# Specifying, validating and generating an agent behaviour using a Goal Decomposition Tree

Gaële Simon, Bruno Mermet, Dominique Fournier, Marianne Flouret

Université of Le Havre

**Abstract.** This paper deals with a goal-oriented agent model called Goal Decomposition Tree (GDT) allowing both to specify and validate the behaviour of an agent. This work takes place in a global framework whose goal is to define a process allowing to start from a problem specification to obtain a validated implementation of a corresponding MAS. The GDT model has been used to specify a prey-predator system which has been verified this way.

## 1 Introduction

Our research deals with methods and models, in order to help multiagent systems designers to manage the complexity of MAS. We do not aim at developing yet another agent model: there are already numerous ones. We aim at helping to develop multiagent systems whose behaviour can be verified. As a consequence, our approach consists of four steps:

1. **an agentification method** that helps to determine the set of agents which must be used to solve a given problem;
2. **an agent design model** to help to design an agent behaviour that can be verified;
3. **a proof system** to prove that the agent model satisfies the main goal of the agent;
4. **an implementation model** that can be automatically generated from the agent design model.

Our aim is to provide a complete MAS design process starting from the problem specification and ending by a MAS implementation. Our agentification method and our implementation model have already been presented in other articles [10, 7]. This paper is focused on the agent model called GDT.

In order to be able to implement and validate an agent, its behaviour must be clearly and formally specified. In our proposal, this specification is based on a Goal Decomposition Tree (GDT) which helps to describe how an agent can manage its goals. It is used

- as a support for agent behaviour validation and proof;
- to guide the implementation of the agent behaviour induced by the tree using an automaton as in our implementation model called SPACE [7].



M. Baldoni, U. Endriss, A. Omicini, and P. Torroni (eds.)
*DALT 2005, Third International Workshop*, Utrecht, July 25th, 2005
http://www.doc.ic.ac.uk/~ue/DALT-2005/

The main contribution of this model is that it can be proven that the specified agent behaviour is correct according to the main goal of this agent. That's why the aim of this model is to provide a declarative description of goals. Several works have already pointed out the advantage to have a declarative description of goals [14], [2], [12]. Many multiagent models or systems are essentially focused on procedural aspects of goals which is important in order to obtain an executable agent. But the declarative aspect of goals is also very important. Indeed, as it is said in [14], "by omitting the declarative aspect of goals, the ability to reason about goals is lost. Without knowing what a goal is trying to achieve, one can not check whether the goal has been achieved, check whether the goal is impossible". In [12], the authors say that declarative goals "provide for the possibility to decouple plan execution and goal achievement". A GDT is a partial answer to these requirements: as it will be shown in next sections, both procedural and declarative aspects of goals management can be described by a GDT.

Another important aspect of a GDT is that it is intended to be used to directly generate the behaviour of the agent to which it is associated. Indeed, as explained in [12], "the biggest role of goals in agents is thus not the ability to reason about them but their motivational power of generating behaviour". Moreover, "A certain plan or behaviour is generated *because* of a goal". It is exactly what a GDT allows to express. Nodes of a GDT correspond to goals the agent has to solve. As in [14], goals are considered as states of the world the agent has to reach.

Inside the tree, a goal is decomposed into subgoals using decomposition operators. The notion of subgoal used here is similar to the one described in [12]: "a goal can be viewed as a subgoal if its achievement brings the agent closer to its top goal". The notion of "closeness" used by van Riemsdjik et al. is specified differently by each decomposition operator. In the same paper, authors distinguish "subgoals as being the parts of which a top goal is composed" and "subgoals as landmarks or states that should be achieved on the road to achieving a top goal". In a GDT, these two kinds of subgoals exist. The fact that a subgoal is of a particular kind is a direct consequence of the decomposition operator which has been used to introduce the subgoal. All works on agent goals management do not use the same notion of subgoal. In [14] or [2], a subgoal is considered as a logical consequence of a goal. So, in these works, subgoals can be considered as necessary conditions for the satisfaction of the parent goal. In our vision, subgoals are on the contrary sufficient conditions for the satisfaction of the parent goal. The method TAEMS [13] does not use goals buts tasks. However subtasks used in TAEMS can be directly compared to subgoals used in our work.

A decomposition operator encapsulates a set of mechanisms corresponding to a typical goals management behaviour ([2], [12], [14]). Each operator is specified by different kinds of semantics:

- a goal decomposition semantics describing how a goal can be decomposed into subgoals with this operator;
- a semantics describing how to deduce the "type" of the parent goal knowing the types of its subgoals;

– a semantics associating an *automata composition pattern* to each operator. These patterns are used incrementally to build the complete automaton describing the agent behaviour;
– a semantics associating a local proof schema. This schema is used to verify the agent behaviour (ie. to prove that its goals management behaviour satisfies its main goal). This semantics is described in [4].

