Specifying recursive agents with GDTs

Bruno Mermet, Gaële Simon

Specifying Recursive Agents with GDTs

Bruno Mermet · **Gaële Simon**

Abstract The purpose of this article is to formalise the notion of *recursive agent* by extending the Goal Decomposition Tree formalism (GDT). A formal semantics of this decomposition is given, as well as the definition of operators to introduce various ways of recursively defining agents. Design patterns, that show various use cases for recursive agents, are also presented. Finally, to preserve the essential GDT characteristics (that is to allow the verification of agents behaviours), we give proof schemas that allow a proof of the correctness of a recursive agent.

Keywords Goal Decomposition Tree · Agent · Recursive agent · Formal Verification

1 Introduction

For about 6 years, the model of Goal Decomposition Trees (GDT) has been developed to help to formally specify agents and multiagent systems [SMF06, MFS06, MSSZ07, MS09]. This model allows the formal specification of agents by a set of properties and a behaviour. Two main kinds of properties are distinguished: invariant properties and leads-to properties. The behaviour is specified by a goal decomposition tree: an agent has a main goal that can be decomposed thanks to operators into subgoals. Each goal can be either decomposed into subgoals or linked to an action that should achieve it. To verify that properties of an agent are established by its behaviour and that it achieves its main goal, the model gives rules called Proof Schemas to obtain the Proof Obligations (formulae in first order logic) whose correctness must

Corresponding author: Bruno Mermet. Phone: 33 2 32 74 43 21. Fax: 33 2 31 74 43 14.

GREYC, CNRS, Université de Caen Basse-Normandie, ENSICAEN.
Boulevard du Maréchal Juin, F-14032 Caen Cedex, France.
`bruno.mermet@univ-lehavre.fr`, `gsimon@iut.univ-lehavre.fr`,

be proven. This model has significant advantages that make it a good choice for specifying multi-agent systems:

- the verification process is fully specified thanks to *proof schemas*;
- the verification of the correctness of the different parts of an agent is compositional. The main consequence is that partial proofs are possible: a local modification of a specification does not require to verify the whole behaviour of the agent again;
- the verification process of a system made of many agents is also compositional and so, proving the correctness of a multiagent system relies essentially on the proofs of the agents, and only few proofs have to be performed at the system level;
- the program implementing the MAS derived from the specifications can be produced automatically thanks to automata.

However, there are few limitations to this model:

- communications between agents are not yet modeled;
- specifying an agent with several achievement goals is not easy, although it is possible;
- specifying agents performing concurrent tasks is not possible.

This article demonstrates how recursive agents can be used to solve these limitations. After a brief reminder of the GDT model, we describe formally how the model can be extended to define recursive agents (informally, a recursive agent is an agent whose behaviour is partly achieved by "sub-agents"). Several design patterns allowing the implementation of standard behaviours with recursive agents are then presented in section 4. Modeling communications and integrating them with those design patterns is also discussed in section 5. The consequences of using recursive agents on the proof obligations are detailed in section 6. Finally, a comparison with other works is performed before concluding.

2 The GDT4MAS model

The GDT4MAS [MS09] model has been developed to specify and to verify the behaviour of multi-agent systems. This model is an extension of the GDT model that was introduced in [MFS06] and that has already been extended in [MSSZ07]. It has been formally defined using the Linear Temporal Logic (LTL) [CM88]. As a consequence, in the sequel, the following standard operators of the temporal logic are used: always ($\Box$), eventually ($\Diamond$) and next ($\circ$).

2.1 Environment

In the GDT4MAS model, an agent is situated in an environment defined as follows:

Definition 1 (Environment) An *environment* is a triple $\mathcal{E} = (V_\mathcal{E}, I_\mathcal{E}, s_\mathcal{E})$, where:

- $V_\mathcal{E}$ is the set of environment variables,
- $I_\mathcal{E}$ is the invariant of the environment,
- $s_\mathcal{E}$ is the set of *stable properties* of the environment.

Please note that:

- environment variables can *a priori* be modified by any agent of the system. Their value can also change without any action of an agent;
- the invariant of the environment is a predicate that is always true. It specifies the set of valid states of the system;
- a stable property S is a predicate that may be false at the begining of the system. However, if S becomes true, it will remains true.

Example 1 (The RoM problem) The RoM (Robots on Mars) problem has been specified in [BFVW03]. Two agents collaborate to clean the Mars Planet represented as a grid. Robot $R2$ can burn pieces of garbage that are on its cell, but it is unable to move. Robot $R1$ can pick up and drop waste, and it can move (the complete solution using GDTs can be found in [MSSZ07]).

In this example, the invariant property specifies the size of the grid and the fact that each cell is either clean or dirty. It can be written: $G \in x_{min}..x_{max} \times y_{min}..y_{max} \rightarrow \{clean, dirty\}$

The fact that a cell different from $R2$'s cell is empty is an example of stable property, as no agent can drop a piece of garbage (except $R1$ on $R2$'s cell).

2.2 Agent Type and Agent

An agent is specified by instantiating an *agent type*. An agent type is specified as follows:

Definition 2 (Agent type) Let $\mathcal{E}$ be an environment. An agent type T is a tuple:

$$(\mathtt{name_t}, Vt_i, Vt_s, Vt_\mathcal{E}, init_t, I_t, L_t, S_t, Actions_t, \mathtt{Beh}_t))$$

Where:

- $\mathtt{name_t}$ is the name of the type;
- Vt_i is the set of internal variables; the internal variables of an agent ag are variables that the other agents cannot see and that may only be modified by ag;
- Vt_s is the set of surface variables of this type and we have $Vt_s \subseteq Vt_i$. Surface variables are variables that other agents can see but cannot modify;
- $Vt_\mathcal{E}$ is the set of environment variables seen; this is a subset of the variables of the environment in which the agent is situated;
- $init_t$ is the list of initial values of the internal variables;
- I_t is the invariant property; this predicate only deals with internal variables;

- L_t is the set of leads-to properties relying on environment and surface variables; informally, $a\, \texttt{leads}-\texttt{to}\, b$ means that if a becomes true, b will eventually be true;
- S_t is the set of the stable properties of the agent;
- $Actions_t$ is the set of capabilities;
- $\texttt{Beh}_t$ describes the behaviour of the agent using a parameterised GDT, or pGDT (see section 2.3).

Example 2 (Agent type of robot $R2$) Robot $R2$ is defined by its position that is determined by two surface variables, (x_{R2}, y_{R2}), its status (busy or not, specified by a boolean variable *busy*), a leads-to property that specifies that it always cleans the cell on which it stays, a set of actions $Actions_2$ (not detailed here) and a behaviour $GDT(x, y)$ not detailed yet. So we have: $T_{R2}(x, y) =$
$$\begin{pmatrix} R2, \{busy, x_{R2}, y_{R2}\}, \{x_{R2}, y_{R2}\}, \{G\}, \{false, x, y\}, \\ busy \in \mathbb{B} \wedge (x_{R2}, y_{R2}) \in \mathbb{N}^2, \\ \{\Box(G(x_{R2}, y_{R2}) = dirty \rightarrow \diamond G(x_{R2}, y_{R2}) = clean)\}, \\ \{\}, Actions_{R2}, GDT_{R2} \end{pmatrix}$$

Example 3 (Invariant) The invariant I_t of an agent a is a predicate that must be verified in every state of a. It states the type of the internal variables of a, but it can also be used to specify other safety properties, as, for instance:

$$battery\_level < 10 \rightarrow wifi = off$$

Given this definition, an agent can be defined as follows:

Definition 3 (Agent) An agent a is defined by a tuple $(name, type, param)$ where:

- *name* is the name of the agent;
- *type* is the type of the agent;
- *param* is a list of values for the parameters of the GDT.

