Operational Semantics of Goal Models in Adaptive Agents

Mirko Morandini
FBK-IRST
Via Sommarive 18
I-38100 Trento, Italy
morandini@fbk.eu

Loris Penserini
University of Utrecht
Padualaan 14, De Uithof
Utrecht, The Netherlands
loris@cs.uu.nl

Anna Perini
FBK-IRST
Via Sommarive 18
I-38100 Trento, Italy
perini@fbk.eu

ABSTRACT

Several agent-oriented software engineering methodologies address the emerging challenges posed by the increasing need of adaptive software. A common denominator of such methodologies is the paramount importance of the concept of goal model in order to understand the requirements of a software system. Goal models consist of goal graphs representing AND/OR-decomposition of abstract goals down to operationalisable leaf-level goals. Goal models are used primarily in the earlier phases of software engineering, for social modelling, requirements elicitation and analysis, to concretise abstract objectives, to detail them and to capture alternatives for their satisfaction.

Although various agent programming languages incorporate the notion of (leaf-level) goal as a language construct, none of them natively support the definition of goal models. However, the semantic gap between goal models used at design-time and the concept of goal used at implementation and execution time represent a limitation especially in the development of self-adaptive and fault-tolerant systems. In such systems, design-time knowledge on goals and variability becomes relevant at run-time, to take autonomous decisions for achieving high level objectives correctly.

Recently, unifying operational semantics for (leaf) goals have been proposed [15]. We extend this work to define an operational semantics for the behaviour of goals in goal models, maintaining the flexibility of using different goal types and conditions. We use a simple example to illustrate how the proposed approach effectively deals with the semantic gap between design-time goal models and run-time agent implementations.

Categories and Subject Descriptors

I.2.11 [**Artificial intelligence**]: Distributed Artificial Intelligence—*Intelligent agents, Languages and structures*; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms

Theory, Design, Languages

Cite as: Operational Semantics of Goal Models in Adaptive Agents, Mirko Morandini, Loris Penserini and Anna Perini, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. XXX-XXX.

Keywords

Goals, Goal Models, Agent Programming, Formal Semantics

1. INTRODUCTION

The concept of goal, defined as a state of affairs that a system wants to achieve or an action that it aims to perform, has received considerable attention in AI in the past, for instance in distributed reasoning [9] and planning [8]. In agent systems, this concept paved the way for defining agent proactivity and autonomy.

Goals have also been recognized to be a powerful abstraction in requirements analysis to capture users' needs and to analyse them with the aim of deriving appropriate requirements for the system-to-be (e.g. [3]). Goal-Oriented Requirements Engineering (GORE) provides modelling languages to represent and analyse users' goals that may be delegated to the system-to-be and progressively refined by AND/OR-decomposition to sub-goals to concretise system requirements (and ultimately system behaviour). The resulting AND/OR graphs are called goal models (GMs).

At design time, a GM represents the purposes behind a system [2, 1] making the dependencies between system goals and stakeholders' goals explicit. Especially for the development of adaptive and fault-tolerant systems, this knowledge is of high importance since typical design decisions and variability are shifted to run-time to gain in autonomy yet respecting high level goals and requirements. Also, goal OR-decomposition offers an effective way to support the evaluation of alternative solutions (possible system behaviours), offering a key analysis method in high-variability design [13].

In this paper we propose formal semantics for goal model execution that build upon the semantics proposed by Riemsdijk et al. [15] for leaf-level goals. We define an abstract architecture and instantiate it, defining a precise run-time behaviour for the achievement of higher level (non-leaf) goals through the achievement of their sub-goals and the satisfaction of their achievement conditions. We formalise goal decomposition in AND and in OR, with the three main goal types *achieve-goal*, *maintain-goal*, and *perform-goal*.

The context of our work is that of engineering self-adaptive systems –that is, systems able to autonomously achieve the objectives they have been designed for, in a dynamic environment– adapting their behaviour to different circumstances. We adopt an agent-oriented approach that incorporates GORE techniques to model users' needs and system requirements in terms of goal models and try to push the adoption of goal models further into the system development and implementation process. That is, we propose to

implement the system-to-be through software agents that have knowledge of and behave according to their own goal model, defined at design-time.

Having knowledge of its goal model, a software agent can act according to it, exploring alternatives at failures or to adapt to changed circumstances. Also an enrichment of this model, e.g. by run-time learning, would be possible, always retaining traceability to the design time.

A tool-supported approach for the generation of code for goal-directed, self-adaptive systems from goal models has been proposed in previous work [11]. A main limitation pointed out by this work is the gap between the semantics of goal models defined at design time and the execution semantics of current agent programming languages. The operational semantics for goal models illustrated in this paper give a formal definition of the behaviour expected from a GM modelled at design time by defining its run-time behaviour. Having operational semantics for goal models at run-time will also provide the basis for an effective monitoring of the system behaviour, a key element for self-adaptivity [7].

The paper is structured as follows. Section 2 discusses related work. Section 3 gives an intuitive idea of goal model semantics at run-time with the help of a simple cleaner agent example, while Section 4 introduces the formalisation of our goal model semantics. Section 5 applies these semantics to the example. Finally, Section 6 gives conclusions and points out future work directions.

2. RELATED WORK

As our approach deals with the existing gap between the design of goal models –used by requirements and software engineering methodologies– and the run-time goals and plans used by agent oriented programming languages, we provide two main streams of related work.

2.1 Goal models for software requirements and design

Goal-oriented requirements engineering frameworks introduce goal models to derive software requirements from the analysis of the alternative ways to met the users' goals. Among the established approaches we mention KAOS [3] and i^* [17].