The section 2 defines the notion of goal as it is used in this work and describes the typology of goals which has been used. The section 3 describes the set of operators which can be used to decompose a goal into subgoals inside the GDT. For each operator, the two first semantics described before are given. The section 4 defines more precisely a Goal Decomposition Tree and shows how a GDT can be built using the tools described in the two previous sections. Last but not least, the section 5 presents a synthetic comparison of the proposed model with well-known goal-oriented models.

## 2 Goals and typology of goals

In the context of a Goal Decomposition Tree, every goal is at least defined by a name and a satisfaction condition. The name is used to make the work of the designer easier. According to the type of each goal, additional information are also used to completely specify the goal. Satisfaction conditions are used to specify goals formally with respect to the declarative requirement for goals described in the previous section. A goal is considered to be achieved if its satisfaction condition is logically true. Satisfaction conditions are expressed using a temporal logic formalism which is a subset of TLA [6]. More precisely, primed variables have been used. For example, if $x$ is a variable, the notation $x$ in a satisfaction condition corresponds the value of $x$ before the resolution of the goal to which the condition is associated. On the contrary, the notation $x'$ corresponds to the value of $x$ after the resolution of the goal. Let notice that more complex temporal logic formula are also used to specify decomposition operators described in section 3. These logical formula are then used during the proof process.

These variables are supposed to be attributes maintained by the agent. Thus, specifying goals of an agent helps also to define the set of variables defining the view of the agent on its environment. For example, in [9], the authors describe a case study where two robots are collecting garbage on Mars. One of the two robots, named R1, must move in order to discover pieces of garbage which must be picked up. As a consequence, for R1, to be in a location where there is a piece of garbage corresponds to a goal. The satisfaction condition of this goal can be defined by: $garbage = true$ where $garbage$ is supposed to be an internal variable of the agent which describes its perception of its current location. This variable is supposed to be updated each time the agent moves i.e. each time its coordinates ($x$,$y$) (which are also internal variables of the agent) are modified.

A typology of goals has been defined in order to distinguish more precisely different ways to manage goals decomposition. The type of a goal has consequences on the solving process of this goal, on the design of its corresponding

behaviour automaton and on the proof process of the behaviour implied by its resolution by the agent.

The first criterion to distinguish goals corresponds to the situation of the goal in the tree. This leads naturally to distinguish two first kinds of goals: elementary and intermediate goals.

*Elementary goals:* they correspond to the leaves of the tree that's why they are not decomposed into subgoals. Furthermore, they are not only defined by a name and a satisfaction condition but also by a set of actions. The execution of these actions are supposed to achieve the goal ie to make its satisfaction condition true. Notice that these actions are related to the capabilities of the agent as described in [2]. They correspond to the procedural aspect of goals described in the previous section. As satisfaction conditions, they are based on variables of the agents. The aim of an action is to modify these variables. For example, for the robot R1 described in section 2, moving one step right can be an elementary goal. Its satisfaction condition is: $x' = x + 1 \wedge y' = y$. The corresponding actions are: $x := x + 1; garbage := containsGarbage(x, y)$. It is supposed that *containsGarbage* is a function associated to the environment allowing the agent to know if there is a piece of garbage at its current location.

*Intermediate goals:* They correspond to the nodes of the tree which are not leaf ones. They are specified by a name, a satisfaction condition and also a Local Decomposition Tree (LDT). A LDT contains a root node corresponding to the intermediate goal and a decomposition operator which creates as many branches (and subgoals) as needed by the operator. It describes how the intermediate goal can be decomposed into subgoals, and sometimes in which context, in order to be achieved. The number and the order of subgoals to be achieved in order to solve the parent goal depends on the chosen operator (see next section for more details).

The second criterion used to define goals is related to the goals satisfiability. Using this criterion, two kinds of goals are again distinguished: Necessarily Satisfiable goals (NS) and Not Necessarily Satisfiable goals (NNS).

*Necessarily Satisfiable goals (NS):* This kind of goals ensures that, once all what must be done to solve the goal has been executed, the satisfaction condition of the goal is always true (the goal is achieved).

*Not Necessarily Satisfiable goals (NNS):* this set of goals is complementary to the previous one. It is the more prevalent case. For this kind of goal, one can not be sure that the goal will be achieved after its actions or its decomposition (and the subgoals associated to this decomposition) have been executed or satisfied. This kind takes into account that some actions or some decompositions can only be used in certain execution contexts. If these contexts are not set when the goal is to be solved, these actions or decompositions become useless.