If $a = (name_a, type_a, param_a)$ is an agent, we will write $name(a)$ to denote its name $(name_a)$, $type(a)$ its type and $param(a)$ its list of parameters.

In this definition, the parameters allow to specialise both the initialisation and the behaviour.

Example 4 (Specification of two robots of type $R2$) Suppose there are two robots of type $R2$. The first one, called $R2_a$, is situated in the cell (x_a, y_a). The second one, called $R2_b$, is situated in the cell (x_b, y_b). These robots will be specified by the following two tuples $(R2_a, T2, (x_a, y_a))$ and $((R2_b, T2, (x_b, y_b)))$.

The complete description of the system can be found in [MS09].

2.3 Behaviour

The behaviour of an agent type is specified by a parameterised Goal Decomposition Tree (or parameterised GDT). A GDT is essentially a tree of goals, in which each goal is decomposed into simpler subgoals, combined via a decomposition operator. The definition of a GDT is given hereafter. A parameterised GDT is obviously a GDT with parameters (see [MS09]).

Definition 4 (GDT) A GDT of an agent type AT is a triple $(pre, tc, root)$ where:

- pre is the *precondition* of the GDT; it must be established by $init_{AT}$ and if the behaviour of the agent must be executed several times, it must be preserved.
- tc is the *triggering context* of the GDT. An agent executes its behaviour each time its triggering context is true (and it is not already executing its behaviour). This triggering context can be set to the boolean constant *true* if the agent must execute its behaviour for all time.
- $root$ is the root goal of the GDT.

Each node (or goal) in a GDT has a name and is formally specified by a *Satisfaction Condition* (SC) and a *Guaranteed Property in case of Failure* (GPF). An SC (resp. GPF) is a temporal formula linking the states before the beginning of the goal resolution process and after it when this process succeeds (resp. fails). If G is a goal, we write SC_G (resp. GPF_G) its SC (resp. its GPF). Notice that, as described in [MSSZ07], GPFs can be inferred from leaf goals. In SCs and GPFs, variables known by the agents (internal or environment variables) can be either primed or unprimed. Unprimed occurences refer to the values of the variables in the state in which the goal resolution process begins whereas primed occurences correspond to the values of the variables in the state in which the goal resolution process ends. Thus, *progress goals* can be defined.

Definition 5 (Progress goal) A *Progress goal* is a goal whose satisfaction condition establishes a relation between the values of the variables before the goal resolution process and the values of the variables after the goal resolution process.

Example 5 (Progress goal) The goal that consists in increasing the value of a variable x is a progress goal that can be specified by the satisfaction condition $x' > x$. In the same way, a goal that specifies that after its execution, the value of a variable y must be equal to the value of the variable x before the resolution process is a progress goal that is specified by the satisfaction condition $y' = x$.

However, goals are not always Progress goals.

Definition 6 (State Reaching goal) A *State reaching goal* is a goal whose SC does not establish a link between the value of variables before the resolution process and after. In the satisfaction conditions of such goals, variables are never primed.

Example 6 (State Reaching goal) The goal *filling a tank* can be specified by the following satisfaction condition $tank = tank_{max}$ where $tank_{max}$ is a constant.

In addition, a goal is typed with two flags:

- the laziness flag: a goal G is said to be *lazy (L)* if its resolution process is executed only if the SC of G is false when G is to be executed. On the contrary, a goal is said to be *non-lazy* (NL) if its resolution process is executed as soon as the goal is to be executed.
- the necessary satisfiability flag: a goal is said to be *necessarily satisfiable* (NS) if each execution of the goal resolution process ensures the establishement of the SC of the goal. Otherwise, the goal is said to be *non-necessarily satisfiable* (NNS). Notice that this flag can be inferred from leaf goals using the process given in [SMF06]. When an NNS goal fails, its GPF is however established. The GPF of NS goals is assumed to be $false$.

Example 7 (Lazy and Non-Lazy goals) A progress goal cannot be a lazy goal. For instance, a progress goal with a satisfaction condition $x' = x + 1$ that requires a change of the value of x during the goal resolution process implies that a goal resolution process runs. A state reaching goal is a goal that specifies a desired state, such as a goal g with the satisfaction condition $x = 10$. In most cases, such a goal will be labelled *Lazy* (L) because, when this goal has to be achieved, is the value of x is already 10, the agent is satisfied and nothing has to be done. However, in some cases, the resolution process associated to g will also perform other tasks that are necessary for the overall behaviour of the agent. In that case, g will be labelled NL.

Example 8 (NS and NNS goals) Consider a robot with an arm and a projector. The robot can always switch its projector on. As a consequence, the goal consisting in switching the projector on is labelled NS (necessarily satisfiable). The arm can also take objects. But this action does not always succeed (the object may be too much round for instance). So the goal consisting in catching an object should be labelled NNS. However, even if the robot fails to catch the object, after the goal, its clamp will be closed. So, the GPF of the goal could specify this.

Finally, the resolution process of a goal can be either an action (for a leaf goal) or a combination of one or several subgoals using a decomposition operator (for non-leaf goals).

2.3.1 Internal goal

Definition 7 (Internal goal) Let A be an agent in an environment $\mathcal{E}$. An *Internal goal* G of a GDT is a 7-tuple:

$$(name, Op, Children, sc, gpf, lz, nsat)$$

where:

- *name* is the name of the goal;
- *Op* is a decomposition operator;
- *Children* is a sequence of goals whose length
- *sc* is the Satisfaction Condition of the goal;
- *gpf* is the Guaranteed Property in case of Failure of the goal;
- *lz* is the laziness flag of the goal;
- *nsat* is the necessary satsfiability flag of the goal; matches the arity of *Op*.

In our model, several decomposition operators have been defined using temporal logic. The definitions given hereafter are only informal ones (see [SMF06] for more details):

- *SeqAnd* is an operator that allows the decomposition of a goal sequentially into two subgoals. For instance, to achieve the goal $x' = (x+1)^2$, we can choose to first achieve the goal $x' = x+1$ and then to achieve the goal $x' = x^2$.
- *SeqOr* is an operator that allows the decomposition of a goal G into two subgoals G_1 and G_2 with the following informal semantics: if the resolution process of G_1 succeeds, then G is achieved. Otherwise, the agent has to attempt to achieve G_2.
- *And* is a non-deterministic operator that allows to decompose a goal into two subgoals that can be achieved in any order to achieve the parent goal. For instance, to achieve the goal $x' = x+3$, we either try to achieve $x' = x+1$ and then $x' = x+2$ or try to achieve $x' = x+2$ and then $x' = x+1$.
- *Or* is a non-deterministic *or* operator.
- *SyncSeqAnd*(V) and *SyncSeqOr*(V) are operators that can lock environment variables V to prevent from any concurrent access.
- $Case(cond_1, cond_2)$ decomposes a goal into two subgoals with conditions for each subgoal.
- *Iter* decomposes a goal G into a subgoal S. Iterating on the resolution of S will eventually achieve goal G. For instance, we could have $SC_G \equiv (x' = x - 10)$ and $SC_S \equiv (x' = x - 1)$.