In *KAOS*, a GM is composed of goals arranged in AND/OR graphs where a goal node can have several parent nodes as it can occur in several decompositions (called reductions). In the *KAOS* metamodel, the *reduction* meta-relationship allows for goal refinement and for modelling alternative ways of achieving goals. Goals are defined informally (abstract) and need then to be refined into more formal (concrete) ones until reaching subgoals (leaf-goals) that can be operationalised through *constraints*, formulated in terms of objects and actions. The semantics of the GM are described by a first order temporal logic that allows for a formal verification of requirements.

The agent-oriented methodology *Tropos* [1] borrows modelling and analysis techniques from these frameworks and integrates them with an agent-oriented paradigm. A core activity along the modelling process is conceptual modelling, which is performed by using a language that offers concepts such as those of actor, goal, plan, resource, capability, and social dependency between actors, and a graphical notation to model these concepts in actor and goal diagrams. A *Tropos* GM (see Figure 1 for an example) is represented as a forest of AND/OR-decomposed goals along with lateral contributions and dependency relations to other actors. Additionally, a GM contains means-ends relationships that define the plans that can be the means to satisfy a goals. The *Tropos* GM can be formalised by the Formal Tropos language (FT), based on linear temporal logic, to define the allowed states of a system. However, in FT, goal decomposition is not natively supported but has to be modelled by binding goal achievement to subgoal success. Both KAOS and FT propose a semantics for GMs but with a different purpose with respect to our work, since they focus on formal verification of requirements specifications.

Liaskos et al. [10] propose a formal language to specify stakeholder preferences and to reason about them with the purpose of supporting the analysis of behaviour variability at the requirements level. Here, GMs have been found to be effective to study stakeholders' goals and alternative ways to fulfil them, that is, to characterize variability in the problem space. Variability in the solution space is then modelled in terms of features-based specification, following product-line engineering approaches. Nevertheless, it has been left as future work to show how goal model variability at requirements level influences the run-time behaviour of a software system.

2.2 Goals at run-time

Concerning the semantics of goals and related concepts in programming languages, we shall mention the following works. Padgham and Lambrix [12] propose a formal relationship between capabilities and BDI concepts — i.e., beliefs, goals and intentions. This work origins in the philosophical idea that 'can' implies both *ability* and *opportunity*.

In [16], the authors propose an interesting study about procedural and declarative goals in agent programming languages, focusing on a declarative notion of subgoals within the 3APL agent programming language. Such an approach for modelling subgoals endowing achievement conditions is also adopted in other agent programmed languages, such as Jadex [14], where procedural plans can start and control the lifetime of new goals. Despite such languages can implement simple AND-decomposition, they do not natively support reasoning on the knowledge level, provided by the GM semantics.

The unifying framework proposed in [15] is important in order to give a solid definition of goal, taking advantage of several contributions and perspectives that range from software engineering methodologies to agent programming languages. Here, the main idea has been to propose an abstract architecture for the goal life-cycle along with an operational semantics for the principal goal types. We build our operational semantics for goals in goal models on top of this work that considers only "leaf goals", that is, goals directly operationalisable by plans.

3. THE GOAL MODEL AT RUNTIME

In this work we cover GMs that consist of multiple AND/OR-decomposed root goals, as in the *Tropos* methodology. These GMs embed some extensions in respect to the traditional one proposed in literature [1, 3, 10]: goals can be enriched with information on their dynamic behaviour at run-time by defining goal types and conditions. Several types of goals have been defined and used in agent-oriented programming languages.

In this paper we consider the three goal types *achieve*, *perform* and *maintain*, described in [4]. *Achieve-goals* are characterized by an achievement condition that specifies when a certain state of affairs is reached. The satisfaction of the goal can be attempted several times till this condition holds. Moreover, a failure condition can terminate goal achievement, defining it as failed. *Perform-goals* execute some actions, finally reporting their success or failure without evaluating conditions. Last, *maintain-goals* try to maintain a certain state of affairs. In literature, different types of semantics have been attributed to maintain-goals. An agent can act reactively or proactively to maintain a state [5]. In the first case (*reactive* maintain-goals), it starts taking action when a particular state is no longer maintained, while in the second case (*proactive* maintain-goals) it tries to act to *prevent* the failure of the maintenance condition.

The implementation of proactive maintenance goals, although suitable for formal verification [6], requires predictive reasoning mechanisms, which are not easily representable through an operational formalisation, and in procedural, event-guided agent languages in general [15].

In this work we focus on *reactive* maintain-goals, which are available on most agent platforms. Such goals are *activated* each time their maintenance condition is not satisfied and *suspended* if the condition holds. Proactive maintain goals would theoretically also be modellable in our framework but ask for a predictive evaluation of maintain-conditions.

3.1 An example: the Cleaner Agent

To illustrate how a goal model captures the intended run-time behaviour, we refer to a very simple cleaner robot scenario, which can be found in several variations in artificial intelligence and multi-agent systems fields.

The *Cleaner Agent* is modelled in an extended version of the *Tropos* modelling language and represents the control software for an autonomous robot that could ideally be employed in an office building (Figure 1).

The *achieve-goal* `RoomClean` has an achievement condition "room clean at the end of the day" and is OR-decomposed into the two alternatives `DryCleaning` and `WetCleaning` (both are *perform-goals*). These "leaf-level" goals are operationalised by plans that give different contribution to the quality requirement *efficiency*, modelled as a softgoal.

Supposing both alternative subgoals are applicable in the current context, the agent will pursue the subgoal that maximizes contribution to its softgoal `efficiency`. The semantics of the goal model now allow designers to characterize various run-time behaviours of an agent, like the ones shown in the following scenarios.

Scenario 1. The agent achieved `DryCleaning` by using a broom, but due to some stubborn dirt, the achievement condition of the main goal `RoomClean` is not yet satisfied. Thus, the agent should retry the other available alternative, the goal `WetCleaning`, hoping that after its achievement the main goal will be achieved.