Information about the satisfiability of a goal is essential not only for the agent to manage its goals but also for the automaton design and the proof process. However, this criterion is not involved in the solving process of a goal. Moreover, using this criterion may help to simplify the tree . Indeed, as it will be shown

in the next section, all the operators do not accept all kinds of goals from this criterion point of view.

The third and last criterion used to distinguish goals is related to the evaluation of the satisfaction condition. Using this criterion, two other kinds of goals can be defined: Lazy goals (L) and Not Lazy goals (NL).

*Lazy goals (L):* when a goal is considered to be a lazy one, its satisfaction condition is evaluated before considering its actions (for an elementary goal) or its decomposition (for an intermediate one). If the satisfaction condition is true, the goal is considered to be achieved which implies that the set of actions or the decomposition associated to the goal are not used. If the satisfaction condition is false, the set of actions or the decomposition are executed. As a consequence, primed variables can not be used in the satisfaction condition associated to a lazy goal.

*Not Lazy goals (NL):* this set of goals is complementary to the previous one. For a not lazy goal, the associated set of actions or decomposition is always executed even if the satisfaction condition is already true. This condition is not evaluated at the beginning of the process, unlike lazy goals.

This criterion can be compared to the requirement for goals management described in [14]: "...given a goal to achieve condition $s$ using a set of procedures (or recipes or plans) $P$, if $s$ holds, then $P$ should not be executed". This requirement corresponds to the solving process of a lazy goal whose satisfaction condition is already satisfied at the beginning of the process.

As a conclusion, in the context of a GDT, each goal can be characterised by three criteria which can be combined independently. Figure 1 summarizes the graphical notations introduced for the two last criteria. Each criterion has two possible values which implies that eight effective kinds of goals can be used in the tree.

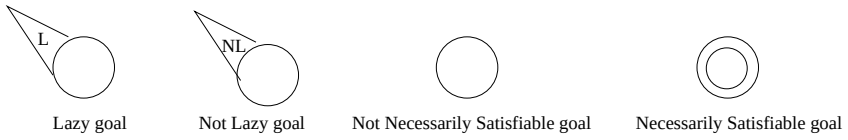


**Fig. 1.** NS, NNS, L and NL goals

Formally, a goal is described by a 6-tuple $< name, sc, el, ns, lazy, LDT\ or\ actions >$ with:

- *name*: string,
- *sc* (satisfaction condition): temporal logic formula,
- *el* (elementary): boolean,
- *ns* (necessarily satisfiable): boolean,
- *lazy*: boolean,
- set of actions (*actions*) or Local Decomposition Tree (*LDT*).

From this definition, the process of goal solving in the GDT can be described by the algorithm given in figure 2. The *solve* function used in this algorithm describes how the tree must be walked during the solving process.

```
boolean solve(G) :
if (G.lazy)
then
       if (G.sc)
       then return(true);
       endif
endif
if (G.el)
then execute(G.actions);
else satisfy(G.LDT);
endif
return(G.sc);
```

**Fig. 2.** Algorithm for solving a goal in a GDT

The *execute* function consists in executing sequentially the set of actions associated to the goal. The *satisfy* function is a recursive one and uses itself the *solve* function which just has been defined. The *satisfy* function is detailed in the next section which also describes each operator which can be used in a GDT.

## 3 Decomposition operators

In this section, available decomposition operators are described. For each operator, the two first semantics are given that is to say the decomposition semantics and the goals types composition semantics. The goals types composition semantics is based only on one criterion defining goals types: their satisfiability mode (NS or NNS). Indeed, the other criteria have not a direct influence on decomposition operators.

Before describing each operator, let precise what means a goal decomposition. Let $A$ be the goal to be decomposed and $Op$ the decomposition operator to be used. As almost all available operators are binary, $Op$ is also supposed to be binary (but it does not modify the semantics of the decomposition). Let $B$ and $C$ be the subgoals introduced by the use of $Op$ to decompose $A$. As described in the previous section, $Op(B, C)$ corresponds to the Local Decomposition Tree associated to $A$. The semantics of this decomposition is that the satisfaction of $Op(B, C)$ (i.e. the LDT) implies the satisfaction of $A$. But what does mean the satisfaction of $Op(B, C)$? It corresponds exactly to the decomposition semantics of each operator. Indeed this semantics describes how many subgoals must be achieved and in which order to be able to satisfy the parent goal. In other words, the *satisfy* function used in the solving algorithm given previously is different for each operator. So in the sequel, for each operator, the *satisfy* function is instantiated. Let us notice that, for all operators, this function uses the *solve* function in order to evaluate the satisfaction of subgoals.

