We can represent diagrammatically this relation by an arrow from t_1 to t_2 (Figure 3). We will call this the *precedence* link. For $\succ$ the following property holds:

Transitivity: if $t_1 \succ t_2$ and $t_2 \succ t_3$ then $t_1 \succ t_3$.

Figure 3: A Precedence Link

So far we define precedence only at leaf level. We can also define precedence between goals. For reasons that will be clearer later we define the strongest notion of goal precedence, the universal goal precedence.

Definition 2.2 Let T_{G_1}, T_{G_2}, be the sets of tasks for the goals G_1 and G_2. We say that G_1 *universally precedes* G_2 ($G_1 \succ G_2$) if and only if $\forall t_1 \in T_{G_1}, \forall t_2 \in T_{G_2}$: $t_1 \succ t_2$.

Or simply: before any of the tasks of G_2 is performed, the goal G_1 has to be satisfied first. Similarly we can define precedence between a goal and a task. Transitivity persists in all cases.

We can now go ahead and add all precedence links between goals and tasks in our airplane ticket example. The result is depicted in Figure (4). The process of identifying these links proved sufficiently intuitive. The analyst reads the goal structure and based on its semantics she adds the respective links. While performing this task in this example we observed the following:

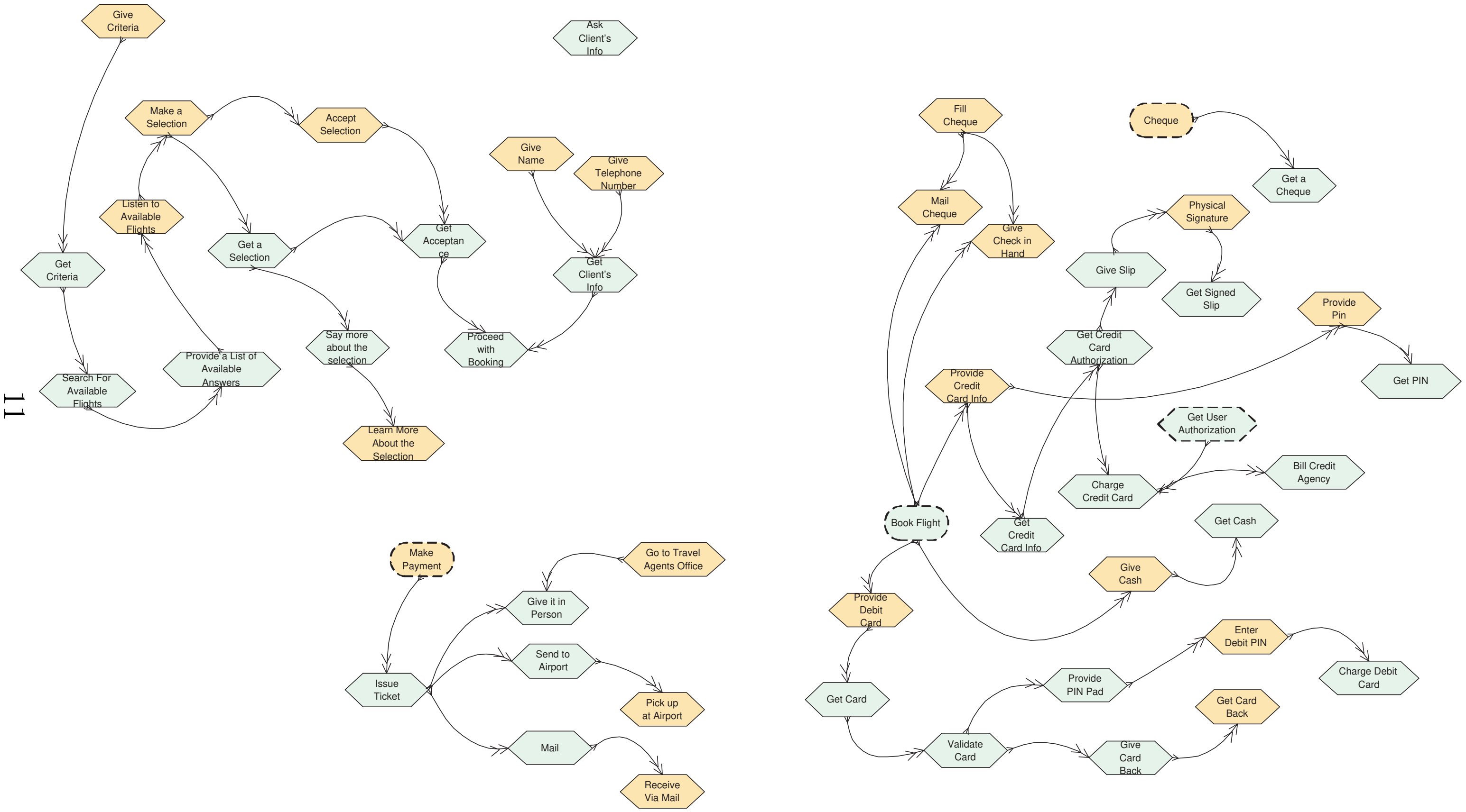

Figure 4: The Precedence Model

- The analyst will very much lean on the fact that transitivity of the precedence relation implies that the establishment of such links is an incremental process. In other words, rather than naively questioning about precedence between all possible pairs of tasks, she reads the links established so far and ignores pairs that already satisfy the precedence relation through transitivity.

- It is intuitively expected (and it happened in our case) that precedence was frequently defined between high level goals. The satisfaction of a high level sub-goal in its temporal dimension can be viewed as a *phase* towards the satisfaction of the root goal. Although not proved in this report, a goal decomposition is very often conceived as a decomposition into phases the sequencing of which is yet to be described. While it is the case that, in the end, operations of one phase may intervene with such of another preventing us from establishing a universal precedence link, clear-cut phase precedence links might prove useful for communicating an initial modeling result. Further elaboration may lift the high level precedence to establish a set of lower level ones over subgoals(/sub-phases) or even tasks.

We must also underline the fact that, at this level we are interested in a sound result and not necessarily a complete one. While soundness of the whole *precedence diagram* is argued by means of the soundness of the elementary precedence links, completeness (no precedence is missing) is something that will try to achieve through a *temporal qualitative analysis* framework.

2.3 "Make", "Break" and "Prevents"

Let us now revisit two link types that are part of the *i** ontology[2] ([18] has a condensed reference) and will prove useful for our purposes: the *"break"* and the *"make"* links. These are directed links that describe how the satisfaction of one goal can (critically) influence the satisfaction of another goal. The former means that the satisfaction of the goal-source cancels the satisfaction of the goal-target. The latter means that the satisfaction of the goal-source implies the satisfaction of the goal-target.

Let us now define the "break" and "make" between tasks using the notion of task satisfaction we discussed in the previous section:

[2] Actually they are part of *Goal Oriented Requirements Language* (*GRL*) which is an enriched version of the *i** language.

Break. A "break" link from task t_1 to task t_2, will represent the fact that if t_1 is satisfied then at the same time t_2 is not satisfied and cannot be satisfied (unless t_1 ceases to be satisfied). Thus, there is no point in time in which both t_1 and t_2 are satisfied. We write $t_1 \stackrel{br}{\rightarrow} t_2$ to represent the respective relation over the set of tasks of a goal graph.

Make. A "make" link from task t_1 to task t_2, will represent the fact that if t_1 is satisfied then at the same time t_2 is also satisfied (and thus does not need to be performed anymore). Thus, there is no state in which t_1 is satisfied without t_2 being also satisfied. We write $t_1 \stackrel{mk}{\rightarrow} t_2$ to represent the respective relation over the set of tasks of a goal graph.

"Make" is reflexive and transitive. "Break", one the other hand does not satisfy any of these. Moreover we will violate the definition to introduce $t_1 \stackrel{br}{\rightarrow} t_1$. This will mean that t_1 can remain satisfied for only one time unit. The reasons why this proved necessary will be explained later.

Finally let us define the relation "Prevents" (which is the dual of precedence):

Definition 2.3 We say that a task t_1 *prevents* a task t_2, when t_2 can be performed only if t_1 is not satisfied. We write $t_1 \stackrel{pv}{\rightarrow} t_2$.

Notice here the use of task performance (as opposed to task satisfaction). This means that as opposed to the "break" link, it is possible for both tasks to be satisfied at a given time instance. For example, t_1 is performed/satisfied exactly after t_2 was (legally) performed. To disallow this we may want to demand $t_2 \stackrel{br}{\rightarrow} t_1$.

We will see a complete example in the next section.