Scenario 2. Suppose that the agent is cleaning the room with a mop, performing the goal `WetCleaning`, and runs out of water. If all the dirty parts of the floor were already cleaned (and the agent can sense this), the achievement condition of `RoomClean` is satisfied and thus, after all, the top goal succeeds.

Interesting considerations arise by modelling these scenarios. The extended goal model allows designers to characterize and focus on the agent's knowledge in terms of goals and their relationships, still maintaining a strong connection between leaf goals and plans as means to achieve them. Modelling different goal types along with specific conditions results in various possible agent behaviours. Such behaviours adhere not only to the semantics of goal AND/OR decomposition but are also driven by the nature of goal types along with their satisfaction conditions.

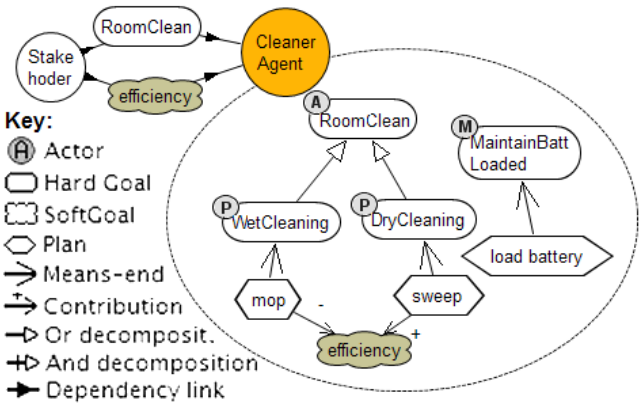

Figure 1: Fragment of a goal model for the Cleaner Agent example.

4. GOAL MODEL SEMANTICS

In this section, we provide operational semantics to deal with non-leaf goals in a GM and we instantiate them for each goal type, illustrating how it naturally adheres both to the semantics of run-time goals (as in agent languages such as JACK, Jadex, and 3APL) and to the interpretation given to hierarchical goal decomposition in GMs of agent-oriented software engineering methodologies like *Tropos* [1] and KAOS [3].

We follow and build on the formalisation used in [15], based on the idea to have an abstract goal architecture that allows for different instances to model the desired run-time behaviour of various types of goals.

Our architecture defines the different states of a goal in the run-time goal satisfaction process and the operational semantics of goal satisfaction for AND- and OR-decompositions in terms of transition rules.

4.1 Abstract semantics for AND/OR decomposition

In the abstract architecture proposed by [15], once adopted, (leaf) goals can have two different states: suspended and active. In the active state planning and execution of plans take place. The satisfaction process for non-leaf goals is more complex, essentially because two facts have to be assessed: the satisfaction of subgoals of AND/OR decompositions and the satisfaction of the conditions defined for a particular goal type. The flexible interplay between these two aspects calls for additional goal states to explicitly represent failure and success of goal achievement.

We define an abstract architecture for non-leaf goals, that includes the following goal states $S = \{$*suspended (S), active-deliberate (AD), active-undefined (AU), active-success (AS), active-failure (AF)*$\}$. In the following we will define rules for the transitions between these states, labelled as *adopt,*

activate, suspend, deliberate, subgoal-achieve, fail, succeed, retry, reactivate, drop-failure, and *drop-success* in Figure 2.

Some of these transitions are governed by *transition actions* (one of ACTIVATE, SUSPEND, FAIL, SUCCEED, RETRY, REACTIVATE, DROPFAILURE, DROPSUCCESS) which are correlated to specific transition conditions. In Section 4.3, these conditions will be instantiated to obtain the desired behaviour for the different goal types, "enabling" the proper actions.

The state of an agent is characterised by a tuple $\langle B, G\rangle$, where B is its actual set of beliefs (the *belief base*), which contains a set of beliefs and known facts about the surrounding world, perceptions, messages and its internal state. G is the set of goals $g_1 \dots g_n$, the agent actually has to pursue, i.e. the *adopted* goals.

A generic non-leaf goal at run-time is defined as $g(C, E, s, \Gamma)$, where $s \in S$ is the actual goal state and Γ is a list of goals that results from a deliberation activity $deliberate(g, B)$, returning applicable subgoals for g: $\Gamma = \{\gamma_1, \dots, \gamma_n\}$. C and E are tuples of the form $\langle condition, action\rangle$, where *action* is one of the transition actions previously defined and *condition* is evaluated in B. A Condition c in E will be evaluated in B when the set of adopted subgoals Γ is empty, a condition in C when $\Gamma \neq \emptyset$. $B \models c$ denotes conditions true with respect to the actual beliefs.

4.2 Transition rules

The operational semantics for our abstract architecture are defined by a set of inference rules that define possible state transitions. Each rule is specified as

$$\frac{L}{R} \qquad \textit{[rule-name]}$$

where R represents a possible state transition of the system under the set of conditions L.

Following [15], we define how single goals evolve, assuming that the goals in G, which the agent has to pursue, are already in the state *Suspended*. New goals can be added to G upon request from outside, triggered by a creation (adoption) condition or as subgoals during goal achievement. Goal adoption is not further discussed here for space reasons[1].

In the following, we define and explain the transition rules for both AND- and OR-decomposed goals.

4.2.1 Goal activation

Goal activation is guided by a condition c. The following two transition rules, *[activateC]* and *[activateE]*, define the state transition from *Suspended* to AD, depending on a condition associated to the action ACTIVATE. The naming includes the transition labels in Figure 2; "E" and "C" denote that the transition is applied for an empty or non-empty set Γ, respectively. Unless otherwise defined, all rules are used for both AND- and OR-decomposition.