### 3.1 *And* operator

This operator corresponds to the well-known logical operator adapted to a temporal context. Its decomposition semantics states that if a goal $A$ can be decomposed in $And(B, C)$, then $A$ can be satisfied (its satisfaction condition is true) if $B$ and $C$ can be satisfied. This two subgoals can be solved in any order. If at least one of these two goals can not be achievedd, the parent goal $A$ is considered to be not achieved. Figure 3 gives the *satisfy* function corresponding to this behaviour.

```
boolean satisfy(And(B,C)) :
goal chosenGoal, remainingGoal;
chosenGoal,remainingGoal := choose(B,C);
if (solve(chosenGoal))
then return(solve(remainingGoal));
else return false;
endif
```

**Fig. 3.** And satisfaction algorithm

The *choose* operator chooses randomly one of the two subgoals which is returned as first result. The other goal is returned as the second result of the function.

Figure 4 shows the semantics of this operator as far as goals types composition is concerned. This figure shows that the two subgoals can be either necessarily satisfiable either not necessarily satisfiable. According to the *And* operator decomposition semantics, if at least one of the subgoals is not necessarily satisfiable, the parent goal is automatically not necessarily satisfiable. It is necessarily satisfiable otherwise.

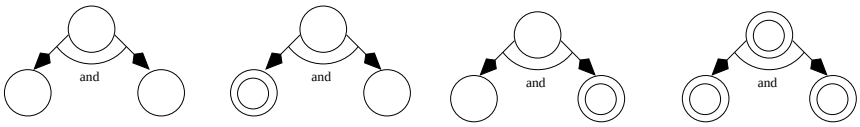


**Fig. 4.** And composition semantics

### 3.2 *Or* operator

This operator corresponds to the standard logical operator (in the same way as AND before).

Figure 5 shows the semantics of this operator as far as goals types composition is concerned. This figure shows that the two subgoals can be either necessarily satisfiable either not necessarily satisfiable. In summary, if at least one of the

subgoals is necessarily satisfiable, the parent goal is automatically necessarily satisfiable. It is not necessarily satisfiable otherwise.

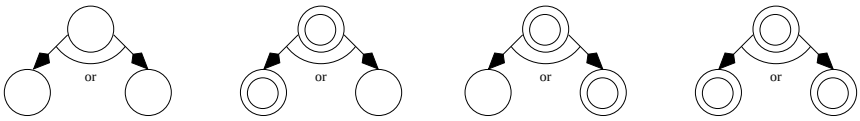


**Fig. 5.** Or composition semantics

### 3.3 *SeqAnd* operator

This operator corresponds to a "sequential And" operator. Indeed, the main difference with the *And* operator is that the two subgoals must be solved in the order specified by the operator. The figure 6 gives the *satisfy* function corresponding to this behaviour. The composition semantics is the same as the *And* operator's one.

```
boolean satisfy(SeqAnd(B,C)) :
if (solve(B))
then return (solve(C));
else return(false);
endif
```

**Fig. 6.** SeqAnd satisfaction algorithm

### 3.4 *SeqOr* operator

The difference between *SeqOr* and *Or* is the same as the one between *SeqAnd* and *And* described in the previous section. The figure 7 gives the *satisfy* function associated to *SeqOr*. Its composition semantics is the same as the *Or* operator's one.

```
boolean satisfy(SeqOr(B,C)) :
if (solve(B))
then return (true);
else return(solve(C));
endif
```

**Fig. 7.** SeqOr satisfaction algorithm

### 3.5 *SyncSeqAnd* operator

This operator is a synchronized version of the *SeqAnd* operator. Unlike *SeqAnd*, this operator ensures that the two subgoals (if they are both solved) are solved without any interruption by another agent. This operator must not be used too much. The agentification method we have proposed [3] is designed to limit cases where this kind of operators must be used by reducing shared variables. Its decomposition semantics, goals types composition semantics and the corresponding satisfaction algorithm are the same as the *SeqAnd* operator's ones.

### 3.6 *SyncSeqOr* operator

The difference between *SyncSeqOr* and *SeqOr* is the same as the one between *SyncSeqAnd* and *SeqAnd* described in the previous section. This operator is a synchronized version of the *SeqOr* operator.