Example 9 (Internal goal and decomposition operator) The main goal G_1 of robot $R2$ is to clean its cell and to be not busy. This can be specified by the following SC : $SC_{G_1} \equiv \neg busy \wedge G(x_{R2}, y_{R2}) = clean$. This goal can be achieved by first picking up the waste on the cell (x_{R2}, y_{R2}) (goal G_2, the robot is then busy) and then by burning the piece of garbage (goal G_3, the robot is not busy anymore). So we specify that G_1 is decomposed into $G_2\, SeqAnd\, G_3$ with $SC_{G_2} \equiv busy \wedge G(x_{R2}, y_{R2}) = clean$ and with $SC_{G_3} \equiv \neg busy \wedge G(x_{R2}, y_{R2}) = clean$.

2.3.2 Leaf goal and action

An action specifies a capability of an agent and is formally defined as follows:

Definition 8 (Action) An *action a* is a 5-tuple $(name_a, pre_a, post_a, gpf_a, nsat_a)$, where:

- $name_a$ is the name of the action,
- pre_a is the precondition of the action (a predicate that must be true in the state in which the action must be executed),
- $post_a$ is the postcondition of the action (the property that is established if the action succeeds),
- gpf_a is the guaranteed property in case of failure of the action (the property that is established if the action fails),
- $nsat_a$ is the necessary satisfiability of the action; it is true if the action always succeeds, false otherwise.

A leaf goal is a node that is not decomposed into subgoals but that has an action linked to it:

Definition 9 (Leaf goal) A *leaf goal* G is a 6-tuple:

$$(name, a, sc, gpf, lz, nsat)$$

where:

- $name$ is the name of the goal,
- sc, gpf, lz and $nsat$ play the same role as for internal goals,
- a is the name of the action that will be executed to achieve the goal.

Example 10 (Leaf goal and Action) The goal $G2$ of robot $R2$ presented in example 9 is a leaf goal. The action linked to it is the action specified by the following tuple: $(pick\_action, \neg busy \wedge G(x_{R2}, y_{R2}) = dirty, busy \wedge G(x_{R2}, y_{R2}) = clean, false, true)$. Indeed, to pick a piece of garbage, the robot arm must be empty and a piece of garbage must be on the cell of the robot. Once the action has been executed, the robot is busy but the cell is clean. We assume that this action always succeeds.

Definition 10 (Goal execution) In the rest of the paper, the expression "executing a goal" is often used. This means "executing its resolution process", that is to say, for an internal goal, executing its subgoals as specified by the operator, and, for a leaf goal, executing the action associated with this goal.

2.4 Formal semantics

The formal semantics of the GD4MAS model has been given in [MSZ08a, MS09] using temporal logic. In particular, each operator is defined by a set of temporal formulae (slightly different depending on whether the goal is L or NL). Some of these formulae are common to all operators, the others are specific. To define these formulae, new temporal variables have been specified:

- $init_g$ is a variable that is true in every state where goal g begins to be executed. It is false in the other states;

- end_g is true in every state where the execution of g ends;
- in_g is true during the execution of the goal g (between $init_g$ and end_g and false otherwise). So, in_g verifies the following formulae:
  - $\Box(init_g \rightarrow in_g)$: when the execution of g begins, in_g is true,
  - $\Box(in_g \wedge \neg end_g \rightarrow \circ in_g)$: if in_g is true and g does not end, then in_g remains true,
  - $\Box(\neg in_g \wedge \circ\neg init_g \rightarrow \circ\neg in_g)$: in_g cannot become true when the execution of g does not start,
  - $\Box(end_g \wedge \circ\neg init_g \rightarrow \circ\neg in_g)$: if the execution of g ends, then in_g becomes false (except if a new execution of g starts immediately)
- sat_g is true when the execution of the goal g ends successfully (the goal is achieved).

Example 11 (SeqAnd formal semantics) The formal semantics of the *SeqAnd* operator is defined by the following formulae ($g1$ and $g2$ are the first and second subgoals of g):

- general formulae:
  - $\Box(\neg in_{g1} \vee \neg in_{g2})$: both subgoals are not executed simultaneously;
  - $\Box(in_{g1} \vee in_{g2} \rightarrow in_g)$: executing a subgoal means executing the parent goal;
  - $\Box(\neg init_g \vee \neg end_g)$: trying to achieve a goal requires at least one state transition;
  - $\Box((in_g \wedge \neg end_g) \rightarrow \neg init_g)$: while the execution of g is in progress, no other execution can begin;
- formulae specific to the *SeqAnd* operator:
  - $\Box(sat_{g1} \rightarrow \circ init_{g2})$: the execution of $g2$ begins just after $g1$ has been achieved;
  - $\Box((end_{g1} \wedge \neg sat_{g1}) \rightarrow \circ end_g)$: if the execution of $g1$ ends with a failure, then the execution of goal g ends;
  - $\Box(sat_{g2} \rightarrow (end_g \wedge sat_g))$: if goal $g2$ is achieved, then g is also achieved;
  - $\Box(end_{g2} \wedge \neg sat_{g2} \rightarrow \circ end_g)$: if $g2$ is not achieved, the execution of g also ends;
  - $\Box(init_g \rightarrow init_{g1})$: starting to execute g consists in starting to execute $g1$.

2.5 Proof

In the context of the GDT4MAS model, several kinds of proof must be performed:

- the preservation of the invariant of the system by each agent;
- the preservation of the invariant of each agent by itself;
- the establishment of the satisfaction condition of the leaf goals if the action linked to them succeeds;
- the establishment of the precondition of an action each time it is used;
- the establishment of the leads-to properties of an agent by its GDT;

– the correctness of each goal decomposition.

The proofs to perform are called *proof obligations* (PO). They are generated using *proof schemas* (PS). Details about proof schemas and their validity can be found in [MSZ08a]. In order to have compositional proofs, a *Proof Context* (PC) is associated with each node. It is used as a local hypothesis for each local proof. The proof context of each node is inferred from the GDT thanks to rules that are given in [MSSZ07] and are validated in [MSZ08a, MSZ08b]. Notice that the proof context of a node only depends on its parent node or its sibling node. Here, a simplified proof schema for the *SeqAnd* operator is given, as well as the sketch of the proof schema of the *Iter* operator.

2.5.1 Verifying a SeqAnd decomposition

Let A be a goal decomposed into $B\, SeqAnd\ C$. The notation $\phi[a/b]$ is used to represent the formula obtained by substituting within ϕ any free occurence of b by a. Using V in the following formula means that this substitution is performed for any free variable occuring in the formula. The simplified proof schema associated with such a decomposition is the following[1]:

$$(SC_B)[V^{tmp}/V'] \wedge (SC_C)[V^{tmp}/V] \rightarrow SC_A$$

This proof schema is called *simplified* because it does not take into account:

– the hypotheses that can be assumed (context, invariants, etc.)
– the fact that environment variables may have been modified between the execution of the resolution process of B and the execution of the resolution process of C
– the fact that the satisfaction conditions of state-reaching goals are specified without using primed variables

Notice that the effective proof schema that can be found in [MSZ08a] takes this into account.

Example 12 (Proof Obligation generation for a SeqAnd) Consider the case where $SC_A \equiv x' = (x+1)^2$, $SC_B \equiv x' = x+1$ and $SC_C \equiv x' = x^2$. Applying the simplified proof schema of the *SeqAnd* operator gives the following proof obligation:

$$x^{tmp} = x + 1 \wedge x' = {x^{tmp}}^2 \rightarrow x' = (x+1)^2$$

Which is obvious (proof is left to the reader).

[1] In this proof schema, V^{tmp} is a set of fresh variables containing a variable v^{tmp} for each variable v of V.