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Activate}\rangle \in C \qquad B \models c}{\langle B, g(C, E, Suspended, \Gamma)\rangle \rightarrow \langle B, g(C, E, AD, \Gamma)\rangle} \qquad \textit{[activateC]}$$

$$\frac{\langle c, \textsc{Activate}\rangle \in E \qquad B \models c}{\langle B, g(C, E, Suspended, \emptyset)\rangle \rightarrow \langle B, g(C, E, AD, \emptyset)\rangle} \qquad \textit{[activateE]}$$

4.2.2 Subgoal achievement

The first step in non-leaf goal achievement consists in revealing its subgoals. For this, the function *deliberate* returns a list Γ of subgoals to satisfy, while the goal state changes from AD to AU. In its simplest form, the deliberation function returns the whole set of subgoals, but also complex algorithms for subgoal discovery could be implemented. No deliberation takes place in the case that $\Gamma \neq \emptyset$[2].

$$\frac{}{\langle B, g(C, E, AD, \emptyset)\rangle \rightarrow \langle B, g(C, E, AU, deliberate(g, B))\rangle} \qquad \textit{[deliberateE]}$$

$$\frac{\Gamma \neq \emptyset}{\langle B, g(C, E, AD, \Gamma)\rangle \rightarrow \langle B, g(C, E, AU, \Gamma)\rangle} \qquad \textit{[deliberateC]}$$

At this point, subgoal adoption (and thus, eventually, their achievement) can take place. AND- and OR-decomposed goals have different achievement semantics. Intuitively, the goal remains in the undefined state AU as long as the result of subgoal achievement is uncertain. Thus, an AND-decomposed goal remains in AU until one subgoal fails (rule *[AND:subg-achieve]*), in which case it will change to the 'provisional' failure state AF *[AND:subg-fail]*. When all subgoals are pursued ($\Gamma = \emptyset$) and the goal is still in state AD, applying *[AND:goal-succeed]* it will transit to the 'provisional' success state AS[3].

Referring to OR-decomposition, a goal transits to AS at first success of a subgoal and to AF if all subgoals fail.

We define that each instance of a subgoal γ updates the belief base with $success(\gamma)$ or $failure(\gamma)$, depending if it was achieved or not. Accordingly, our formalisation provides this information to the belief base when a goal is dropped. To ensure that transitions triggered by true conditions have precedence over adopting a new subgoal, the next four transition rules also need the precondition $\neg\exists\langle c, a\rangle \in C \,.\, (B \models c) \wedge a \in \{\textsc{Fail}, \textsc{Succeed}\}$

$$\frac{\gamma_i \in \Gamma \quad \langle B, adopt(G, \gamma_i)\rangle \rightarrow \langle B', G\rangle \qquad B' \models success(\gamma_i)}{\langle B, g(C, E, AU, \Gamma)\rangle \rightarrow \langle B', g(C, E, AU, \Gamma \setminus \{\gamma_i\})\rangle} \qquad \textit{[AND:subg-achieve]}$$

$$\frac{\gamma_i \in \Gamma \quad \langle B, adopt(G, \gamma_i)\rangle \rightarrow \langle B', G\rangle \qquad B' \models failure(\gamma_i)}{\langle B, g(C, E, AU, \Gamma)\rangle \rightarrow \langle B', g(C, E, AF, \Gamma \setminus \{\gamma_i\})\rangle} \qquad \textit{[AND:subg-fail]}$$

$$\frac{\gamma_i \in \Gamma \quad \langle B, adopt(G, \gamma_i)\rangle \rightarrow \langle B', G\rangle \qquad B' \models failure(\gamma_i)}{\langle B, g(C, E, AU, \Gamma)\rangle \rightarrow \langle B', g(C, E, AU, \Gamma \setminus \{\gamma_i\})\rangle} \qquad \textit{[OR:subg-achieve]}$$

[1] In brief, to guarantee that the control of subgoals is left to the parent goals in a goal tree, external creation requests (in an agent architecture typically messages) and creation conditions should be allowed only for root goals.

[2] This particular behaviour would be required for temporal goal suspension, which is not further detailed here.

[3] 'provisional' for the reason that it is not sure if a goal in these states is dropped with failure/success, because this depends also on various achievement and failure conditions and on an eventual process repetition, whose formal semantics are defined later on.

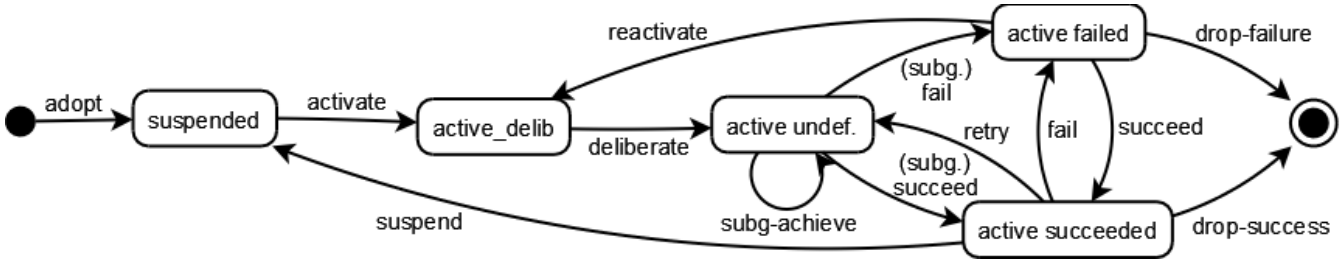

Figure 2: Possible states and transitions in the abstract architecture for non-leaf goals in goal models.

$$\frac{\gamma_i \in \Gamma \quad \langle B, adopt(G,\gamma_i)\rangle \rightarrow \langle B', G\rangle \qquad B' \models success(\gamma_i)}{\langle B, g(C,E,AU,\Gamma)\rangle \rightarrow \langle B', g(C,E,AS,\Gamma \setminus \{\gamma_i\})\rangle} \textit{[OR:subg-succeed]}$$

In these four rules we introduced the function $adopt(G, g)$ to define adoption of a subgoal, that is, adding the (sub)goal g to the goal base G, in order to start its achievement process. Eventually, this will result in a new belief B'. The next transition rule defines how to satisfy the main precondition of the former four rules, the transition from $\langle B, adopt(G,\gamma)\rangle$ to B', that is, adopting the subgoal γ_i in order to start its achievement process, and waiting until γ_i is dropped:

$$\frac{adopt(G,\gamma_i) \rightarrow G \cup \{\gamma_i\} \qquad \langle B, G \cup \{\gamma_i\}\rangle \rightarrow \langle B', G\rangle}{\langle B, adopt(G,\gamma_i)\rangle \rightarrow \langle B', G\rangle}$$

where the function $disp(G, \gamma_i)$ returns $G \cup \{\gamma_i\}$. The new belief B' is the result of the application of transitions for the satisfaction of the goal γ_i, that concludes with some transition rule that drops γ_i from G.