### 3.7 *Case* operator

This operator decomposes a goal into subgoals according to conditions defined by logical expressions. These logical expressions use the same variables as satisfaction conditions. The decomposition semantics of this operator states that the disjunction of the logical expressions corresponding to the two conditions must be true when the parent goal is decomposed. The principle is that if a condition is true, the corresponding subgoal must be solved. The satisfaction of the parent goal depends on the satisfaction of the chosen subgoal. This semantics is summarised by the associated *satisfy* function given in figure 8.

```
boolean satisfy(Case(A,B,cond_a)):
if (cond_a)
then return(solve(A))
else return(solve(B))
```

**Fig. 8.** Case satisfaction algorithm

As far as the composition semantics of the operator is concerned, there are four possible trees as shown in figure 9. If subgoals are both necessarily satisfiable, the parent goal is necessarily satisfiable. If at least one of the subgoals is not necessarily satisfiable, the parent goal is not necessarily satisfiable. It is very important to notice that the property of being "necessarily satisfiable" is a little bit different in the context of the case operator. Indeed, here, a subgoal is necessarily satisfiable only if its associated condition is true. For the other operators, when a goal is declared to be necessarily satisfiable, it is true in any context. This characteristic is particularly useful for the proof process.

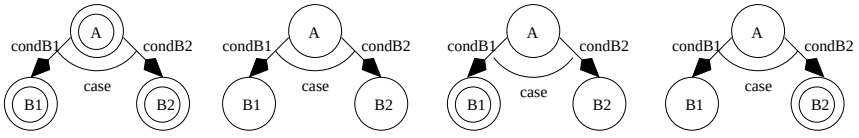


**Fig. 9.** Case operator composition semantics

### 3.8 *Iter* operator

This operator is an unary one. The main difference between this operator and the others is that its behaviour depends on the satisfaction condition of the parent goal. The decomposition semantics of this operator states that the parent goal

will be satisfied after several satisfaction steps of the subgoal. In other words, the satisfaction condition of the subgoal must be true several times in order the satisfaction condition of the parent goal to become true.

This operator is very important because it takes into account a progress notion inside a goal solving process. For example, let suppose that the satisfaction condition of the parent goal $A$ is "to be in $(x, y)$ location". Let suppose that the agent can only move one step at a time. As a consequence, the solving of $A$ must be decomposed into $n$ solving of the subgoal "move one step", $n$ being the number of steps between the current location of the agent and the desired final location.

This operator can only be used when the satisfaction of the subgoal implies a progress in the satisfaction of the parent goal. In other words, each time the subgoal is satisfied, the satisfaction of the parent must be closer. However, sometimes it is possible that the subgoal can not be satisfied (because the context of the agent has changed for example). In this case, the satisfaction degree of the parent goal stays at the same level and the subgoal must be solved again. The important characteristic of this operator is that the satisfaction level of the parent goal can not regress after a satisfaction step of the subgoal, even if this step has failed. If it is the case, it means that the *Iter* operator should not have been used. The proof schema associated to the *Iter* operator helps to verify this property.

The overall behaviour of the operator described in the previous paragraph is summarised by the associated *satisfy* function given in figure 10.

```
boolean satisfy(Iter(sc,B)) :
boolean satisfied;
repeat
       repeat
              satisfied = solve(B);
       until (satisfied or sc)
until (sc)
return true;
```

**Fig. 10.** Iter satisfaction algorithm

The goals types composition semantics of this operator is summarised in figure 11. It shows that the subgoal can be either necessarily satisfiable either not. However, the parent goal is always necessarily satisfiable. Indeed, the behaviour of the operator implies that the solving process of the subgoal stops when the satisfaction condition of the parent goal is true which implies that this one is necessarily satisfiable.

### 3.9 Comparison with other works

Other works on goals management of an agent propose mechanisms to express relations between goals or subgoals. In this paragraph, two of them are detailed

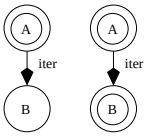


**Fig. 11.** Iter composition semantics

in order to be precisely compared with GDT. In GOAL [2], the authors propose a global logical framework in order to formalise the goal management behaviour of an agent. In this framework, the state of an agent is defined by a mental state $< B, G >$ which consists of the beliefs and goals of the agent. Beliefs and goals are modelled by logical formula: $B(F)$ is a belief and $G(F)$ is a goal, $F$ being a logical formula.

In this framework, a goal cannot be deduced from the set of beliefs. When a goal is achieved, it (and all of its logical consequences) is removed from the goals base. Then it is added to the set of beliefs.

The behaviour of the agent is specified by a set of *conditional actions*. A conditional action is a pair $\phi \rightarrow do(a)$ where $\phi$ is a condition (a logical formula) and $a$ is an *action*. There are three kinds of actions:

- *beliefs management actions*: theses actions allow to manage the set of beliefs:
  - $ins(\phi)$ adds $B(\phi)$ to the set of beliefs,
  - $del(\phi)$ deletes $B(\phi)$ from the set of beliefs;
- *goal management actions*: these actions allow to explicitly manage goals:
  - $adopt(\phi)$ adds $G(\phi)$ to the set of goals,
  - $drop(\phi)$ deletes $G(\phi)$ from the set of goals;
- *basic actions*: these actions are described by a *semantic function* $\mathcal{T}$, a partial function that modifies the set of beliefs of the agent.