2.5.2 Verifying an Iter decomposition

We do not give here the detailed proof schema of the *Iter* operator, but only its principle. When a goal G is decomposed into a subgoal S thanks to the *Iter* operator, we have to prove that goal G will eventually be achieved. To do so, a *variant* is used. A variant is a decreasing sequence defined on a well-founded structure (a well-founded structure is a pair $(set, \leq)$ on which each decreasing sequence is bounded). So proof obligations are generated to establish that:

– each time S is executed (successfully or not), the variant decreases;
– when the variant reaches its lower bound, the goal G is achieved.

This proof process is a standard proof process for loop termination. A consequence is that each time an *Iter* operator is used, we prove that the number of iterations that may be performed is finite.

2.6 External goals

In order to specify collaborating agents, external goals have been defined.

Definition 11 (External goal) An *external goal* E of a GDT T of an agent A is a leaf goal of T specified by a tuple $(name, sc, lz)$. Such a goal is a goal that cannot be achieved by A. However, it is required to prove that another agent in the system will achieve such a goal when needed. When E is to be executed by A, A waits until E is achieved. An external goal is obviously an NS goal.

Example 13 (External goal) In the RoM problem, $R1$ must wait for R2's cell to be empty before dropping the waste it carries on it. This goal is not achieved by $R1$ but by $R2$. So, it is an external goal.

The proof process associated with external goals is described in [MS09].

2.7 Graphical representation

Internal goals and leaf goals are represented by ellipses[2]. When a goal is decomposed into two subgoals, the parent goal is linked to its children with two edges and the name of the decomposition operator is written between both edges. If a goal is decomposed into one subgoal (this is the case with the *Iter* operator), the name of the operator is written near the edge. An NS goal is represented by two concentric ellipses. The name of the action linked to a leaf node is written under the representation of the node.

An external goal is represented by a rectangle.

A Lazy goal is indicated by a kind of tab with an “L” label. Non-Lazy goals are optionally indicated by the same tab with an “NL” label.

[2] This ellipse may be a circle.

In most cases, the figure representing a goal (either an ellipse or a rectangle) is labeled with the name of the goal. However, it can also be directly labeled with its satisfaction condition.

Example 14 (Graphical representation of a GDT) In the case of the RoM problem, the GDT of robot $R1$ is shown in figure 1. Details about this GDT can be found in [MSSZ07].

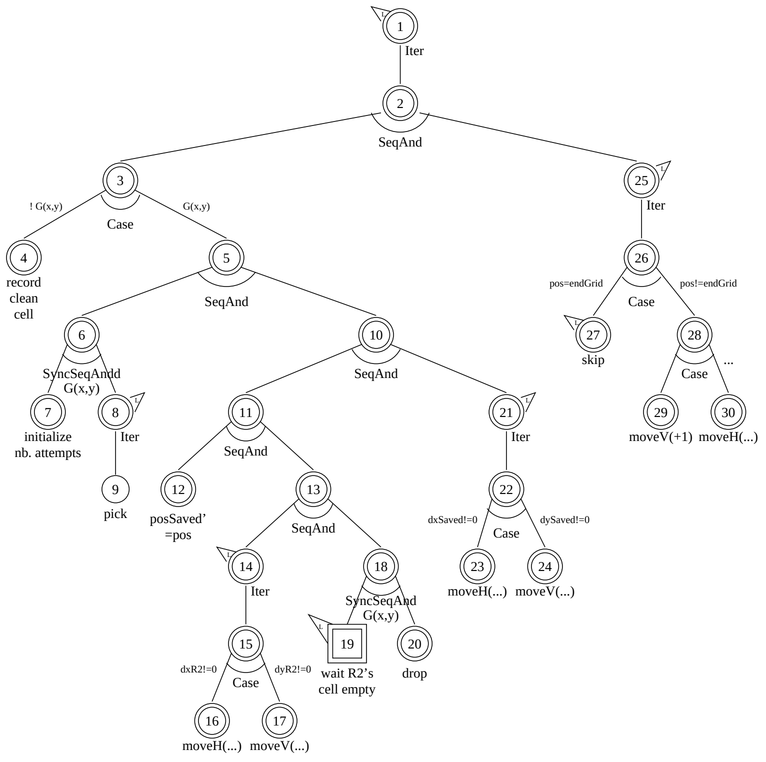

Fig. 1 example of a GDT

2.8 Related work

The GDT4MAS model is connected to several different kinds of works. A detailed comparison with these works can be found in [SMF06] but is summarised hereafter. First of all, there are links with formal agent models like MetateM [Fis94], Desire [BvET97] or Gaïa [WJK00]. These works are focused on agent models on which it is possible to reason. There are also links between our proposal and agent programming languages like AgentSpeak [Rao96], 3APL [DdBDM03], ConGolog [GLL00]. These languages allow to specify agents behaviours which can be directly executed, which is one of the goals of the GDT model. However, 3APL does not allow to prove the specified behaviour and ConGolog is dedicated to situation calculus. Our approach can also be compared to goal-oriented MAS development methods like Prometheus [KW05], MaSe [CHSS01], KAOS or Tropos. Indeed, our proposal is also intended to provide a complete MAS design process from the specification to the implementation. Moreover our approach takes place in the framework of "formal transformation systems" as defined in [CHSS01]. Other works use the term of *subgoal*, but with different meanings. For instance, in [dBHvdHM07], a *subgoal* S of a goal G is a logical consequence of G, whereas in [TPW03], a subgoal S of a goal G is a goal whose achievement is necessary to achieve G and as a consequence, corresponds to our subgoals when the *And* operator is used. Last but not least, our proposal can be directly compared to SMA verification methods. Two subtypes can be distinguished: theorem proving based (like in PROSOCS) and model checking.

PROSOCS [BED$^+$06] agents are agents whose behaviour is described by goal decomposition rules *à la* Prolog. Rules are parameterised by time variables allowing to perform proofs about the evolution of the system state. Many characteristics of PROSOCS agents are very interesting for performing proofs, and a proof procedure has been implemented in Prolog. However, the system is limited to propositional logic formulas. The Goal method [dBHvdHM07] also has a proof model, but the notion of Progf Obligation is missing.

Model checking is a verification method consisting in testing all the situations which may be encountered by the system. Two kinds of model checking techniques can be distinguished: bounded model checking [BFVW03] and unbounded model checking [ALW04,RL04,KLP04]. However, with the two techniques, proofs can only be performed on finite models or on models that can be considered as finite ones.

3 The recursive extension of the GDT4MAS model

As stated in the introduction, the GDT4MAS model lacks expressiveness. However, the gaps detected (several achievement goals, concurrent tasks, message reception management) can all be filled by introducing parallelism into the agents. Indeed, managing concurrent task and several achievement goals can be obviously achieved by introducing parallelism. But managing message reception can also be performed so: with such a system, a task can be dedicated to the management of the message reception, letting the main process run normally, and being able to react as soon as possible to message reception.

Using an agent for each task is a way to specify parallel tasks. As a consequence, as our model allows the specification of agents, it seems a natural way to specify an agent needing parallel tasks by using several internal agents.

3.1 Recursive agents

Given definitions 2 and 1, it can be seen that an agent type

$$T = (\mathtt{name_t}, Vt_i, Vt_s, Vt_{\mathcal{E}}, init, I_t, L, S_t, Actions, \mathtt{Beh}_t)$$

can be considered as an environment $E_t = (V_{\mathcal{E}t}, I_{\mathcal{E}t}, s_{\mathcal{E}t})$ computed as follows:

$$\begin{aligned} V_{\mathcal{E}t} &= Vt_i \cup Vt_{\mathcal{E}} \\ I_{\mathcal{E}t} &= I_t \wedge I_{\mathcal{E}} \\ s_{\mathcal{E}t} &= S_t \end{aligned}$$

So, the following transformer function is introduced:

Notation 1 envir function *Let T be an agent type. We note envir(T) the environment that can be computed from T.*

This transformation allows the consideration of sub-agents:

Definition 12 (Recursive agent and Sub-agent) Let A and R be two agents. If the environment of A is *envir*(R), then R is called a *recursive agent.* A is called a *sub-agent* of R.