Subgoals that are themselves decomposed to non-leaf goals, will follow the semantics defined in this work. When they are dropped (applying *[DropSuccess]* or *[DropFailure]*, defined later in this section) the agent's belief base is always updated with $success(g)$ or $failed(g)$, where g denotes an unique identifier of a goal instance. In the case that the subgoals are leaf goals, they will be instantiated for example according to Riemsdijk's semantics [15]. We require that also these goals annotate their success or failure in the agent's belief base.

Now we define what happens if a goal is still in the state AD, but its subgoal list Γ is empty. The following rules define that a goal, if it is AND-decomposed and still in AD (thus, no subgoal failed), passes to the provisional success state AS. Conversely, an OR-decomposed goal fails if none of its subgoals succeeded:

$$\frac{\neg\exists\langle c, \textsc{Fail}\rangle \in E.(B \models c)}{\langle B, g(C,E,AU,\emptyset)\rangle \rightarrow \langle B, g(C,E,AS,\emptyset)\rangle} \textit{[AND:subg-succeed]}$$

$$\frac{\neg\exists\langle c, \textsc{Succeed}\rangle \in E.(B \models c)}{\langle B, g(C,E,AU,\emptyset)\rangle \rightarrow \langle B, g(C,E,AF,\emptyset)\rangle} \textit{[OR:subg-fail]}$$

4.2.3 Success and failure triggered by conditions

The following rules define the possibility to transit to the states AS and AF depending on conditions related to the actions Succeed and Fail.

Satisfied success and failure conditions lead from AU to the states AS and AF, respectively. In the case that both conditions are true, failure conditions have precedence.

Moreover, two of these rules also consider transitions from AS to AF and vice-versa, respectively, limited to the case that $\Gamma \neq \emptyset$. The transition $AF \rightarrow AS$ will be triggered only if a subgoal of an AND-decomposed achieve-goal fails, but its achievement condition holds. Conversely, the transition $AS \rightarrow AF$ is used if in an OR-decomposed goal a subgoal succeeds, but the condition associated to the action Fail is true. In these two rules, $\mathbf{X} \in \{AU, AF\}$ and $\mathbf{Y} \in \{AU, AS\}$.

$$\frac{\Gamma \neq \emptyset \quad \neg\exists\langle d, \textsc{Fail}\rangle \in C.(B \models d) \qquad \langle c, \textsc{Succeed}\rangle \in C \qquad B \models c}{\langle B, g(C,E,\mathbf{X},\Gamma)\rangle \rightarrow \langle B, g(C,E,AS,\Gamma)\rangle} \textit{[cond-succeedC]}$$

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Fail}\rangle \in C \qquad B \models c}{\langle B, g(C,E,\mathbf{Y},\Gamma)\rangle \rightarrow \langle B, g(C,E,AF,\Gamma)\rangle} \textit{[cond-failC]}$$

$$\frac{\neg\exists\langle d, \textsc{Fail}\rangle \in C.(B \models d) \qquad \langle c, \textsc{Succeed}\rangle \in E \qquad B \models c}{\langle B, g(C,E,AU,\emptyset)\rangle \rightarrow \langle B, g(C,E,AS,\emptyset)\rangle} \textit{[cond-succeedE]}$$

$$\frac{\langle c, \textsc{Fail}\rangle \in E \qquad B \models c}{\langle B, g(C,E,AU,\emptyset)\rangle \rightarrow \langle B, g(C,E,AF,\emptyset)\rangle} \textit{[cond-failE]}$$

4.2.4 Goal dropping triggered by conditions

The following transition rules define when to drop a goal from the goal base G. When dropping a goal from the state AS, the fact $success(g)$ is added to the agent's belief. Dropping it from AF, $failed(g)$ is added.

$$\frac{\Gamma \neq \emptyset \quad g(C,E,AS,\Gamma) \in G \qquad \langle c, \textsc{DropSuccess}\rangle \in C \qquad B \models c}{\langle B, G\rangle \rightarrow \langle B \cup success(g), G \setminus \{g(C,E,AS,\Gamma)\}\rangle} \textit{[drop-successC]}$$

$$\frac{g(C,E,AS,\emptyset) \in G \qquad \langle c, \textsc{DropSuccess}\rangle \in E \qquad B \models c}{\langle B, G\rangle \rightarrow \langle B \cup success(g), G \setminus \{g(C,E,AS,\emptyset)\}\rangle} \textit{[drop-successE]}$$

$$\frac{\Gamma \neq \emptyset \quad g(C,E,AF,\Gamma) \in G \qquad \langle c, \textsc{DropFailure}\rangle \in C \qquad B \models c}{\langle B, G\rangle \rightarrow \langle B \cup failed(g), G \setminus \{g(C,E,AF,\Gamma)\}\rangle} \textit{[drop-failureC]}$$

$$\frac{g(C,E,AF,\emptyset) \in G \qquad \langle c, \textsc{DropFailure}\rangle \in E \qquad B \models c}{\langle B, G\rangle \rightarrow \langle B \cup failed(g), G \setminus \{g(C,E,AF,\emptyset)\}\rangle} \quad \textit{[drop-failureE]}$$

4.2.5 Reactivation, Suspension, and Retry

Goal achievement might not always be straight forward. To avoid error propagation, often, a failure-avoiding behaviour will be desired. Moreover, some goals will require permanent maintenance of some state of affairs. Thus, we introduce transition rules guided by conditions and used to backtrack in the goal achievement process, either from the success or the failure state.