2.4 Events

Let us now move further and add one more elements in our precedence diagram. In particular we still need to express:

1. The different outcomes the performance of a task might have that should influence the desired flow of activity. For instance we should not assume that a `[Get Credit Card Authorization]` is something that will trivially be performed. The question is:

2. Events that do not relate to the performance of a task but still influence the flow of activity. For example just before the travel agent issues the ticket, a cheaper seat becomes available in a different flight. Such events introduce goals that have to be satisfied in response to their occurrence.

Although in the bibliography (e.g. [15], KAOS) one the generic terms pre-condition, post-condition or trigger are used, here we will use the term *events*. Events express non-deterministic, non-intentional (subject to the actors within our scope) factors that influence the satisfaction of goals.

Exactly as we distinguish performance from satisfaction in tasks, we distinguish *occurrence* from *evidence* (effect) in events. Again an occurrence is necessary for the evidence to appear, but the evidence can be canceled (as opposed to the historical fact of the occurrence). Moreover, the evidence may appear without the event to occur (a task or another event might have been responsible). Therefore, events can be used in any side of the relations/links defined so far, with the difference that the terms "performed" and "satisfied" should be accordingly replaced with "occurred" and "evident" respectively for the definitions to accommodate events.

In the precedence diagrams, we represent events with triangles. Figure 5 shows an elaborated detail of the precedence diagram we saw in previous sections. The upper figure shows the precedence and the "prevents" links whereas the lower one focuses on "break" and "make" links.

So, for example, the agent will provide to the customer a list of available flights provided that she has performed some query (precedence link) and the event that no records returned did not occur ("prevents" link). The event cannot occur unless the search has been performed. The fact that the agent will read out a list of results means that the occurrence of the event ⟨No Suitable Flights (Found)⟩ will not be possible from that point ("break" link). However if the event ⟨No Suitable Flights⟩ becomes true earlier, then the tasks `[Give Criteria]` and `[Search for Available Flights]` are "broken" in a sense that they need to be performed again (if we need to satisfy some root goals). There is a self-loop at each event to denote that after it occurs and brings the appropriate effect to the tasks, it also takes its part to this effect at the next state.

Finally, notice that if the customer initiates to give her name and telephone number before the agent asks, the result sought by the task `[Ask Client's Info]` is now achieved ("make" link).

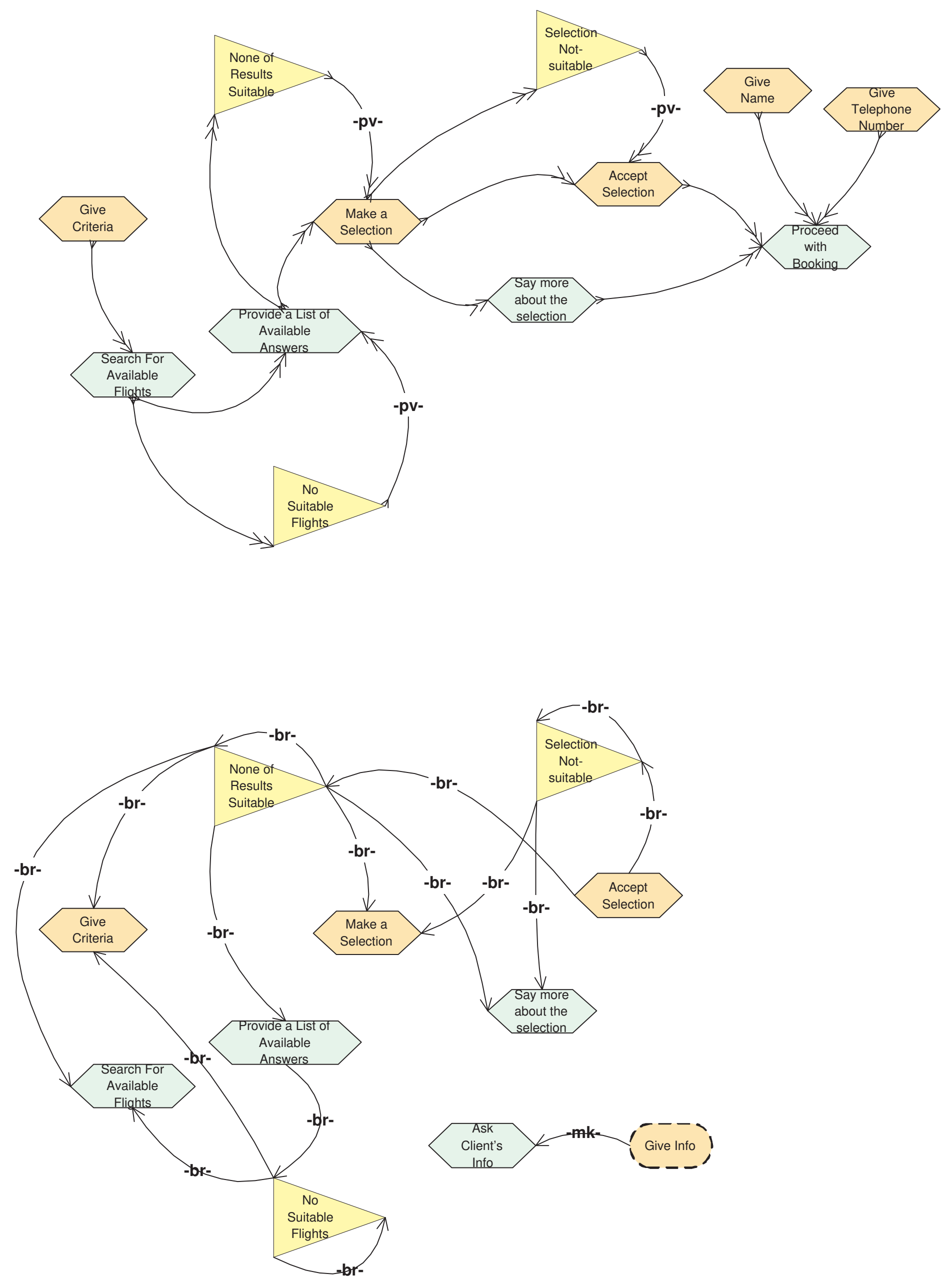

Figure 5: “`[Book Ticket]` precedence diagram”

3 Deriving the State Model

3.1 Preliminaries

We will now try to derive a Kirpke structure from a precedence diagram and a goal model.

First (re)define the sets T, E, G of *boolean variables* (*labels*) each one representing the leaf level tasks, the events and the non-leaf goals that appear in the precedence diagram, respectively. We will use the terms "label" and "variable" interchangeably. Also, define the following semantics that should be understood subject to the discussion of the previous sections:

- Variable $t \in T$ is $true$ if and only if the task it represents is satisfied.
- Variable $e \in E$ is $true$ if and only if the event it represents is evident.

Moreover, define a set $\hat{G}$ of boolean functions that represent non-leaf goals that appear in our precedence diagram. We construct the set as follows: for each non-leaf goal that appears in the precedence diagram, go to the goal graph and consider the AND/OR subtree that is rooted at this goal. Construct the equivalent function over the leaf-level tasks $\hat{g}(T_G)$, $T_G \subseteq T$ of the subtree and add it to $\hat{G}$. Then define a one-to-one mapping between functions in $\hat{G}$ and variables in G as follows:

- Variable $g \in G$ is $true$ if and only if the corresponding $\hat{g}(T_G)$ evaluates to $true$.

Let us finally recall the relations between the several entities in the precedence diagram by refining their semantics for our boolean variables. For $a_1, a_2 \in T \cup E \cup G$:

- $a_1 \succ a_2$ if and only if there is a precedence link from the entity represented by a_1 to the entity represented by a_2.
- $a_1 \stackrel{br}{\rightarrow} a_2$ if and only if there is a "break" link from the entity represented by a_1 to the entity represented by a_2.
- $a_1 \stackrel{mk}{\rightarrow} a_2$ if and only if there is a "make" link from the entity represented by a_1 to the entity represented by a_2.

- $a_1 \stackrel{pv}{\rightarrow} a_2$ if and only if there is a "prevents" link from the entity represented by a_1 to the entity represented by a_2.

The properties of these relations are discussed in previous sections.

Finally, recall that A_G is a minimal set of tasks that satisfy the root goal of the decomposition graph. We extend this set to:

$$L_G = A_G \cup \{g \in G \text{ such that } g \text{ is an ancestor of some } t \in A_G\}$$

Selecting exactly one of each OR-node's children and traversing the decomposition graph from its root accordingly, the subtree that forms reflects such a set. We call L_G the *alternative subtree* for A_G.