For instance, here is how our *SeqAnd* operator could be translated in Goal. Let suppose that $A$ is a goal that is decomposed in *SeqAnd(X,Y)*, That is to say *SeqAnd(X,Y)* $\Rightarrow A$.

The behaviour of the agent corresponding to the resolution of the goal $A$ can then be described by the following conditional actions:

- $G(A) \wedge \neg B(X) \rightarrow do(adopt(X))$ (if $A$ must be solved and $X$ is not yet believed, then $X$ becomes a goal of the agent);
- $G(A) \wedge B(X) \wedge \neg B(Y) \rightarrow do(adopt(Y))$ (if $A$ must be solved and $X$ has already been achieved (and is thus a belief), then $Y$ becomes a goal of the agent);
- $G(A) \wedge B(X) \wedge B(Y) \rightarrow do(ins(A))$ (if $A$ must be solved and $X$ and $Y$ have been achieved, then $A$ is achieved and can be added to the set of beliefs of the agent. It will also be removed from the set of goals of the agent because Goal agents implement the blind commitment strategy).

Of course, it is assumed that there are also rules to solve goals $X$ and $Y$ which are not detailed here. However, with our model, $X$ and $Y$ are removed from the set of goals remaining to solve by the agent after the resolution of $A$. This can not be expressed in GOAL because conditional actions can not be combined, for instance to be sequentialised. More generally, the hierarchical structure of our

model allows a progressive specification of the agent behaviour which is more difficult with Goal. Last but not least, more elements can be proven with our model than with GOAL. For example, relations between goals like "*ITER*" can not be proven with GOAL. Last but not least, our model allows to perform proofs using first order logic which is not the case with GOAL.

Our decomposition operators can also be compared to the Quality Accumulation Functions (QAF) proposed in TAEMS [13]. TAEMS is a modelling language allowing to describe activities of agents operating in environments where responses by specific deadlines may be required. That's why TAEMS represents agent activities in terms of task structures at multiple levels of abstraction, each with a deadline. A task is described by a name, an arrival time, an earliest start time and a deadline. A task structure is a graph where tasks can be decomposed into subtasks using QAFs. A QAF specifies how the quality of a task can be computed using qualities of its substasks. The quality of a task evaluates the level of completion of the task. Let notice that an important difference between QAF and decomposition operators is that QAF are used in a bottom-up process whereas decomposition operators are used in a top-down process. Indeed, subtasks must have been completed (even with a 0 quality) before the QAF can be used to compute the quality of the supertask. On the contrary, a decomposition operator is used to choose and order the subgoals to be solved in order to satisfy the parent goal.

## 4 The GDT design process

A Goal Decomposition Tree (GDT) specifies how each goal can be solved by an agent. More precisely, the root node of the tree is associated to the main goal of the agent, i.e. the one which is assigned to the agent by the used agentification method ([10] [15]). If this goal is achieved by the agent, the agent is considered to be satisfied from the multiagent system point of view. The tree describes how this goal can be decomposed in order to be achieved using a solution which must be the most adapted to the agent context as possible. Notice that the overall tree can be seen as a collection of local plans allowing to solve each goal. A local plan corresponds to a Local Decomposition Tree associated to a subgoal. The main difference with plans used in [1] is that, in a GDT, they are organised hierarchically. A GDT is very close to the tasks graph used in TAEMS [13]. This graph describes relationships between tasks an agent may have to achieve. Nevertheless, tasks are not goals because their satisfaction is not evaluated by a satisfaction condition (or a predicate) but by a quality measure. In TAEMS, a graph, instead of a tree, is needed because relations between goals, different from decomposition ones, can be expressed. For example, one can specify that solving one goal can prevent from solving another one. This kind of relation can not be directly specified in a GDT but they are not really needed. Indeed, unlike the tasks graph proposed by TAEMS, a GDT is not used by the agent to perform planning tasks but to specify a behaviour which can for example be the result of such planning tasks.

The building process of the GDT consists of four steps. In a first step, a tree must be built by the designer, starting from the main goal of the agent

using a top-down process. This first step allows to introduce subgoals with their satisfaction condition, elementary goals with their associated actions and also decomposition operators. The designer must also decide for each goal if it is lazy or not. During this step, the designer must also define invariants associated to the tree. These invariants specify properties of the problem to be solved by the MAS which must always be true during the life of agents of the system. For example, in a prey/predator problem, an invariant specifies that "only one agent can be located in a given cell of the grid". These invariants are used during the proof process.