The following two rules define how to restart the goal achievement process, including subgoal deliberation, when in the state AF. Remaining subgoals in Γ are deleted. Such a transition is needed to repeat goal achievement if subgoal achievement failed and goal failure should be avoided. It is worth noticing that, if more transition rules are applicable at the same time and no rule is more specific than the others, precedence to the application of transition rules is given by the order of definition of the conditions at the instantiation of a goal.

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Reactivate}\rangle \in C \qquad B \models c}{\langle B, g(C,E,AF,\Gamma)\rangle \rightarrow \langle B, g(C,E,AD,\emptyset)\rangle} \quad \textit{[reactivateC]}$$

$$\frac{\langle c, \textsc{Reactivate}\rangle \in E \qquad B \models c}{\langle B, g(C,E,AF,\emptyset)\rangle \rightarrow \langle B, g(C,E,AD,\emptyset)\rangle} \quad \textit{[reactivateE]}$$

Similar rules are needed for goal suspension after a successful goal execution (this is typically needed for maintain-goals). The list of dispatched subgoals is emptied, thus such transitions are not suitable for modelling *context conditions*, which should temporary suspend a goal and subsequently reactivate it, resuming from the previous state.

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Suspend}\rangle \in C \qquad B \models c}{\langle B, g(C,E,AS,\Gamma)\rangle \rightarrow \langle B, g(C,E,Suspended,\emptyset)\rangle} \quad \textit{[suspendC]}$$

$$\frac{\langle c, \textsc{Suspend}\rangle \in E \qquad B \models c}{\langle B, g(C,E,AS,\emptyset)\rangle \rightarrow \langle B, g(C,E,Suspended,\emptyset)\rangle} \quad \textit{[suspendE]}$$

The last rule applies only to goals with a non-empty subgoal list Γ. It backtracks from AS to the undefined state AD, where goal achievement can be retried, trying with the remaining subgoals in Γ. This transition can be applied if an OR-decomposed goal succeeds referring to the achievement of one of its subgoals but the goal's achievement conditions are not satisfied:

$$\frac{\Gamma \neq \emptyset \qquad \langle c, \textsc{Retry}\rangle \in C \qquad B \models c}{\langle B, g(C,E,AS,\Gamma)\rangle \rightarrow \langle B, g(C,E,AU,\Gamma)\rangle} \quad \textit{[retryC]}$$

4.3 Instantiation of the abstract architecture

The abstract architecture for non-leaf goals in GMs, with its different actions and conditions that drive and guide the goal satisfaction process, is now adapted to the behaviour needed for the various types of goals and the interplay of their achievement and failure conditions with the subgoal achievement process. In the following, we instantiate the architecture giving precise semantics for the most significant goal types: perform-goals, achieve-goals, and (reactive) maintain-goals, as introduced in Section 3.

4.3.1 Perform-goals

Perform-goals are available in most agent languages, to execute plans without demanding that they must reach some particular state [4].

In a goal model, we associate the following semantics to a perform-goal: depending on the decomposition type, all (for AND) or at least one (for OR) of the subgoals have to be achieved to achieve the goal. The following instance of our abstract architecture defines a simple perform-goal with no explicit conditions, that fails if the subgoals cannot be achieved at the first try:

$$P \equiv g(E,C), with \;\; E = C = \{\langle true, \textsc{Activate}\rangle, \langle true, \textsc{DropFailure}\rangle, \langle true, \textsc{DropSuccess}\rangle\}$$

Alternative run-time semantics associated to perform goals define that failure has to be avoided and thus goal achievement has to be restarted if the goal enters a failure state (also called *recurrent* or *retry-perform goals* in literature). This can be realised by replacing, in both E and C, the condition $\langle true, \textsc{DropFailure}\rangle$ with $\langle true, \textsc{Reactivate}\rangle$. In this interpretation, also failure conditions can be needed. In this paper, failure conditions will be introduced later for achieve-goals.

4.3.2 Achieve-goals

In general, achieve-goals have a *success condition* (or *achievement condition*) s that has to be satisfied, and usually also a *failure condition* (or *drop condition*) f. Only these two conditions guide the dropping of an achieve-goal from the goal base, regardless of the satisfaction of subgoals. For example, if all subgoals fail, but the goal success condition is satisfied, then the goal is dropped with success. Moreover, we define that the success and failure conditions should be tested not only at the end, but also during subgoal achievement.

The achieve-goal can also have different behaviours to manage failure: if the success condition is still not met after the subgoals are processed, the goal can a) restart the achievement process or b) fail completely.

Moreover, adding the condition $\langle \neg s, \textsc{Retry}\rangle$ to the set C of conditions tested if $\Gamma \neq \emptyset$, we can achieve the failure-preventing behaviour shown in the example in Section 5, that is, for failed OR-decompositions, goal achievement is restarted with the remaining subgoals.

The following instantiation models the behaviour a):

$$A(s,f) \equiv g(E,C), \; with \; E = H \cup \{\langle \neg s \vee f, \textsc{Fail}\rangle\} \quad and \; C = H \cup \{\langle f, \textsc{Fail}\rangle, \langle \neg s, \textsc{Retry}\rangle\}$$

with the following set of conditions H, included in both E and C:

$$H = \{\langle true, \textsc{Activate}\rangle, \langle f, \textsc{DropFailure}\rangle, \langle s, \textsc{Succeed}\rangle, \langle s, \textsc{DropSuccess}\rangle, \langle \neg s, \textsc{Reactivate}\rangle\}$$

Behaviour b) can be obtained from the previous one replacing $\langle \neg s, \text{REACTIVATE} \rangle$ with $\langle \neg s, \text{DROPFAILURE} \rangle$.