3.2 The Kirpke Structure

We can now define a *Kirpke structure* for an alternative A_G to be a tuple $\langle S, A, s_0, I, R\rangle$ where:

- S is a set of states.
- $A = T \cup E \cup G$ is a set of labels.
- s_0 is a set of initial states. This will be the state in which all labels are *false*.
- I a labeling function $S \mapsto 2^A$
- R a transition relation in $S \times S$, which is defined below.

Let (s, s') be an arbitrary pair of states. Let $a_1 = v_1, \ldots, a_n = v_n$ and $a_1 = v'_1, \ldots, a_n = v'_n$ be the corresponding label values. $(s, s') \in R$ (transition from s to s' is possible) if and only if for all i one of the following is true:

- $a_i \in G$.
- $v_i = false$, $v'_i = false$ and $\forall j$ such that $a_j \stackrel{mk}{\rightarrow} a_i$, $v'_j = false$.
- $v_i = true$, $v'_i = true$ and $\forall j$ such that $a_j \stackrel{br}{\rightarrow} a_i$, $v'_j = false$.
- $v_i = true$, $v'_i = false$ and either ($\exists j$ such that $a_j \stackrel{br}{\rightarrow} a_i$, $v'_j = true$) `or` ($a_i \stackrel{br}{\rightarrow} a_i$).

- $v_i = false$, $v'_i = true$ and *either*:
  - $\exists j$ such that $a_j \overset{mk}{\rightarrow} a_i$, $v'_j = true$

  or all of the following are true:

  - $a_i \in A_G$
  - $\forall j$ such that $(a_j \succ a_i$ and $a_j \in L_G)$, $v_j = true$.
  - $\forall l$ such that $a_l \overset{br}{\rightarrow} a_i$, $v'_l = false$.
  - $\forall k$ such that $a_k \overset{pv}{\rightarrow} a_i$, $v_k = false$.

Notice that when the variable in question receives both a "make" and a "break" link, we bias the definition towards the occurrence of a transition. Also, "make", "break" and "prevents" links do not need to originate from nodes of the same alternative in order to take place in the transition conditions.

3.3 Producing the SMV Model

Let us know see how we can construct a transition template for creating the equivalent SMV model.

Let us first construct the alternative. Let o be an SMV variable representing an OR node with domain $Dom(o) = \{a \mid a \text{ direct child of } o\} \cup \{NA\}$. That is each of these variables takes as values the signatures[3] of the variables that represent its children plus the reserved value 'NA'. Thus we declare:

$\mathtt{o} : \{\text{`}\mathtt{child_1}\text{'}, \text{`}\mathtt{child_2}\text{'}, \ldots, \text{`}\mathtt{NA}\text{'}\}$

Let o_{pre} be the closest ancestor of o that is also an OR-node (and thus is represented as discussed above). Let also $o_v \in Dom(o_{pre})$ and o_v represents either an ancestor of o, or o itself. Then we use the following template to construct the transition assignment for o:

```
init(o) := 'NA';
next(o) := case
       (o = 'NA') ∧ (o_pre = 'o_v') : Dom(o)\{'NA'};
       1 : o;
```

[3]In practice we change the signatures to avoid confusions between identifiers and values.

```
esac;
```

If, however, o has no OR-ancestor then:

$\texttt{init(o)} := \text{'}\texttt{NA}\text{'};$
$\texttt{next(o)} := \texttt{case}$
$\quad (\texttt{o} = \text{'}\texttt{NA}\text{'}) : \texttt{Dom(o)} \backslash \{\text{'}\texttt{NA}\text{'}\};$
$\quad \texttt{1 : o;}$
$\texttt{esac;}$

Let now $a \in T \cup E$ be a variable in the Kirpke structure that represents a leaf-level task or an event. We declare each of these to be a boolean variable for our SMV model. For the non-leaf goals $g \in G$, we introduce SMV macro definitions that implement the functions $\hat{g} \in \hat{G}$ we saw in previous section. We might want to extend G in order to cover goals that do not appear in the precedence diagram but appear in the goal model. Then let:

- P be the set of all variables/macros p for which $p \succ a$.
- D be the set of all variables/macros d for which $d \overset{pv}{\rightarrow} a$.
- B be the set of all variables/macros b for which $b \overset{br}{\rightarrow} a$.
- M be the set of all variables/macros m for which $m \overset{mk}{\rightarrow} a$.
- a_{pre} be the closest OR-ancestor of a and $a_v \in Dom(a_{pre})$ such that a_v is either an ancestor of a, or a itself. Similarly, define p_{pre} and p_v for $p \in P$.

Then we can use the following rule to construct a's initialization and transition assignments:

$\texttt{init(a)} := \texttt{0};$
$\texttt{next(a)} := \texttt{case}$
$\quad (\texttt{a}) \wedge (\bigvee_{\texttt{b}\in\texttt{B}} \texttt{next(b)}) : \texttt{0};$
$\quad (\texttt{!a}) \wedge (\bigvee_{\texttt{m}\in\texttt{M}} \texttt{next(m)}) : \texttt{1};$
$\quad (\texttt{a}_{\texttt{pre}} = \text{'}\texttt{a}_{\texttt{v}}\text{'}) \wedge (\texttt{!a}) \wedge \bigwedge_{\texttt{p}\in\texttt{P}} [(\texttt{p}_{\texttt{pre}} \neq \text{'}\texttt{p}_{\texttt{v}}\text{'}) \vee \texttt{p}] \wedge \bigwedge_{\texttt{d}\in\texttt{D}} \texttt{!d} \wedge \texttt{!}(\bigvee_{\texttt{b}\in\texttt{B}} \texttt{next(b)}) : \{\texttt{0}, \texttt{1}\};$

```
        1 : a if a does not have a self break
        1 : 0 if a has a self break
esac;
```

If a or p does not have an OR-ancestor we simply ignore the corresponding term.

The use of the `next` operator in the transition conditions introduces some restrictions to the way we order the transition assignment for each variable. Trivially if a precedence diagram contains cycles of “make” and “break” links, then the SMV model cannot be produced. Thus, although in theory we can demand that a non-deterministic choice can be made for the ordering of the transition assignments, here we will simply demand that the precedence diagram does not contain such cycles.

Also, there is a detail concerning $\stackrel{mk}{\to}$, that must be attended here. In particular a chain of “make” links should imply that the “make” effect is propagated simultaneously in all members of the chain. This requires that before we produce the SMV model, we introduce redundant “make” links so that for every node of the original “make” chain there is a “make” link to each of its (originally indirect) successors. Moreover, for every “break” link that begins from nodes in the chain, “break” links to the appropriate targets must be added accordingly.

Complete examples of derived SMV models can be found in the Appendices. Note that they do not always reflect the above templates (some terms might be absent), but still they are correct for the example diagrams.

4 Analysis

4.1 Basic CTL Formulae

The CTL properties we will consider here concern mostly the way with which our root goals are satisfied. That is we will examine what holds true of false along paths that lead to a state where the macro definition of the goal model valuates to true.

After a closer look to the definition of the Kirpke structure we notice that in most cases each state comes with a self loop. That is, along every path there might be an arbitrary and possibly infinite number of transitions on which nothing changes. This has the following implications:

- The operator `AX` might return a (conceptually) vacuous *false*. For example, even if we know that at the next "state" only the (currently non-satisfied) task `q` can be performed, `AX q` will still refuse to be *true* assuming that one of the next states is not doing anything.

- There are paths that end up looping infinitely around a self loop. This also results to a restricted use of `AF`. For example the assertion "if `that` happened then it is guaranteed that goal `g` is satisfied some time in the future" cannot be expressed through `that -> AF g`.

The problem with these self-loops is that they do not serve a particular conceptual purpose. We introduce them to cancel any synchronization phenomena in our analysis. More specifically, by disallowing the system to remain idle, that is by replacing {`0,1`} with {`1`}, the length of the paths in the precedence diagram would become meaningful for our analysis. Which is of course undesirable, since it destroys the idea of parallelism over the activity of the participating actors. (We come back to this issue in the last section.)

Nevertheless, we can still employ the existential operators (and `AG`) to produce formulas that are meaningful. So, for example:

1. "The goal `g` can be satisfied":

```
EF g
```

2. "The goal `g` is satisfied only if `p` is satisfied":

```
AG (g -> p)
```

3. "Although an event `e` occurs, the goal `g` can still be satisfied":

```
AG (e -> EG g)
```

4. "A task `t` may be preformed/satisfied more than once, for the goal `g` to be satisfied":

```
EF (t & EF(!t & EF g))
```

5. "It is possible for `p` to be performed/occur before `q`, still the goal `g` is eventually satisfied":

```
E[!q U p &!q & EF(g)]
```

6. "q must be observed satisfied/evident at least once, before g is eventually satisfied":

```
!E[!q U g]
```

... and so on.