Definition 13 (Recursive Multi-Agent System) A *recursive Multi-Agent System* is a multi-agent system made of at least one recursive agent. Such a system can also contain non-recursive agents.

Please note that *sub-agents are also agents*, and so:

– are associated with an agent type with a GDT;
– are specified by an invariant;
– have a set of internal variables;
– run with an interleaving parallelism model;
– ...

Please note also that *a recursive agent is also an agent* and so, the fact that it has a recursive structure is not seen by other agents of the system in which it is situated. This is a key point to preserve the compositional aspect of the GDT model.

By introducing recursive agents, we first want to allow an agent to perform several plans in parallel. As a consequence, sub-agents have to contribute to the goal of the agent they are part of. To do so, new decomposition operators must be introduced. But before that, we have to be able to specify that an agent is *included* in another agent.

Here are some identified properties about sub-agents. Let A be an agent situated in an environment E. Let S be a sub-agent of A. The following properties must be verified:

– S cannot be a sub-agent of another agent (this differs from other recursive agent models, as the one described in [HDF08]),
– S cannot read variables of E that A is unable to see,

- internal variables of A must be considered by S as environment variables because they can also be modified by other sub-agents of A,
- S must be able to exchange information with other sub-agents of A.

We then need to describe the structure of a recursive agent model.

3.2 Relationship between an agent and its sub-agents

In order to help the reader and to maintain consistency with the existing terminology, we have chosen to reuse the two following terms introduced in [HDF08].

Definition 14 (Context agent) Let A and a be two agents where a is a sub-agent of A. A is called the *context agent* of a.

Definition 15 (Content agent) Let A and a be two agents where a is a sub-agent of A. a is called a *content agent* of A.

A shared point between the huge number of agent's definitions is that agents are autonomous entities [FG98,WJ95]. This is also our point of view. However, as content agents have to participate to the goals of their context agent, it seems natural that their context agent has a kind of control over them. So, we choose to give to context agents the following responsabilities:

- it is the role of the context agent to specify when a content agent starts to live;
- although a content agent can die when it wishes, its context agent can also force it to die.

These responsabilities will be operated differently, according to the decomposition operators. As a consequence, it is also clear that a recursive agent is not a MAS: its context agents are parts of a structured single behaviour.

3.3 Parallel decomposition operators (PDO)

3.3.1 Informal definition

Until now, decomposition operators consisted in decomposing a goal into one or several subgoals. Now, new decomposition operators are needed in order to help to decompose a context agent into several content agents. More precisely, as content agents have to contribute to goals of the context agents, these operators decompose a goal of the context agent into sub-agents that will contribute to the resolution of this goal. More formally:

Definition 16 (Parallel Decomposition Operator (PDO)) Let A be a context agent and $ca = \{a_i, 1 \leq i \leq n\}$ the set of its content agents. A *Parallel Decomposition Operator* is an operator allowing the decomposition of a goal G of A into a list of pairs (SC_j, a_j) where a_j is an agent of ca and SC_j is the satisfaction condition of the main goal of a_j.

To simplify, we will write in the sequel that a goal is decomposed by a PDO into subagents.

From then on, two PDOs can be informally described:

Definition 17 (The ParAND PDO) The ParAND PDO (Parallel And) allows a decomposition of a goal G of a context agent A into a set of content agents ca with the following semantics: when A has to achieve a goal G, all the content agents of ca start the execution of their behaviour. As soon as one of these agents stops with a failure (it fails to achieve its main goal), A stops all the other agents of ca and ends the resolution process of G. G may be achieved or not. If all the agents of ca stop successfully, the resolution process of G ends and G is achieved.

Definition 18 (The ParOR PDO) The ParOR PDO (Parallel Or) allows a decomposition of a goal G of a context agent A into a set of content agents ca with the following semantics: when A has to execute goal G, all the content agents of ca start the execution of their behaviour. As soon as one of these agents stops successfully, A stops all the other agents of ca, ends the resolution process of G and G is achieved. If all the agents of ca stop with a failure, the resolution process of G ends. G may be achieved or not.

A synchronised version of the $parAND$ operator is also proposed: the $SyncParAND_V$ operator. This operator enables to lock a set V of environment variables during the execution of the parallel decomposition.

3.3.2 Formal semantics

The semantics of the ParAND and ParOR operators can be now formally specified. We assume that the goal of the context agent to be decomposed is called G. We also consider that this goal will be achieved thanks to n content agents a_i and that the main goal of each agent a_i is called mg_i and its satisfaction condition is SC_i.

For each content agent a_i, A new variable must be introduced: $ended_i$, that memorises the fact that a_i has already ended or not. In other words, when $ended_i$ is true, it means that agent a_i has ended since the beginning of the execution of G. The values of these variables can be computed has follows:

$$\Box(\forall i.(init_G \rightarrow \neg ended_i)) \tag{1}$$
$$\Box(\forall i.(\neg ended_i \wedge \neg end_{mg_i} \rightarrow \circ\neg ended_i)) \tag{2}$$
$$\Box(\forall i.(end_{mg_i} \rightarrow ended_i)) \tag{3}$$
$$\Box(\forall i.(ended_i \wedge in_G \rightarrow \circ ended_i)) \tag{4}$$

Formula 1 specifies that when the execution of G begins, no content agent may have ended. Formula 2 specifies that a non-ended agent cannot be *ended* if it does not end. Formula 3 specifies that an agent becomes *ended* when it ends. And finally, formula 4 specifies that during the execution of G, an *ended* content agent remains *ended*.

As the parent goal of a PDO is a goal, formulae defining the value of in_G and that were given in section 2.4 apply and so, they are not given in the following table. On the contrary, as a PDO does not decompose a goal into subgoals but into content agents, general formulae of the other decomposition operators that were also given in section 2.4 do not apply.

Formal semantics of the parAND operator is given in table 1.

$$init_G \rightarrow \bigwedge_{i=1}^{n} (init_{mg_i}) \tag{5}$$

$$\forall i.\, \big((end_{mg_i} \wedge \neg sat_{mg_i}) \rightarrow \big(end_G \wedge (\forall j.(\neg ended_j \rightarrow end_{mg_j}))\big)\big) \tag{6}$$

$$(\forall i.\, (ended_i) \wedge \forall i.\, (end_i \rightarrow sat_i)) \rightarrow (end_G \wedge sat_G) \tag{7}$$

Table 1 ParAND formal semantics

These formulae have the following meanings:

- formula 5 specifies that all content agents start when the execution of G starts;
- formula 6 specifies that if one of the content agents ends with failure, then the execution of G ends and the other content agents are interrupted (That is to say, for each content agent of an agent A, if it ends with a failure, the executions of the parent goal and of the other content agents of A end) ;
- formula 7 specifies that if all the content agents end successfully, then the execution of G ends successfully. Please note that this formula does not specify that content agents that have ended before the last one ended successfully. Indeed, in this case, formula 6 specifies that the other agents are stopped.