4.3.3 Maintain-goals

As discussed in Section 3, in this paper we limit to *reactive* maintain-goals, that are endowed with a *maintenance condition* m and in most languages also with a *drop condition* d to remove the goal from the list of goals to pursue [15].

Intuitively, maintain-goals try to maintain a certain condition true and never end their life-cycle, unless they are explicitly dropped from the set of adopted goals (that is, from the set of goals the agent actively pursues at a certain moment). The transitions correspond to the ones in achieve-goals, but the goal is suspended if m is satisfied and dropped only if d is true:

$$M(m,d) \equiv g(E,C),\ \textit{with}\ E = H \cup \{\langle \neg m \vee d, \text{FAIL} \rangle\}$$
$$\textit{and}\ C = H \cup \{\langle d, \text{FAIL} \rangle, \langle \neg m, \text{RETRY} \rangle\},\ \textit{where}$$
$$H = \{\langle \neg m, \text{ACTIVATE} \rangle, \langle d, \text{DROPFAILURE} \rangle, \langle m, \text{SUCCEED} \rangle,$$
$$\langle m, \text{SUSPEND} \rangle, \langle \neg m, \text{REACTIVATE} \rangle\}$$

Some definitions of maintain-goal include also a *target condition*. Having both a maintain- and a target condition, the goal is activated each time the maintain-condition is violated, while it is suspended only if the target condition is satisfied. These property allows for a behaviour with a hysteresis in goal activation, preventing unwanted continuous switching between activation and suspension. For example, if the room temperature has to be maintained at 20 °C, each time the heating is turned on, it should heat till 22 °C. To obtain this behaviour, the goal architecture has to be instantiated as $M(m,t,d)$, with all occurrences of m in $M(m,d)$ changed to t, except for $\langle \neg m, \text{ACTIVATE} \rangle$, .

4.4 Discussion: goal types in goal models

Endowing goals in GMs with the semantics defined in this section allows designers for modelling a wide range of complex agent's behaviours by combining goal AND/OR-decomposition with different goal types and conditions. However, not all combinations are meaningful, either for modelling or for implementation purposes.

For example, in using the refinement process for abstract goals within a goal model, a maintain-goal can be decomposed either (a) to more specific maintain-goals, or (b) by defining the goals to achieve or perform, in order to maintain the required state. However, at run-time, maintain-goals have the property that they are not dropped when they reach the desired state, but suspended, waiting for reactivation. Thus, child goals of the type *maintain* would never return a positive or negative outcome to their parent goal, unless they are explicitly dropped.

For this reason, to achieve a predictable behaviour, we set as –not necessarily minimal– restrictions to goal models at run-time, that only the leaf-most maintain-goals should be implemented and that decomposition of achieve- and perform-goals to maintain-goals is not allowed.

Moreover, to obtain a comprehensible behaviour, achievement conditions for subgoals (individual subgoal conditions in OR-decomposition, and the union of subgoal conditions in AND-decomposition) should possibly be at least as strong as the parent goal's condition, although this might often not be assured when working with informal languages in a complex environment.

5. APPLICATION OF THE SEMANTICS

In this section we use the simple cleaner agent example introduced in Section 3.1 (Figure 1) to illustrate the application of the proposed operational semantics. We manually execute some steps of this example, and show its run-time behaviour.

We detail the satisfaction process for the goal `RoomClean (RC)`, with the achievement condition that the room has to be clean (supposing the belief base will then contain the predicate *room.clean*). The goal is OR-decomposed into two goals `WetCleaning (WC)` and `DryCleaning (DC)`, both goals of type *perform*, and thus without specific achievement condition. We suppose that the cleaner agent is working in a room, but after a while it encounters some stubborn dirt that cannot be completely removed by the broom (e.g., colour spots after painting the walls). In this example we expect the behaviour outlined in the scenarios in Section 3.1: the cleaner first pursues `DryCleaning`, due to a higher contribution to the softgoal `efficiency`. Sweeping succeeds (plan `sweep`), but the floor is still not clean and so the agent, to avoid failure, also cleans using the mop (plan `mop`).

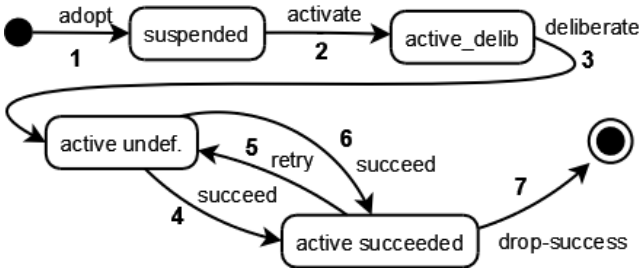

Figure 3: Possible life-cycle of the goal `RoomClean` in the Cleaner Agent example.

We show a step-by step application of the proposed transition rules, instantiating the achieve-goal `RoomClean` with the transition rules for an OR-decomposition. We apply the set of conditions defined in Section 4.3 for achieve-goals of the form $A(s,f)$, with satisfaction condition s=*room.clean* and without failure condition ($f = false$).