4.2 Paying for the airplane ticket

Let us now see a hypothetical[4] scenario on how model checking and query checking can be used in order to explore the state-transition model we have created.

In Figures 6 and 7 a revised version of the goal model and the corresponding precedence diagram for the `[Make Payment]` and `[Receive Payment]` subgoals is depicted. Notice that we have done an important change to the conceptual model: there are pairs of tasks that describe interactions between the Traveler and the Agent. These where introduced because initially each decomposition is developed subject to one actor at a time. However a close examination leads us to the observation that when these interaction pairs are performed simultaneously they rather refer to the same task. For instance `[Give PIN]` and `[Get PIN]` or `[Give Cash]` and `[Get Cash]`. Moreover it is clear that, in this, the task will be performed by one of the interacting actors. Note that this process of redefining the delegation schema is part of the Tropos methodology.

The SMV model can be found in Appendix A.

From the whole model let us focus on one detail: the ticket can be issued once the customer claims to have paid the ticket (i.e. there are no pending tasks from her side) independent on whether the agent has received the payment. Thus, the model is now asked: "is there a possibility that the ticket is issued, the customer has paid it but still the payment hasn't been received? If yes which payment methods introduce this possibility?". The query that will help us through that would be the following:

[4]Unfortunately XChek would not compile the particular model, thus the "example" that is illustrated in this section is based on manual exploration using NuSMV.

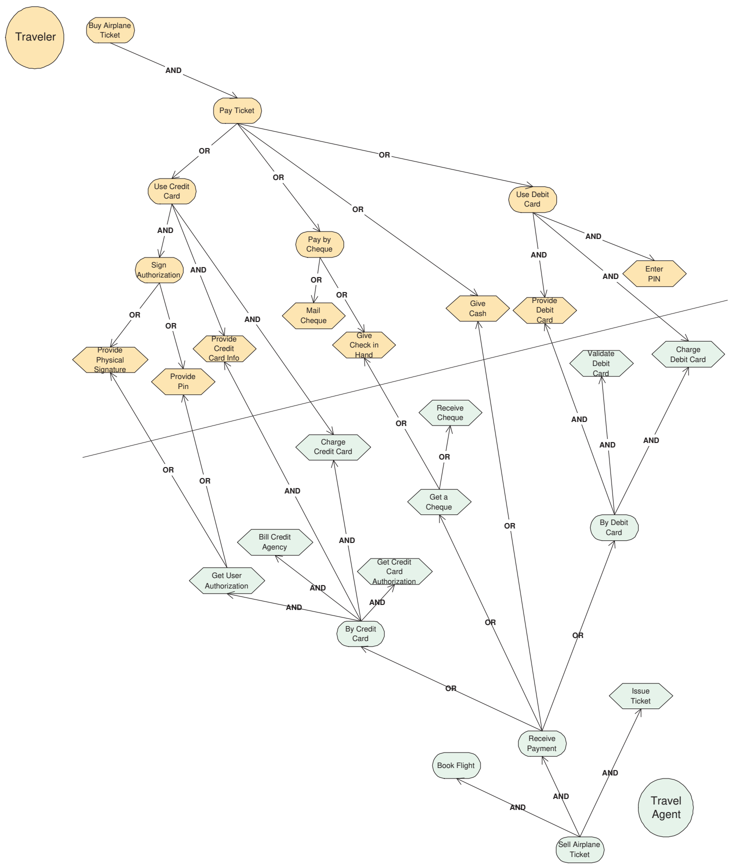

Figure 6: Goal decomposition for `[Pay Ticket]`

```
EF (?x{Traveler_UseCreditCard, Traveler_UseDebitCard,
Traveler_GiveCash, Traveler_PayByCheque} & Agent_IssueTicket &
!Agent_ReceivePayment & EF(PaymentOK))
```

Of course, the predicates for the placeholder are macros, which does not seem to be allowed in TQSolver. Note also that `PaymentOk` refers to whether both the Traveler and the Agent have fullfiled their payment goals. In theory, and some manual exploration using NuSMV, the result should be something like the following:

```
?x = Traveler_UseCreditCard xor Traveler_PayByCheque
```

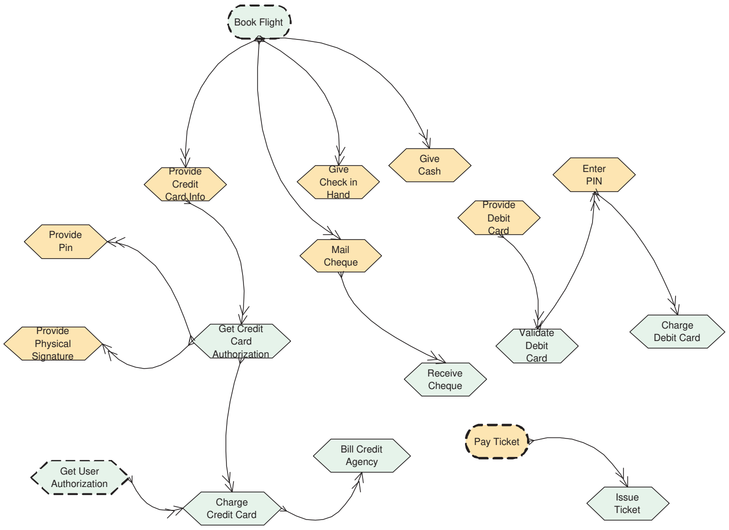

Figure 7: Precedence diagram for `[Pay Ticket]`

Next the modeler will be asked why this might happen. Presumably there are tasks that concern the Agents payment that are pending when the ticket is issued. Thus we would ask questions like:

```
EF(?x{Agent_BillCreditAgency, Agent_ChargeCreditCard} &
Traveler_UseCreditCard & Agent_IssueTicket & !Agent_ReceivePayment &
EF(PaymentOK))
```

Should return:

```
?x = (¬Agent_BillCreditAgency ∧¬Agent_ChargeCreditCard) ∨
(¬Agent_BillCreditAgency ∧ Agent_ChargeCreditCard)
```

... which means that any of these two tasks may be pending given their precedence. Similarly:

```
EF(?x{Agent_ReceiveCheque, Agent_GiveChequeInHand,
Traveler_MailCheque} & Traveler_PayByCheque & Agent_IssueTicket &
!Agent_ReceivePayment & EF(PaymentOK))
```

Should return something equivalent to:

?x = ¬Agent_ReceiveCheque ∧¬Agent_GiveChequeInHand ∧
Traveler_MailCheque

Meaning that this situation might only occur when a cheque has been mailed but not yet received (and was not given in person).

The response of the analyst would be to have a more strict constraint on when the ticket is issued. She would rather add a precedence link from [Receive Payment] to [Issue Ticket].

Note, however, that we where unable to see how TLQSolver answered and, probably more importantly, how much time it would need to give us such answers.

4.3 Booking the Flight

Here we will bother with another branch of the initial [Buy an Airplane Ticket] goal, namely the [Book Ticket Part]. It is the example we saw in Figure 5 page 15. The decomposition graph is shown in Figure 8. Both diagrams are result of elaboration described in the previous example: pairs of tasks representing simultaneous interactions between agents are replaced by tasks delegated to the actor who actually performs the task.

This example does not involve alternatives but it illustrates the use of events, "make" and "break" links.

In this example although we were able to load the model, TQSolver did not seem to answer the answers correctly. We had to employ NuSMV again in order to correct the answers. Moreover TQSolver does not support a) macros in the query formula b) more than 2 nested temporal operators under certain circumstances. The SMV model can be found in Appendix B.

We introduce a few query types that express possible questions and concerns of the stakeholders trying to understand the model.