Formal semantics of the parOR operator is given in table 2.

$$init_G \rightarrow \bigwedge_{i=1}^{n} (init_{mg_i}) \tag{8}$$

$$\forall i.\, \big((end_{mg_i} \wedge sat_{mg_i}) \rightarrow \big(end_G \wedge sat_G \wedge (\forall j.(\neg ended_j \rightarrow end_{mg_j}))\big)\big) \tag{9}$$

$$(\forall i.\, (ended_i) \wedge \forall i.\, (end_{mg_i} \rightarrow \neg sat_i)) \rightarrow \circ end_G \tag{10}$$

Table 2 ParOR formal semantics

Formula 9 specifies that when one of the content agents ends successfully, then the parent goal ends also successfully and the other content agents must be interrupted.

Finally, formula 10 specifies that if all the content agents end with failure then the execution of G ends (if already ended content agents had ended successfully, by formula 9, other content agents should have been stopped.

3.4 Using recursive GDT4MAS agents

In previous articles, we have shown that many features of the GDT4MAS model make it suitable to solve concrete problems. For instance, the fact that environment variables can evolve regardless of the agents makes our model usable to model complex systems. The compositional aspect of the proof is also a key feature, because, contrary to verification methods based on model-checking, the verification process remains tractable even with many agents and infinite state spaces. The application of the model to few case studies (a prey-predator system - in which we have proven that the predators catch the moving prey -, the robots on mars problem, and an extension of this problem with several robots of type $R2$) has shown it should be adequate to specify and implement real-size problems, even if the lack of a CASE-tool to support the method prevents us from proceeding to such an application. We claim that the extension proposed in this paper not only does not reduce the usability of the method, but rather should increase it.

There are two main ways to use recursive agents: with a top-down approach or with a bottom-up one. The first case will appear during the design phase, when having to design an agent that must perform several tasks in a concurrent way. The second case is a consequence of the specifications of the studied problem: when an agent relies on several modules, or when the robot being modeled is made of several actuators, and if certain tasks can only be performed by one of these modules or actuators, the designer will use recursive agents.

4 Design patterns using PDO

As stated above, recursive agents can be used to solve different kinds of standard problems. As a consequence, to help the designer to build a MAS, several design patterns are given in order to answer to these different situations.

4.1 Graphical representation

The overall graphical representation of a GDT has already been presented. To distinguish children nodes of PDO that correspond to content agents, they are represented by hexagons. Such an hexagon is labeled by a couple “name : SC” where name is the name of the content agent and SC is the satisfaction condition of its main goal.

4.2 Several autonomous actuators

A basic usage of PDOs is to allow an agent to perform several tasks simultaneously. This is for instance necessary for a robot with several actuators working independently, like an autonomous motion system and an autonomous camera: the robot can move while observing its environment. In this case, a content agent can be associated with each actuator. For instance, in figure 2, agent A_1 would be in charge of moving the robot whereas agent A_2 would manage the camera. The GDT of the content agents are not represented.

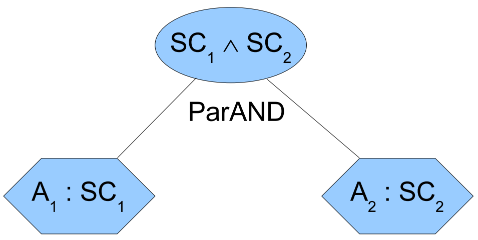

Fig. 2 design pattern for a robot with several autonomous actuators

4.3 Several achievement goals

Another trivial usage of the ParAND PDO is the following: an agent has several achievement goals to achieve. With the previous GDT models, this could be implemented by two different means. The first solution (see figure 3) was to decompose the main goal of the agent with an AND operator between the different achievement goals of the agent. However, this meant that the agent had to achieve completely one of the goals before trying to achieve another one. The second solution (see figure 4) was to propagate the conjunction of satisfaction conditions of the different achievement goals to the leaf nodes. This leads however to a less clear specification.

With the recursive GDT model, the more intuitive specification given in figure 5 can be used. In this version, two content agents (whose GDTs are not represented on the figure) are dedicated to achieve each one of the 2 achievement goals.

4.4 Allowing an external event to interrupt a task

Suppose that an agent A has, as part of its GDT, a task T to perform (that is to say a goal to achieve). Suppose also that depending on external events (a message income, an evolution of the environment, etc.), this task becomes not relevant anymore. This can be implemented thanks to recursive agents by the GDT structure presented in figure 6. In this figure, agent A uses two content

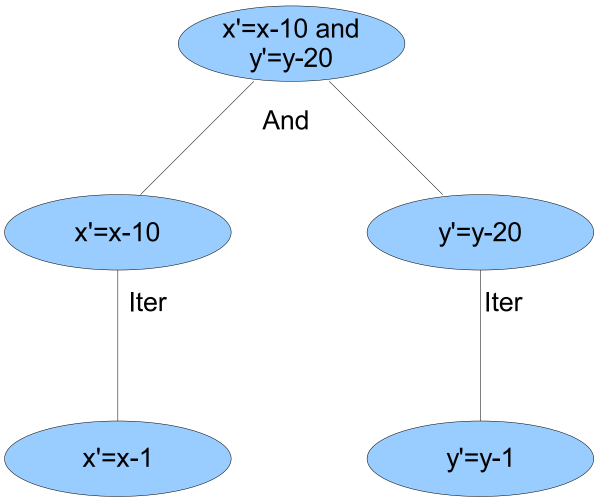

Fig. 3 2 achievement goals: first old version

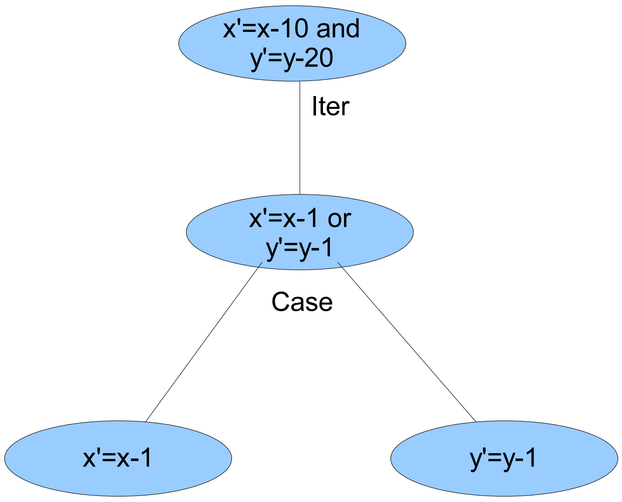

Fig. 4 2 achievement goals: second old version

agents. The first one, A_T is dedicated to the achievement of the task T. The second one, A_E, detects the external event E that would make T not relevant anymore. The expected behaviour of A is the following: if T is achieved before E occurs, then T is achieved for A and it can continue to execute its normal behaviour. On the contrary, if E occurs before T is achieved, the resolution process has to be stopped and A must continue its normal behaviour. This correponds to the ParOr Semantics. The behaviour of agent A_E is very simple: its GDT is made of a single node, its root node, which is also an external goal (and thus, a leaf node), waiting for the external event.