After the adoption of `RoomClean`, the condition $\langle true, \text{ACTIVATE} \rangle$ enables the application of the rule *[activateE]* and the goal passes from state S to state AD (**2** in Figure 3). Then, with the rule *[deliberateE]*, the state changes to AU (**3**), and the function *deliberate* returns the subgoals $\Gamma = \{$`DC, WC`$\}$. Now we expect that `DC` is adopted and executed and returns with success. Notice that in this paper we do not cope with the operational semantics for subgoal prioritisation by softgoal contribution [13]. The rule *[OR:subg-succeed]* now applies (**4**), as shown here:

$$\frac{\dfrac{disp(G,DC) \rightarrow G \cup \{DC\} \qquad \langle B, G \cup \{DC\} \rangle \rightarrow \langle B', G \rangle}{\langle B, disp(G,DC) \rangle \rightarrow \langle B', G \rangle} \qquad B' \models success(DC)}{\langle B, g(C,E,AU,\{DC,WC\}) \rangle \rightarrow \langle B', g(C,E,AS,\{WC\}) \rangle}$$

The execution of `DC` can be derived by an application of transition rules that end with a rule which models *success*(`DC`) in B'.

Now, to apply a transition starting from the 'provisional' success state AS, the condition s has to be evaluated. As

just described, there are still some colour spots on the floor and thus s is not satisfied. Therefore, the only true precondition of a transition rule from AS is that of *[retryC]*: $\langle \neg s, \text{RETRY} \rangle$. So the goal state changes back to AU **(5)**. The only goal remaining in Γ, `WC`, will now be pursued.

We suppose that after having cleaned most of the floor, this subgoal fails, because the robot runs out of water. However, the stubborn spots were removed, and thus the condition s is satisfied. The rules *[OR:subg-fail]* and *[cond-succeedE]* are now candidates for the next transition. Since $B \models s$, only the latter can be applied and the goal state changes again to AS **(6)**. Finally, the rule *[drop-successE]* can be applied **(7)**, the goal is dropped and the predicate *success*(`RoomClean`) is added to the agent's beliefs.

With this simple example we can already observe that the agent exhibits a failure-preventing behaviour, by means of reasoning on the structure of its goal model.

6. CONCLUSIONS AND FUTURE WORK

In this paper we proposed and illustrated the interplay between (extended) goal models –conceived as graphs of goal AND/OR decompositions– and run-time goal types along with their achievement conditions. To deliver on this aim, building upon the proposal by Riemsdijk et al. [15], we characterized the behaviour of goals in goal models at run-time, providing their operational semantics.

Goal models allow designers to characterize an agents' behaviour in terms of (less and more concrete) goals and their relationships. Maintaining these goal models at run-time and defining how they guide the run-time behaviour, an agent is able to use the information available in the models as a means for run-time adaptivity and fault tolerance. However, the semantics proposed in this paper currently cover only a subset of goal model concepts and relationships and do not cope with the behaviour resulting from complex reasoning mechanisms, as available for goal adoption, optimisation, conflict resolution, learning or decision making. Also, these semantics are not amenable for formal verification, e.g. by model checking.

Up to our knowledge, currently no agent-oriented language natively supports goal models at run-time. We are working in parallel on a prototype implementation based on the *Jadex* agent framework, adding a new layer of abstraction to support whole goal models at run-time, and completing a tool, called *t2x*, for an automatic mapping from *Tropos* goal models to *Jadex* agent code [11].

Another important direction is to use the proposed semantics to validate a goal-directed design and to test if the run-time behaviour of an agent is compliant with its goal-directed design. Moreover, we investigate on the possibility to give to this software, that knows its goal model, the ability to enrich and modify this model at run-time, to achieve an adaptive behaviour, learning from failures and collaboration. Our vision is to give automated feedback from these models to support the developer in improving the correspondent design time models.

7. REFERENCES

[1] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, July 2004.

[2] C. Cheong and M. Winikoff. Hermes: Designing Goal-Oriented Agent Interactions. In *Proceedings of the 6th Int. Workshop on Agent-Oriented Software Engineering (AOSE-2005)*, 2005.

[3] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. In *6IWSSD: Selected Papers of the Sixth International Workshop on Software Specification and Design*, pages 3–50, Amsterdam, The Netherlands, The Netherlands, 1993. Elsevier Science Publishers B. V.

[4] M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Goal types in agent programming. In *ECAI*, pages 220–224, 2006.

[5] S. Duff, J. Harland, and J. Thangarajah. On proactivity and maintenance goals. In *5th International Conference on autonomous agents and multiagent systems (AAMAS '06)*, pages 1033–1040, New York, NY, USA, 2006. ACM.

[6] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in Tropos. In *IEEE Int. Symposium on Requirements Engineering*, pages 174–181, Toronto (CA), Aug. 2001. IEEE Computer Society.

[7] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[8] M. Ghallab, D. S. Nau, G. Malik, and P. Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004.

[9] V. Lesser. A retrospective view of fa/c distributed problem solving. *Systems, Man and Cybernetics, IEEE Transactions on*, 21(6):1347–1362, 1991.

[10] S. Liaskos, A. Lapouchnian, Y. Yu, E. Yu, and J. Mylopoulos. On goal-based variability acquisition and analysis. In *14th IEEE Int. Conf. on Requirements Engineering*, Minneapolis, 2006.

[11] M. Morandini, L. Penserini, and A. Perini. Automated mapping from goal models to self-adaptive systems. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 485–486, September 2008. Tool demo.

[12] L. Padgham and P. Lambrix. Formalizations of Capabilities for Bdi-Agents. *Autonomous Agents and Multi-Agent Systems*, 10:249–271, 2005.

[13] L. Penserini, A. Perini, A. Susi, and J. Mylopoulos. High variability design for software agents: Extending tropos. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2(4), 2007.

[14] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A bdi reasoning engine. In *Multi-Agent Programming*, pages 149–174. Springer, USA, 9 2005. Book chapter.

[15] B. van Riemsdijk, M. Dastani, and M. Winikoff. Goals in agent systems: A unifying framework. In *Proceedings of the 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 713–720. IFAAMAS, 2008.

[16] M. B. van Riemsdijk, M. Dastani, and J.-J. C. Meyer. Subgoal semantics in agent programming. In *EPIA*, pages 548–559, 2005.

[17] E. Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, University of Toronto, Department of Computer Science, 1995.