> "I am the travel agent. What can I do right after a particular task has been satisfied?"

```
EF (!Traveler_MakeASelection & EX(Traveler_MakeASelection & EX
  (?x{Agent_SayMoreAboutTheSelection, Traveler_AcceptSelection,
 Agent_ProceedWithBooking}) & EF (DependencyGoalsAccomplished)))
```

This query is not supported for reasons described above. The answer we would except is something like:

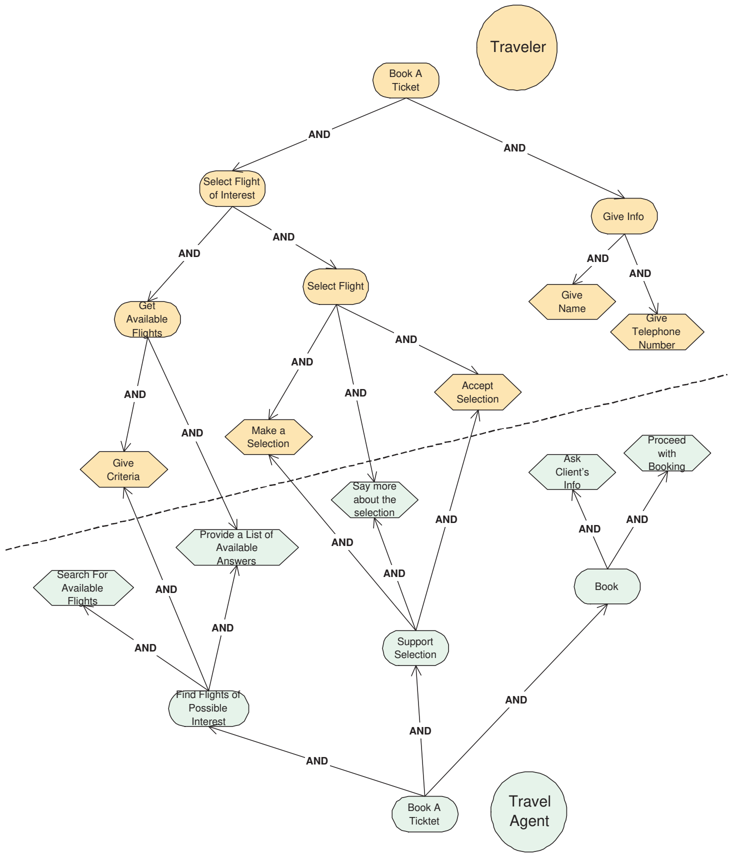

Figure 8: The decomposition graph for `[Book Ticket]`

```
Agent_SayMoreAboutTheSelection | Traveler_AcceptSelection
```

"I performed a task twice. Why, what happened?"

We ask TQSolver the following question:

```
EF(Agent_SearchForAvailableFlights & (EX ?x{Evt_NoSuitableFlights,
  Evt_NoneOfResultsSuitable, Agent_SearchForAvailableFlights}))
```

Returns:

```
Done in:  0.017s
Solution 1 is:
!Evt_NoSuitableFlights & Evt_NoneOfResultsSuitable &
!Agent_SearchForAvailableFlights
Solution 2 is:
!Evt_NoSuitableFlights & !Evt_NoneOfResultsSuitable
Agent_SearchForAvailableFlights
```

We ignore Solution 2 and focus on solution 1 where a case that the task `[Search For Available Flights]` becomes negated is given. TQSolver, though, fails to give a third solution:

```
Evt_NoSuitableFlights & !Evt_NoneOfResultsSuitable &
        !Agent_SearchForAvailableFlights
```

which NuSMV thinks to be an applicable one and is not implied by any of the above.

"I am the agent. An event is evident. What happens next?"

For instance:

```
EF(Evt_NoSuitableFlights & EX ?x{Agent_SearchForAvailableFlights,
                    Traveler_GiveCriteria})
```

Which should return:

```
¬ Agent_SearchForAvailableFlights
```

> "I am giving you the following case: The agent starts searching for available flights but the traveler hasnt given any criteria. Is this implied by our model?"

Here we have a simple SMV property:

```
EF(Agent_SearchForAvailableFlights & !Traveler_GiveCriteria &
            EF(DependencyGoalsAccomplished))
```

which proves false. Or we may ask:

> "The user makes a selection and right away, without asking anything more, she requests to book for the flight in question. Can you book a flight without informing the traveler about all the details?"

```
EF(!Agent_SayMoreAboutSelection & Agent_ProceedWithBooking &
            EF(DependencyGoalsAccomplished))
```

which proves false.

4.4 State Explosion and Workarounds

The SMV model of an arbitrary precedence diagram will contain as many boolean variables as the nodes that appear in the diagram. Moreover, at least in principle, the less precedence relations we include in the diagram, the closer the number of states will be to its upper bound.

To this we must add a general requirement for *immediate* results. As opposed to verifying a complete modeling result, in which case we may accept a significant time investment, when exploring a model possibly while being created we need to be able to get fast add-hoc answers to our questions.

For example, creating the SMV model of the full precedence schema as it appears in Figure 2 (more than 50 variables), although it has strict precedence constraints, leads us to more than a few minutes of computation[5] which is not acceptable for our purposes.

Although a discussion about abstraction in light of the existence of the precedence and decomposition models, might prove very interesting (even a

[5] Pentium 4, 2GHz, 1Gbyte

reason for adapting a framework like the one attempted in this report), here we only give some simple examples of theoretical and empirical reasoning that can help us reduce the size of the state models.

Consider a goal model and its precedence diagram. Let g_1, g_2 and g_3 be variables representing high level goals of the goal model. Let $g_1 \succ g_2 \succ g_3$. Assume also that none of the tasks T_{g_2} under g_2, are sources or targets of "break" links. Then, to check any *path* property that does not contain any of the tasks in T_{g_2}, in the underlying model we may replace the set T_{g_2} with a single variable g_2. The same *summarization* of precedence chains can be applied even if the participating tasks do not form the set T_g of any high level goal g.

The assumption may not be as strong as it seems. For instance, goals `[Receive Payment]` for the Travel agent and `[Obtain Ticket]` for the Traveler, clearly satisfy a precedence relationship and no obvious "break" link can be established in a sense that there is nothing in the `[Obtain Ticket]` phase that can cancel the satisfaction of the goal `[Receive Payment]`. Exceptions would rather be dealt by introducing event-handling goals like `[Reissue and Resent Ticket]`, which we do not discuss here. In Figure 4 (page 11) we could remove all tasks that are under the goal `[Book Flight]` if we want to examine the behavior of the system after the flight has been booked (therefore dealing with `[Book Flight]` as if it were an atomic operationalized task).

Abstraction techniques based on the *cone of influence* ([8]) can also be employed. By removing the semantics from the links that appear in the precedence diagram, the resulting graph can be viewed as a *dependency graph* over the variables that define the state space. Then if we want to explore the model until a particular task is performed, we can consider the subgraph that only contains nodes from which the node that represents that task is reachable. For instance, in Figure 4 page 11 again, if we want to examine the behavior of the model until the ticket has been booked, we may omit the tasks that relate to the payment of the ticket.

5 Conclusions and Future Work

5.1 Tropos for Modeling Behavior

In this report we introduced an extension of the *i** ontology so that we can introduce allowable sequences with which tasks and goals are satisfied. This alone comes with a rich agenda for further research. What can we represent using this primitive precedence link? Are there things (behavioral requirements) that we cannot represent? (a quick answer is yes, *of course*) How easy and intuitive is it to represent what we claim that is, though, representable? And more importantly: is this constraint-based paradigm indeed useful for modeling requirements and how can it be combined with its scenario-constructive dual (i.e. to introduce a number of scenarios and demand that any behavior that does not comply with one of them is unexpected/undesired)?

But even if the notion of precedence proves useful, a more detailed examination must follow. For example, is there a distinction between intentional and non-intentional precedence? Because there should be a difference between the statement "you cannot receive a letter unless someone sends it to you first" and the statement "we cannot issue the ticket, unless we explain the customer all the details about it". Trivially the former is a domain axiom and the latter a requirement. Furthermore how do we refine precedence relations and how can this refinement be combined with goal reduction? For instance, when trying to model the lifecycle of an IT project we initially say that, "roughly", the goal "Design Software" precedes the goal "Develop Software". However when it gets to define the details (the tasks) of each phase, it proves that the two phases may overlap, depending on e.g. the implementation method.

5.2 Behavioral and Functional Variability

The reason why we chose a constraint-based approach is because we wanted to allow a high degree *implicit variability* in our models. The more the allowable courses of tasks, the more flexible (customizable) the software that satisfies the modeled requirements will be. But the problem then is how we manage this variability in order to personalize it correctly each time.

In functional variability this has two phases: *filtering* certainly undesirable alternatives (e.g. the ones that contain tasks the user cannot perform) and *optimizing* among the remaining ones (e.g. pick the one that satisfies the user preferences best).

In behavioral variability, the way it is presented in this report, we already have a proposal on tackling the filtering issue: disallow the alternatives that do not conform to temporal properties posed by the user. This may influence functional filtering as well, as we saw in our examples. But what about optimization? How can we say that one scenario is more desired than another? For example can a personalization profile be a vector of *utilities over temporal properties*. For example, it may prove that we can objectively define that scenarios satisfying φ are graded with marks x and y subject to their `[Security]` and `[Convenience]` respectively (which notice that are non-functional goals). Then ask the user to talk only about how important is Security and Convenience for her.