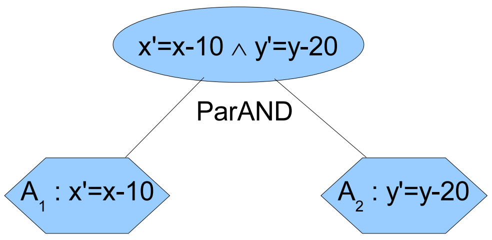

Fig. 5 2 achievement goals: new version

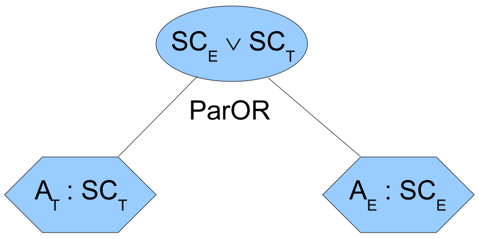

Fig. 6 Interruptible behaviour

4.5 Sharing reactive and autonomous behaviours

Suppose an agent A has an autonomous cognitive behaviour. Suppose then that this agent must also respond to external events in a reactive manner. Such an agent can be decomposed into two content agents: the first one, A_C will achieve the cognitive behaviour of A. The second one, A_R will take into account the reactive behaviour. We might imagine that the end of the cognitive behaviour determines the end of the agent execution, and that the reactive behaviour must remain active while the cognitive behaviour has not been completely performed. This can be implemented by a GDT structure following the schema presented on the figure 7 for agent A and on the figure 8 for the agent A_R. On this last figure, the exernal goal waits for an event to occur (its SC is a formula describing the event) and the goal labeled SC_R is the root goal of a sub-tree. The behaviour of this agent is then the following:

– it waits for an event to occur (external goal);
– when this event occurs ($SeqAnd$), it reacts (sub-tree with root goal SC_R);
– as the root goal of this agent is not achieved (its SC is $false$), it waits for the event again;

Please note that this implies to allow *iter* parent goals that are not necessarily satisfiable, which was not allowed before. This can be done because in that case, the termination is however guaranteed (because the execution of A_C will eventually stop and this will stop the execution of A_R, as specified by the formal semantics of $ParOR$). This termination problem will be discussed in section 6.

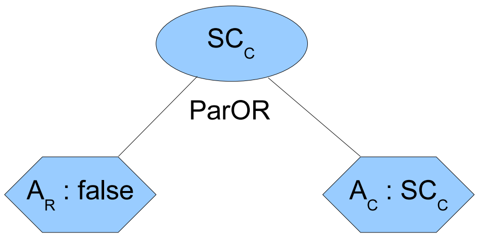

Fig. 7 Reactive and autonomous agent

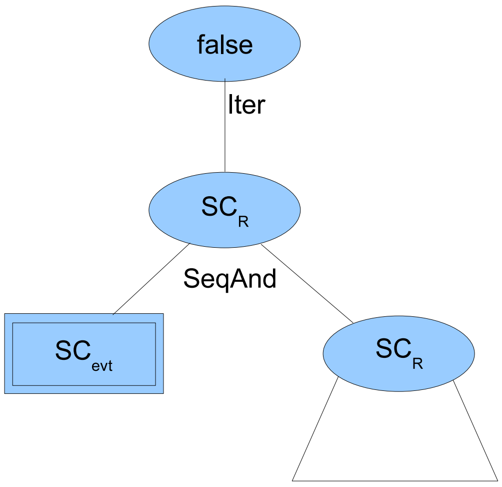

Fig. 8 Reactive and autonomous agent: reactive component

5 Communications

Reacting to a message without waiting explicitly for it was not very easy to specify with the previous version of the GDTs. Indeed, it required to test very often if the message had been received or not. The recursive agents help to give a pretty solution to this problem: the design pattern *sharing reactive and autonomous behaviours* presented above in section 4.5 can be used and the external goal of the reactive component (see figure 8) can be replaced by a leaf goal with the *received()* action attached to it.

To examplify this, in this article, a simplified version of a communication system is described. Firstly, only peer-to-peer messages are studied and not broadcast communications. We also suppose that each agent has an infinite message queue where all incoming messages are stored until they are read. We also suppose messages are read in the order they are received and that reading a message removes it from the queue. Other kinds of communications could be easily modeled but this is not the main aim of this article.

5.1 Sending a message

We choose to specify the sending of a message by an atomic action $send(dest, msg)$ where dest is the identifier of the agent the message is sent to. This action adds to the end of the message queue of the $dest$ agent the couple $(from, msg)$, where $from$ is the identifier of the sender and msg is the message itself. As any agent can write a message into the message queue of a given agent, it means that message queues must be considered, for the proof, as environment variables. So, we choose to represent message queues by a unique functional environment variable mq[3]. As in [MS09], we suppose that $\mathcal{A}$ is an environment variable containing the set of agents of the MAS. The type of mq is the following:

$$mq \in \mathcal{A} \rightarrow seq(\mathcal{A} \times MSG)$$

where MSG is the type of the messages (no assumption on this type is made here) and seq is a set-theory constuctor for sequences.

As a consequence, writing $send(dest, msg)$ as an action of an agent a will be a shortcut for the following expression:

$$mq'(dest) = mq(dest) \bullet (a, msg)$$

where $\bullet$ denotes the addition at the end of a sequence.

5.2 Reading a received message

Reading messages received by an agent a consists in reading the head of $mq(a)$. This will be performed by an action noted *(sender',msg')=received()* returning two values: the sender of the message and the message itself. This action is a shortcut for the following action:

$$(sender', msg') = head(mq(a)) \wedge mq' = tail(mq(a))$$

By convention, if $mq(a)$ is an empty sequence, $head(mq(q))$ equals $(null, null)$ and $tail(mq(a))$ is an empty sequence.

5.3 Managing communications

Managing message sending is straightforward: the *send* action must be associated with a leaf node specifying when a message must be sent.

Waiting for a message is also easy: an external goal consisting in waiting for the message queue to be modified can be introduced in the GDT where the message is awaited. This goal can be part of a content agent dedicated to message reception management, as specified in the introduction of this section, while another content agent achieves the main goal.

[3] Considering message queues as a unique environment variable is only for the proof. Concretely, the message queue of an agent should be a variable of the agent itself.

6 Proof schemas

As stated in section 2.5, a key concept of the GDT4MAS model is the notion of proof schema. This concept, that can also be found for instance in the B method for the verification of standard software [Abr96], gives a way to generate a set of formulae whose truth establishes the correctness of the software. As new constructs are introduced in this article, the associated proof schemas must be provided.

6.1 Variables status

Let A be a context agent. Let b and c be two content agents of A. Then, internal variables of A and environment variables of A are environment variables for b and c. b (resp. c) may moreover have its own internal variables that can be neither seen nor modified by c (resp. b) or A. However, b (resp. c) may have surface variables whose values can be read by c (resp. b) and A.

6.2 Relationships between agents

In the work considered here, a content agent represents a part of a context agent, and as a consequence, cannot be a part of several context agents. So, in the model presented here, the MAS structure is a tree of agents.

6.3 Operators' Proof Schemas

6.3.1 Proof schema associated with parAND

When a goal G is decomposed into a set of content agents sa_i by a *ParAND* operator, it means that when all the content agents have achieved their main goal, the parent goal is also achieved. So, we would expect a proof schema such as: $\bigwedge(SC_i) \rightarrow SC_G$ (where SC_i is the satisfaction condition of the main goal of a_i).

But the two following constraints must be considered:

- as content agents do not end at the same time, environment variables can be modified between the end of a content agent and the end of the last content agent.
- internal variables of content agents cannot be seen by the context agent.

As a consequence, in the proof schemas, the satisfaction conditions of the main goal of the content agents must be restricted to surface variables. We note P_s the projection of a property P of an agent a to the surface variables of a[4]. Then, the first part of the proof schema associated with the *ParAND*

[4] and so, $(SC_i)_s$ is the projection of SC_i to the surface variables of a.

operator, and that helps to satisfy formula 7, is the following:

$$\bigwedge_{i=1}^{n}((SC_i)_s) \rightarrow SC_G \tag{11}$$

Moreover, we also have to establish rule 5 which specifies that when the context agent has to achieve goal G, each subagent of G starts to execute its GDT. Given the definition of the triggering context of an agent given in [MSSZ07], this can be ensured by proving that the context C_G of the parent goal of the *parAND* operator implies the triggering context TC_i of each of the content agents introduced by the operator. So, the second part of the proof schema associated with the *parAND* operator is the following:

$$C_G \rightarrow \bigwedge_{i=1}^{n}(TC_i) \tag{12}$$

For the synchronised version $SyncParAND_V$, the projection used in equation 11 is performed on surface variables and on the locked variables.