In order to make the building process of the tree easier, we are currently defining what can be seen as design patterns i.e. rules which can be used to choose the right operator in particular contexts. For example a rule is focused on the problem of interdependency between goals. When this property exists between two goals $A$ and $B$, it means that the satisfaction of $A$ has an influence on the satisfaction of $B$. When detected, this property can help to guide the choice of the decomposition operator. For example, let suppose that a goal $G$ can be satisfied if two subgoals $B$ and $C$ are satisfied. The *And* operator may be used to model this hypothesis. But if another hypothesis indicates that the satisfaction of $B$ can prevent from the satisfaction of $C$, the *And* operator can not be used anymore, but must be replaced by the *SeqAnd* operator.

In the second step of the GDT design process, the designer must decide for each elementary goal if it is necessarily satisfiable or not. In a third step, the type of each intermediate goal, as far as satisfiability is concerned, is computed using the goals types composition semantics of each used decomposition operator. Unlike the first step, this step is a down-top process. During this process, inconsistencies or potential simplifications can be detected. In that case, the first step must be executed again in order to modify the tree. Figure 12 shows such a simplification for the *SeqOr* operator which can be detected during this step. The first tree can be replaced by the second tree because if the first subgoal of a *SeqOr* is a necessarily satisfiable one, the second subgoal will never be solved (see the definition of the decomposition semantics of this operator in section 3.4).

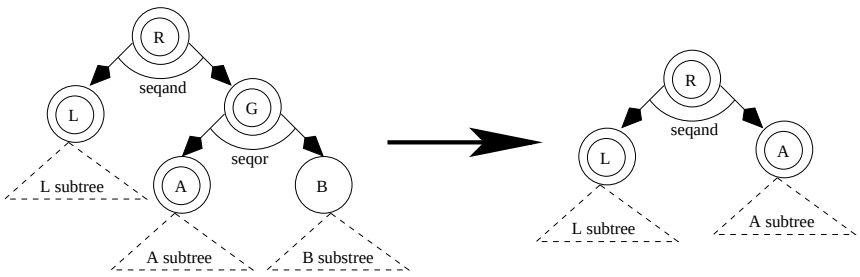


**Fig. 12.** Tree simplification with SeqOr

Once the three first steps have been achieved, a proof of the tree can be built in a fourth step. The process used to achieve this proof is described in [4] . Again, this step can lead to detect inconsistencies in the tree based on proof failures. In a last step, the validated tree is used to build the behaviour automaton of the agent which can then be implemented. This process is also described in [11].

As explained before, the building of the tree leads also to the definition of variables and actions of the agent which are essential parts of an agent model. As a consequence, the GDT and the associated design process can be seen as a design tool for a validated agent model in the context of a MAS design.

## 5 Comparison with other goal-oriented models

| **Model** | **Goal Expression** | **Goal management hypotheses** | **Action kinds** | **Plan language (operators)** |
|---|---|---|---|---|
| **Winikoff** | satisfaction failure | DS, DI, CG, PG, KG | GA, BI, BD, BAN | sequencing, parallelism, conditional selection |
| **AgentSpeak** | none | none | GA, GD, BI, BD, BAN | and, conditional selection (context) |
| **Goal** | satisfaction | DS | GA, GD, BI, BD, BAS | only atomic conditional actions |
| **GDT** | satisfaction | DS | GA, GD, BI, BD derived from the GDT, BAS | many |

**Table 1.** Goal management comparison

The table 1 compares our agent model with a few other ones with a goal oriented point of view: Winikoff et al's model [14], AgentSpeak [9] and GOAL [2]. In the *Goal expression* column, it is specified whether a formal satisfaction condition and a formal failure condition is expressed for each goal in the model. For the models having only a procedural point of view, like AgentSpeak, there is no formal expression of goals. Only the Winikoff's model explicitly gives a formal failure condition, making a distinction between a plan failure and a goal failure.

Among the *Goal management hypotheses*, we distinguish the five characteristics described in [14, 8]. The *Drop Successful* attitude (DS) consists in dropping a goal that is satisfied. The *Drop Impossible* attitude (DI) consists in dropping a goal that becomes impossible to satisfy. Goals are *persistent* (PG) if a goal is dropped only if it has been achieved or if it becomes impossible to solve. The other characteristics correspond to constraints on the goal management process. The *Consistent Goals* property (CG) is satisfied if the set of goals the agent has

to solve must be consistent (if $a$ is a goal, $not(a)$ cannot be a goal). Finally, the *Known Goals* (KG) property specifies that the agent must know the set of all its goals. The model we propose does not need CG, KG and PG constraints to be verified.