And finally how do we add another interesting dimension to the issue of variability: *dynamic task delegation*. Is it interesting to distinguish between identical scenarios based on who (e.g. the system versus the user) performs its elementary tasks. And again how do we pose filters and utilities here?

Obviously, more extended and detailed case studies, plus a different approach on temporal analysis tools is required to prove the usefulness of the above.

5.3 From Semi-Formal to Formal

This report has claims the need for a semi-formal approach that may precede goal-oriented formalisms (i.e. KAOS) and serves a different motivation. However the connection between the two has to be established. This has two directions:

- First, how can such semi-formal framework serve as a "front-end" to formalization frameworks? For example, since the precedence relation correspond to a known LTL pattern (specifically $\mathtt{S} \succ \mathtt{P}$ can be translated to `(!P W S)` according to [1]) then a precedence diagram may automatically cast to a KAOS specification skeleton. KAOS analysts will need to refine the LTL specifications to reflect late/more specific requirements or requirements that the semi-formal model could not express. In parallel, the formal specification experts will need to introduce all those non-intentional elements and properties that were hidden (on purpose) in the semi-formal model.

- Conversely, what other LTL specification patterns could we adopt as links between our goals, tasks and events? For instance could we afford an "in between invariance" relation between three such entities, how would this be represented diagrammatically, how intuitive would it be and how frequently this would be used (as a measure of the need to introduce it)?

Finally, can the derivation of the Kirpke structure be adapted so that formalists can already assume that particular properties are preserved? And even within the semi-formal framework, what are the goal reduction and precedence introduction rules that guarantee the preservation of properties? How can goal decomposition and precedence links help us come up with the appropriate abstraction/concretization functions that both help us cope with the state explosion problem (which was evident in our example) and have a conceptual basis?

5.4 The state based analysis framework

The self-loops that are introduced in each state prevent us from using particular CLT formulas that involve universal quantifiers. To avoid this we could remove non-determinism and introduce as many processes as the tasks, supplied with the fairness constraint to be active infinitely often. However, the result of such a practice in terms of required computational resources is unknown.

Moreover, it should be the case that the satisfaction of particular temporal formulas can be guaranteed without the need to apply any transformation to the Kirpke structure, by simple diagrammatic reasoning. For instance a chain of precedence links, where no other links are involved, is guaranteed to satisfy particular existential response properties.

Finally, a necessary future task is a detailed examination of the constraints that the derivation procedure currently poses to the modeling framework, and whether these constraints are reasonable or not. For example, cycles of "break" links are currently forbidden.

On the other hand particular decisions concerning the mapping from precedence diagrams to Kirpke structures should be reexamined. For example, should "make" and "break" links indeed influence tasks that belong to different alternatives? Is this a realistic case?

5.5 Query Checking

The basic limitation of restricting the number of predicates of interest did not prove a problem for state properties and models of the size of the examples we saw. The opinion of this report is that TQSolver could operate in two modes:

A naive one that would examine each predicate of the set separately. Such a mode would probably afford `?x{*}` that is, an unrestricted set of interest. For instance the question "if this happens what might happen next (among an unrestricted set of things)?" would fit here. Such queries are the most useful in our context.

An advanced one that would examine formulas over a restricted set of predicates, as it does now.

Unfortunately, not having the opportunity to extensively work on TQ-Solver we cannot comment more on its performance with respect to the size of the model, the size of the placeholder set etc.

Appendix

A SMV model for the [Pay Ticket] subtree

```
MODULE main
VAR
Traveler_ProvideCreditCardInfo: boolean;
Traveler_ProvidePhysicalSignature: boolean;
Traveler_ProvidePIN: boolean;
Traveler_MailCheque: boolean;
Traveler_GiveChequeInPerson: boolean;
Traveler_ProvideDebitCard: boolean;
Traveler_EnterDebitPIN: boolean;
Traveler_GiveCash: boolean;


Agent_GetCreditCardAuthorization:boolean;
Agent_ChargeCreditCard:boolean;
Agent_BillCreditAgency:boolean;

Agent_ReceiveCheque:boolean;
Agent_ValidateDebitCard:boolean;
Agent_ChargeDebitCard:boolean;
Agent_IssueTicket:boolean;
Agent_BookFlight:boolean;

-- OR_Nodes
Alt_Trav_MakePayMent:{NA, Credit, Cheque, Debit, Cash};
Alt_Trav_CreditSignAuth:{NA, Physical, Pin};
Alt_Trav_CheckSend:{NA, Mail, InHand};
Alt_Agent_RecPayMent:{NA, Credit, Cheque, Debit, Cash};
Alt_Agent_CreditSignAuth:{NA, Physical, Pin};
Alt_Agent_ChekRec:{NA, Mail, InHand};

ASSIGN


init(Traveler_MailCheque):= 0;
init(Traveler_GiveChequeInPerson):= 0;
init(Traveler_GiveCash):= 0;
init(Traveler_ProvideDebitCard):= 0;
init(Traveler_ProvideCreditCardInfo):= 0;
init(Agent_ValidateDebitCard):= 0;
init(Traveler_EnterDebitPIN):= 0;
init(Agent_GetCreditCardAuthorization):= 0;
init(Agent_ChargeCreditCard):= 0;
init(Agent_ChargeDebitCard):= 0;
init(Agent_ReceiveCheque):= 0;
init(Traveler_ProvidePhysicalSignature):= 0;
init(Traveler_ProvidePIN):= 0;
init(Agent_BillCreditAgency):= 0;
init(Agent_IssueTicket):= 0;
init(Agent_BookFlight):= 0;
```

```
init(Alt_Trav_MakePayMent):= NA;
init(Alt_Trav_CreditSignAuth):= NA;
init(Alt_Trav_CheckSend):= NA;
init(Alt_Agent_RecPayMent):= NA;
init(Alt_Agent_CreditSignAuth):= NA;
init(Alt_Agent_ChekRec):= NA;


-- ALTERNATIVE'S SELECTION
next(Alt_Trav_MakePayMent):= case
    (Alt_Trav_MakePayMent = NA):{Credit, Cheque, Debit, Cash};
    1:Alt_Trav_MakePayMent;
esac;

next(Alt_Trav_CreditSignAuth):= case
    (Alt_Trav_CreditSignAuth = NA) & (Alt_Trav_MakePayMent = Credit):{Physical, Pin};
    1:Alt_Trav_CreditSignAuth;
esac;

next(Alt_Trav_CheckSend):= case
    (Alt_Trav_CheckSend = NA) & (Alt_Trav_MakePayMent = Cheque):{Mail, InHand};
    1:Alt_Trav_CheckSend;
esac;


next(Alt_Agent_RecPayMent):= case
    (Alt_Agent_RecPayMent = NA):{Credit, Cheque, Debit, Cash};
    1:Alt_Agent_RecPayMent;
esac;

next(Alt_Agent_CreditSignAuth):= case
    (NA = Alt_Agent_CreditSignAuth)& (Alt_Trav_MakePayMent = Credit):{Physical, Pin};
    1:Alt_Agent_CreditSignAuth;
esac;

next(Alt_Agent_ChekRec):= case
    (NA = Alt_Agent_ChekRec)& (Alt_Trav_MakePayMent = Cheque):{Mail, InHand};
    1:Alt_Agent_ChekRec;
esac;

-- TRANSITIONS


next(Agent_BookFlight):= case
          (!Agent_BookFlight):{0,1};
          1        :Agent_BookFlight;
        esac;


next(Traveler_MailCheque):= case
(Alt_Trav_CheckSend = Mail) &
          (Agent_BookFlight) & (!Traveler_MailCheque) : {0,1};
          1        :Traveler_MailCheque;
        esac;
```

```
next(Traveler_GiveChequeInPerson):= case
(Alt_Trav_CheckSend = InHand) &
(Alt_Agent_ChekRec = InHand) &
          (Agent_BookFlight) & (!Traveler_GiveChequeInPerson) : {0,1};
          1        :Traveler_GiveChequeInPerson;
        esac;

next(Traveler_GiveCash):= case
(Alt_Trav_MakePayMent = Cash) &
(Alt_Agent_RecPayMent = Cash) &
          (Agent_BookFlight) & (!Traveler_GiveCash) : {0,1};
          1        :Traveler_GiveCash;
        esac;


next(Traveler_ProvideDebitCard):= case
(Alt_Trav_MakePayMent = Debit) &
(Alt_Agent_RecPayMent = Debit) &
          (Agent_BookFlight) & (!Traveler_ProvideDebitCard) : {0,1};
          1        :Traveler_ProvideDebitCard;
        esac;

next(Traveler_ProvideCreditCardInfo):= case
(Alt_Trav_MakePayMent = Credit) &
(Alt_Agent_RecPayMent = Credit) &
          (Agent_BookFlight) & (!Traveler_ProvideCreditCardInfo) : {0,1};
          1        :Traveler_ProvideCreditCardInfo;
        esac;


next(Agent_ValidateDebitCard):= case
(Alt_Agent_RecPayMent = Debit) &
          (Traveler_ProvideDebitCard) & (!Agent_ValidateDebitCard) : {0,1};
          1        :Agent_ValidateDebitCard;
        esac;

next(Traveler_EnterDebitPIN):= case
(Alt_Trav_MakePayMent = Debit) &
          (Agent_ValidateDebitCard) & (!Traveler_EnterDebitPIN) : {0,1};
          1        :Traveler_EnterDebitPIN;
        esac;

next(Agent_GetCreditCardAuthorization):= case
(Alt_Agent_RecPayMent = Credit) &
          (Traveler_ProvideCreditCardInfo) & (!Agent_GetCreditCardAuthorization) : {0,1};
          1        :Agent_GetCreditCardAuthorization;
        esac;

next(Agent_ChargeCreditCard):= case
(Alt_Agent_RecPayMent = Credit) &
          (Agent_GetCreditCardAuthorization)&(Agent_GetUserAuthorization)&(!Agent_ChargeCreditCard):{0,1};
          1        :Agent_ChargeCreditCard;
        esac;

next(Agent_ChargeDebitCard):= case
(Alt_Agent_RecPayMent = Debit) &
```

```
          (Traveler_EnterDebitPIN) & (!Agent_ChargeDebitCard) : {0,1};
          1        :Agent_ChargeDebitCard;
esac;


next(Agent_ReceiveCheque):= case
(Alt_Agent_RecPayMent = Cheque) &
          (Traveler_MailCheque) & (!Agent_ReceiveCheque) : {0,1};
          1        :Agent_ReceiveCheque;
        esac;


next(Traveler_ProvidePhysicalSignature):= case
(Alt_Trav_CreditSignAuth = Physical) &
          (Agent_GetCreditCardAuthorization) & (!Traveler_ProvidePhysicalSignature) : {0,1};
          1        :Traveler_ProvidePhysicalSignature;
        esac;

next(Traveler_ProvidePIN):= case
(Alt_Trav_CreditSignAuth = Pin) &
          (Agent_GetCreditCardAuthorization) & (!Traveler_ProvidePIN) : {0,1};
          1        :Traveler_ProvidePIN;
        esac;

next(Agent_IssueTicket):= case
          (Traveler_PayTicket) & (!Agent_IssueTicket) : {0,1};
          1        :Agent_IssueTicket;
        esac;


next(Agent_BillCreditAgency):= case
(Alt_Agent_RecPayMent = Credit) &
          (Agent_ChargeCreditCard) & (!Agent_BillCreditAgency) : {0,1};
          1        :Agent_BillCreditAgency;
        esac;




DEFINE
-- Goal Model Structure.

Traveler_UseCreditCard := Traveler_ProvideCreditCardInfo & Traveler_SignAuthorization;

Traveler_SignAuthorization :=
(Traveler_ProvidePhysicalSignature | Traveler_ProvidePIN) &
((!Traveler_ProvidePhysicalSignature) | (!Traveler_ProvidePIN));

Traveler_PayByCheque :=
(Traveler_MailCheque | Traveler_GiveChequeInPerson) &
((!Traveler_MailCheque) | (!Traveler_GiveChequeInPerson));

Traveler_UseDebitCard:= Traveler_ProvideDebitCard & Traveler_EnterDebitPIN & Agent_ChargeDebitCard;

Traveler_PayTicket :=
(Traveler_UseCreditCard & !Traveler_UseDebitCard & !Traveler_GiveCash & !Traveler_PayByCheque) |
(!Traveler_UseCreditCard & Traveler_UseDebitCard & !Traveler_GiveCash & !Traveler_PayByCheque) |
```

```
(!Traveler_UseCreditCard & !Traveler_UseDebitCard & Traveler_GiveCash & !Traveler_PayByCheque) |
(!Traveler_UseCreditCard & !Traveler_UseDebitCard & !Traveler_GiveCash & Traveler_PayByCheque);

-- For the Agent.


Agent_GetUserAuthorization :=
(Traveler_ProvidePhysicalSignature | Traveler_ProvidePIN) &
((!Traveler_ProvidePhysicalSignature) | (!Traveler_ProvidePIN));


Agent_ReceivePayment :=
(Agent_GetACheque & !Traveler_GiveCash & !Agent_ByDebitCard & !Agent_ByCreditCard) |
(!Agent_GetACheque & Traveler_GiveCash & !Agent_ByDebitCard & !Agent_ByCreditCard) |
(!Agent_GetACheque & !Traveler_GiveCash & Agent_ByDebitCard & !Agent_ByCreditCard) |
(!Agent_GetACheque & !Traveler_GiveCash & !Agent_ByDebitCard & Agent_ByCreditCard) ;

Agent_ByCreditCard :=
Traveler_ProvideCreditCardInfo &
Agent_ChargeCreditCard &
Agent_GetCreditCardAuthorization &
Agent_BillCreditAgency &
Agent_GetUserAuthorization;

Agent_GetACheque :=
(Agent_ReceiveCheque | Traveler_GiveChequeInPerson) &
((!Agent_ReceiveCheque) | (!Traveler_GiveChequeInPerson));

Agent_ByDebitCard :=
Traveler_ProvideDebitCard &
Agent_ValidateDebitCard &
Agent_ChargeDebitCard;

PaymentOK := Agent_ReceivePayment & Traveler_PayTicket;
```