In the *Action kinds* column, we precise what kind of actions the language provides. These actions can be divided into 3 types: actions concerning goals management (goal dropping GD, goal adoption GA), actions concerning beliefs management (belief insertion BI, belief deletion BD) and all other actions that we call Basic Actions. These actions may be *Specified* in the language (BAS) or only Named (BAN). BAS are essential to allow a proof process.

Finally, in the last column, we tried to enumerate the *Goal decomposition operators* provided by the language. For the model described in this paper, see section 3. The plan language of Goal is rather a rule language. But for each rule, only one action may be executed: there is, for instance, no sequence operator. In a GDT, plans rely on goal decompositions, and as a consequence, the expressivity of our plan language is also the expressivity of our goal decomposition language.

## 6 Conclusion

In this article, we presented a goal-oriented behaviour model of an agent relying on a Goal Decomposition Tree. The *goal* notion is central in the development of an agent. This appears for instance in the *desires* concept of the BDI model, or is the basis of other methods such as Goal or Moise ([2], [5]). Using a decomposition tree, the user can progressively specify the behaviour of an agent. Thus, goals and plans are closely linked: the decomposition tree of a goal is the plan associated to this goal. A part of goal decomposition operators involves undeterminism, which is necessary for autonomous agents. Of course, using our model, an agent designer must specify each goal by a satisfaction condition. This may seem difficult, but the experience shows that rapidly, unexperimented designers can write the right satisfaction condition. Moreover, this model can be verified using our proof method. The model can then be automatically translated into a behaviour automaton which is, as a consequence, validated also. This automaton can then be implemented inside agents which can be developped using any MAS development platform. However, the design and the proof process are strongly disconnected. So, the designer can develop the GDT without taking care of the proof process. This model has been used to specify prey agents behaviour inside a prey/predator system [11]. The resulting GDT contains sixteen nodes. This system have been also verified using the produced GDT. As shown before, this model can be seen as a tool for agents design. That's why we are going to develop an interpreter which can directly simulate the behaviour of agents from their GDT. The idea is to obtain, as in TAEMS, a method for fast prototyping with validation in parallel.

## References

1. R.H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifiable multi-agent programs. In M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *ProMAS*, 2003.

2. F.S. de Boer, K.V. Hindriks, W. van der Hoek, and J.-J.Ch. Meyer. Agent programming with declarative goals. In *Proceedings of the 7th International Workshop on Intelligent Agents VII. Agent Theories Architectures and Language*, pages 228–243, 2000.
3. M. Flouret, B. Mermet, and G. Simon. Vers une méthodologie de développement de sma adaptés aux problèmes d'optimisation. In *Systèmes multi-agents et systèmes complexes : ingénierie, résolution de problèmes et simulation, JFIADSMA'02*, pages 245–248. Hermes, 2002.
4. D. Fournier, B. Mermet, and G. Simon. A compositional proof system for agent behaviour. In *Proceedings of SASEMAS'2005*, 2005. to appear.
5. J.F. Hubner, J.S. Sichman, and O. Boissier. Specification structurelle, fonctionnelle et deontique d'organisations dans les sma. In *Journees Francophones Intelligence Artificielle et Systemes Multi-Agents (JFIADSM'02)*. Hermes, 2002.
6. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 1994.
7. B. Mermet, G. Simon, D. Fournier, and M. Flouret. SPACE: A method to increase tracability in MAS Development. In *Programming Multi-agent systems*, volume 3067. LNAI, 2004.
8. A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In *Proceeding of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 439–449. San Mateo. CA. Morgan Kaufmann, 1992.
9. A.S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *MAAMAW'96*, volume 1038, Eindhoven, The Netherlands, 1996. LNAI.
10. G. Simon, M. Flouret, and B. Mermet. A methodology to solve optimisation problems with MAS, application to the graph coloring problem. In Donia R. Scott, editor, *Artificial Intelligence : Methodology, Systems, Applications*, volume 2443. LNAI, 2002.
11. G. Simon, B. Mermet, D. Fournier, and M. Flouret. The provable goal decomposition tree : a behaviour model of an agent. Technical report, Laboratoire Informatique du Havre, 2005.
12. M.B. van Riemsdijk, M. Dastani, F. Dignum, and J.-J.Ch. Meyer. Dynamics of declarative goals in agent programming. In *Proceedings of Declarative Agent Languages and Technologies (DALT'04)*, 2004.
13. R. Vincent, B. Horling, and V. Lesser. An agent infrastructure to build and evaluate multi-agent systems: the java agent framework and multi-agent system simulator. In *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, 2001.
14. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, 2003.
15. M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.