B SMV model for the [Book Ticket] subtree

```
MODULE main
VAR
-- Tasks
  Agent_SearchForAvailableFlights: boolean;
  Agent_ProvideListOfAvailableFlights: boolean;
  Agent_SayMoreAboutTheSelection:boolean;
  Agent_ProceedWithBooking:boolean;
  Agent_AskClientsInfo:boolean;


  Traveler_GiveCriteria: boolean;
  Traveler_MakeASelection: boolean;
  Traveler_AcceptSelection: boolean;
  Traveler_GiveName: boolean;
  Traveler_GiveTelephoneNumber: boolean;
```

```
-- Events
  Evt_NoSuitableFlights: boolean;
  Evt_NoneOfResultsSuitable: boolean;
  Evt_SelectionNotSuitable: boolean;

ASSIGN

init(Agent_SearchForAvailableFlights) :=  0;
init(Agent_ProvideListOfAvailableFlights) :=  0;
init(Agent_SayMoreAboutTheSelection) :=  0;
init(Agent_ProceedWithBooking) :=  0;
init(Agent_AskClientsInfo) :=  0;


init(Traveler_GiveCriteria) :=  0;
init(Traveler_MakeASelection) :=  0;
init(Traveler_AcceptSelection) :=  0;
init(Traveler_GiveName) :=  0;
init(Traveler_GiveTelephoneNumber) :=  0;

init(Evt_NoSuitableFlights) :=  0;
init(Evt_NoneOfResultsSuitable) :=  0;
init(Evt_SelectionNotSuitable) :=  0;


-- Agent Transitions


next(Agent_SearchForAvailableFlights) :=  case
     (Agent_SearchForAvailableFlights)
       & (next(Evt_NoneOfResultsSuitable) | next(Evt_NoSuitableFlights)): 0;
     (!Agent_SearchForAvailableFlights) & (Traveler_GiveCriteria)
       & !(next(Evt_NoneOfResultsSuitable) | next(Evt_NoSuitableFlights)): {0,1};
     1: Agent_SearchForAvailableFlights;
esac;

next(Agent_ProvideListOfAvailableFlights) :=  case
     (Agent_ProvideListOfAvailableFlights) & (next(Evt_NoneOfResultsSuitable)):0;
     (!Agent_ProvideListOfAvailableFlights) & (Agent_SearchForAvailableFlights)
       & (!Evt_NoSuitableFlights) & !(next(Evt_NoneOfResultsSuitable)):{0,1};
     1: Agent_ProvideListOfAvailableFlights;
esac;

next(Agent_SayMoreAboutTheSelection) :=  case
     (Agent_SayMoreAboutTheSelection)
       & (next(Evt_SelectionNotSuitable) | next(Evt_NoneOfResultsSuitable)):0;
     (!Agent_SayMoreAboutTheSelection) & (Traveler_MakeASelection)
       & !(next(Evt_SelectionNotSuitable) | next(Evt_NoneOfResultsSuitable)):{0,1};
     1: Agent_SayMoreAboutTheSelection;
esac;


next(Agent_ProceedWithBooking) :=  case
     (!Agent_ProceedWithBooking) & (Traveler_AcceptSelection)
       & (Agent_SayMoreAboutTheSelection)
       & (Traveler_GiveName) & (Traveler_GiveTelephoneNumber):{0,1};
     1: Agent_ProceedWithBooking;
```

```
esac;

next(Agent_AskClientsInfo) :=  case
     (!Agent_AskClientsInfo) & (next(Traveler_GiveInfo)):1;
     (!Agent_AskClientsInfo):{0,1};
     1: Agent_AskClientsInfo;
esac;



-- Traveler


next(Traveler_GiveCriteria) :=  case
     (Traveler_GiveCriteria)
       & (next(Evt_NoneOfResultsSuitable) | next(Evt_NoSuitableFlights)):0;
     (!Traveler_GiveCriteria)
       & !(next(Evt_NoneOfResultsSuitable) | next(Evt_NoSuitableFlights)): {0,1};
     1: Traveler_GiveCriteria;
esac;

next(Traveler_MakeASelection) :=  case
     (Traveler_MakeASelection)
       & (next(Evt_SelectionNotSuitable) | next(Evt_NoneOfResultsSuitable)):0;
     (!Traveler_MakeASelection) & (Agent_ProvideListOfAvailableFlights)
       & (!Evt_NoneOfResultsSuitable)
       & !(next(Evt_SelectionNotSuitable) | next(Evt_NoneOfResultsSuitable)):{0,1};
     1: Traveler_MakeASelection;
esac;

next(Traveler_AcceptSelection) :=  case
     (!Traveler_AcceptSelection) & (Agent_MakeASelection)
       & (!Evt_SelectionNotSuitable):{0,1};
     1: Traveler_AcceptSelection;
esac;

next(Traveler_GiveName) :=  case
     (!Traveler_GiveName):{0,1};
     1: Traveler_GiveName;
esac;

next(Traveler_GiveTelephoneNumber) :=  case
     (!Traveler_GiveTelephoneNumber):{0,1};
     1: Traveler_GiveTelephoneNumber;
esac;


-- Events


next(Evt_NoSuitableFlights) :=  case
     (Evt_NoSuitableFlights)
       & (next(Agent_ProvideListOfAvailableFlights)|next(Evt_NoneOfResultsSuitable)):0;
                                                            -- Not Supposed to occur
     (!Evt_NoSuitableFlights) & (Agent_SearchForAvailableFlights)
       & !(next(Agent_ProvideListOfAvailableFlights)|next(Evt_NoneOfResultsSuitable)):{0,1};
     1:0;
```

```
esac;

next(Evt_NoneOfResultsSuitable) :=  case
     (Evt_NoSuitableFlights) & next(Traveler_AcceptSelection):0; -- Not Supposed to occur
     (!Evt_NoneOfResultsSuitable) & (Agent_ProvideListOfAvailableFlights)
       & !next(Traveler_AcceptSelection):{0,1};
     1:0;
esac;


next(Evt_SelectionNotSuitable) :=  case
     (Evt_SelectionNotSuitable) & next(Traveler_AcceptSelection):0; -- Not Supposed to occur
     (!Evt_SelectionNotSuitable) & (Traveler_MakeASelection)
       & !next(Traveler_AcceptSelection):{0,1};
     1:0;
esac;


DEFINE

Agent_FindFlightsOfPossibleInterest :=
  Agent_SearchForAvailableFlights &
  Traveler_GiveCriteria &
  Agent_ProvideListOfAvailableFlights;

Agent_SupportSelection :=
  Traveler_MakeASelection &
  Traveler_AcceptSelection &
  Agent_SayMoreAboutTheSelection;

Agent_Book :=
  Agent_ProceedWithBooking &
  Agent_AskClientsInfo;

Agent_BookATicket := Agent_FindFlightsOfPossibleInterest
                     & Agent_SupportSelection
                     & Agent_Book;


Traveler_GetAvailableFlights :=
  Traveler_GiveCriteria &
  Agent_ProvideListOfAvailableFlights;

Traveler_SelectFlight :=
  Traveler_MakeASelection &
  Agent_SayMoreAboutTheSelection &
  Traveler_AcceptSelection;

Traveler_GiveInfo :=
  Traveler_GiveName &
  Traveler_GiveTelephoneNumber;

Traveler_SelectFlightOfInterest := Traveler_GetAvailableFlights & Traveler_SelectFlight;
Traveler_BookATicket := Traveler_SelectFlightOfInterest & Traveler_GiveInfo;


DependencyGoalsAccomplished := Agent_BookATicket & Traveler_BookATicket;
```

References

[1] A specification pattern system. http://www.cis.ksu.edu/santos/spec-patterns/.

[2] Annie I. Antón, W. Michael McCracken, and Colin Potts. Goal decomposition and scenario analysis in business process reengineering. In *International Conference on Advanced Information Systems Engineering (CAiSE 1994)*. Springer Verlag, 1994.

[3] Glenn Bruns and Patrice Godefroid. Temporal logic query cheching. In *16th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2001.

[4] R.J.A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12), 1998.

[5] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: The Tropos project. In *Information Systems*, Elsevier, Amsterdam, The Netherlands, 2002.

[6] William Chan. Temporal-logic queries. In *12th Conference on Computer Aided Verification*, pages 450–463, ser. Lecture Notes in Computer Science, vol 1855. Chicago, IL, USA, 2000. Springer.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[9] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the Workshop on Logic of Programs*. Springer, 1981.

[10] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. Technical Report UM-CS-1998-035, 1998.

[12] Steve Easterbrook, Robyn Lutz, Richard Covington, John Kelly, York Ampo, and David Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1), 1998.

[13] M. Fowler and K. Scott. *UML Distilled.* Addison Wesley, 1997.

[14] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in Tropos, 2001.

[15] Günter Gans, Gehrard Lakemeyer, Matthias Jarke, and Thomas Vits. SNet:a modeling and simulation environment for agent networks based on i* and congolog. In *14th International Conference on Advanced Information Systems Engineering (CAiSE 2002)*, Toronto, Canada, May 2002. Springer Verlag.

[16] Giuseppe De Giacomo, Yves Lesperance, and Hector J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[17] Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Reasoning with goal models. In *Proceedings 21st International Conference on Conceptual Modeling ER 2002*, Tampere, Finland, October 7-11 2002.

[18] Goal-oriented requirements language. http://www.cs.toronto.edu/km/grl/.

[19] Arie Gurfinkel, Benet Devereux, and Marsha Chechik. Model exploration with temporal logic query checking. *SIGSOFT Softw. Eng. Notes*, 27(6):139–148, 2002.

[20] G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers. *Software and Systems Modeling*, 1, 2003.

[21] Bowen Hui, Sotirios Liaskos, and John Mylopoulos. Requirements analysis for customizable software: A goals-skills-preferences framework. In *11th IEEE International Requirements Engineering Conference*, 2003.

[22] Hermann Kaindl. A design process based on a model combining scenarios with goals and functions. *IEEE Transactions on Systems, Man, and Cybernetics*, 30(5), 2000.

[23] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *Proc. FSE'10: 10th ACM SIGSOFT Symp. on the Foundations of Software Engineering*, Charleston, November 2002.

[24] Lin Liu and Eric Yu. Design web-based systems in social context: A goal and scenario based approach. In *14th International Conference on Advanced Information Systems Engineering (CAiSE 2002)*, Toronto, Canada, May 2002. Springer Verlag.

[25] John Mylopoulos, Lawrence Chung, and Brian A. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *Software Engineering*, 18(6):483–497, 1992.

[26] John Mylopoulos, Lawrence Chung, and Eric SK Yu. From object-oriented to goal oriented requirements analysis. *Communications of the ACM*, 1(42):31–37, 1999.

[27] D.L. Parnas and Madey J. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.

[28] J.P. Quielle and J.Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th International Symposium on Programming*, pages 337–350, 1982.

[29] Colette Rolland, Carine Souveyet, and Camille Ben Achour. Guiding goal modeling using scenarios. *IEEE Transactions on Software Engineering*, 24(12), 1998.

[30] Alistair Sutcliffe. Scenario-based requirements analysis. *Requirements Engineering Journal*, 3(1):48–65, 1995.

[31] Axel van Lamsweerde. Building formal requirements models for reliable software. In *Invited Paper for 6th International Conference on Reliable Software Technologies*, Leuven, 2001. Lecture Notes in Computer Science, Vol. 2043, Springer-Verlag.

[32] Axel van Lamsweerde and Laurent Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, 1998.

[33] Eric S. K. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE Int. Symposium on Requirements Engineering (RE'97)*, Washington D.C., USA, January 1997.