6.3.2 Proof schema associated with ParOR

The reasoning process to determine the proof schema associated with the *parOR* operator is the same as the one done for the *parAND* operator. So, the proof schema is made of two parts:

- the first one must establish that the parent goal is achieved when one of the content agents achieves its main goal, and so, helps to establish rule 9;
- the second one must establish that content agents start their execution when the context agent starts executing the parent goal, and so, helps to establish rule 8.

This leads to the following proof schema made of two parts   where $(SC_i)_{s,e}$ is the projection of SC_i to the union of the surface variables and the environment variables of the context agent :

$$\bigvee_{i=1}^{n}((SC_i)_{s,e}) \rightarrow SC_G \tag{13}$$

Please note that the satisfaction conditions of the main goals of the content agents are projected on surface variables and also on environment variables because as soon as a content agent achieves its main goal, the parent goal of the *parOR* operator is achieved.

$$C_G \rightarrow \bigwedge_{i=1}^{n}(TC_i) \tag{14}$$

6.4 Necessary Satisfiability inference

As explained before, the Necessary Satisfiability flag of internal goals of an agent are inferred from the leaf nodes by rules associated with each decomposition operator. Such rules can also be given for parallel operators.

6.4.1 NS inference for the ParAND operator

The parent goal of a *ParAND* operator is NS if all the main goals of its subagents are NS.

6.4.2 NS inference for the ParOR operator

The parent goal of a *ParOR* operator is NS if at least one of the main goals of its subagents is NS.

6.5 GPF Propagation rules

Guaranteed Properties in Case of Failure are properties that are guaranteed to be established by the execution of a goal if it is not achieved. As explained in previous papers about GDTs [MSSZ07], they can be computed in a bottom-up manner from leaf nodes to the root node. The same process can be applied to parallel operators.

6.5.1 GPF inference for the ParAND operator

The parent goal of a *parAND* operator fails as soon as one of its subagents ends without having achieved its main goal. So, we have:

$$GPF_G = \bigvee_{i=1}^{n} ((GPF_i)_{s,e}) \tag{15}$$

where GPF_i is the GPF of the main goal of the content agent a_i. The projection is on surface variables and environment variables because as soon as a content agent ends without having achieved its main goal, the parent goal fails.

6.5.2 GPF inference for the ParOR operator

The parent goal of a *parOR* operator fails if all the subagents have ended without having achieved their main goal. So we have:

$$GPF_G = \bigwedge_{i=1}^{n} ((GPF_i)_{s}) \tag{16}$$

6.6 Termination

Termination is a necessary property to ensure that the behaviour of an agent is correct. Until now, in the standard GDT model, agents had to terminate, and thus, a variant had to be associated with each *iter* operator to ensure this. However, as stated above in section 4.5, it may be interesting to have a subagent whose behaviour does not end explicitly. However, as this content agent must eventually terminate, such a content agent must be used under constraints.

Definition 19 (infIter) A new iteration operator called *infIter* is introduced. The definition of this operator is identical to that of the standard *iter* operator but termination is not required.

Definition 20 (Infinite agent) an *infinite agent* is an agent whose GDT contains an *infIter* operator.

Please note that the behaviour of an infinite agent may be finite.

Proposition 1 Infinite agent usage *An infinite agent can only be used as a subagent of a parOR operator and of which at least one subagent is finite and has an NS main Goal.*

This last proposition guarantees the termination of a recursive agent even if some of its content agents are infinite.

7 Example of application

We propose to extend the problems of robots on Mars already presented in this paper by giving more details on the structure of the robot $R2$. Robot $R2$ has an arm that can collect wastes and put them in a tank with a limited capacity of 10 pieces of garbage. When this tank contains at least 3 pieces of garbage, a conveyor belt brings all the wastes one by one to the incinerator of the robot. From an external point of view, the behaviour of $R2$ remains the same: when its cell is dirty, the robot will eventually clean the cell. From an internal point of view, however, the robot is made of several systems that can act concurrently.

7.1 Robot $R2$

Robot $R2$ has a single internal variable: the tank, represented by an integer variable T. The root goal of $R2$ G_1 is to have its cell empty and not to have its tank full. So we have $SC_{G_1} = G(x_{R2}, y_{R2}) = \mathit{clean} \wedge T < 10$. Robot $R2$ will begin to execute its behaviour when its cell is not empty or when its tank contains at least 3 pieces of garbage. So we have $TC_{R_2} = G(x_{R2}, y_{R2}) = \mathit{dirty} \vee T \geq 3$. Then the behaviour of $R2$ consists in cleaning its cell and saving the tank to be full. These two tasks will be performed by the two effectors of the robots (the arm and the conveyor belt) and so, are associated with the root goal using a $SyncParAND$ operator. The GDT of $R2$ is shown in figure 9.

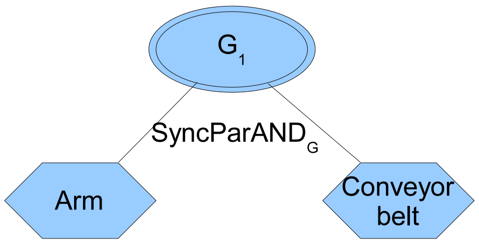

Fig. 9 GDT of robot $R2$g

7.2 The arm of $R2$

The arm of $R2$ has a single internal variable, *busy*, a boolean that is true if and only if the arm carries a waste. As a content agent of $R2$, its environment variables are the internal variables and the environment variables of $R2$. Here, environment variables of the arm are G, the grid (G is an environment variable of $R2$) and the tank T (T is an internal variable of $R2$). The triggering context of $R2$ is *true*. Two actions are available for the arm: *pick* and *drop*. Both actions are NS. Their preconditions are $pre_{pick} \equiv \neg busy \wedge G(x_{R2}, y_{R2}) = dirty$ and $pre_{drop} \equiv busy \wedge T < 10$. Their postconditions are $post_{pick} \equiv busy \wedge G(x_{R2}, y_{R2}) = clean$ and $post_{drop} \equiv \neg busy \wedge T' = T + 1$.

The GDT of the arm of $R2$ is presented in figure 10. The main goal of the arm is lazy. Indeed, if the cell (x_{R2}, y_{R2}) is not dirty, the arm has nothing to do. Otherwise, it must first pick the piece of garbage and then, it must drop the waste it carries in the tank as soon as it is not full. As the arm is unable to empty the tank, emptying the tank is an external goal. Please note that the arm can drop its waste in the tank while the conveyor belt is emptying it (as soon as there is at least one slot available).

7.3 The conveyor belt

The conveyor belt has no internal variable. Its only environment variable is the tank (T). The triggering context of $R2$ is *true*. The only action this agent can perform is the *convey* action. Its precondition and postcondition are $pre_{convey} \equiv T > 0$ and $post_{convey} \equiv T' = T - 1$. This agent has also a leads-to property: $T = 10\, leads - to\, T = 0$.

The main goal of the conveyor belt is to prevent the tank from being full. This goal is of course lazy. If the tank contains less than 3 wastes, there is

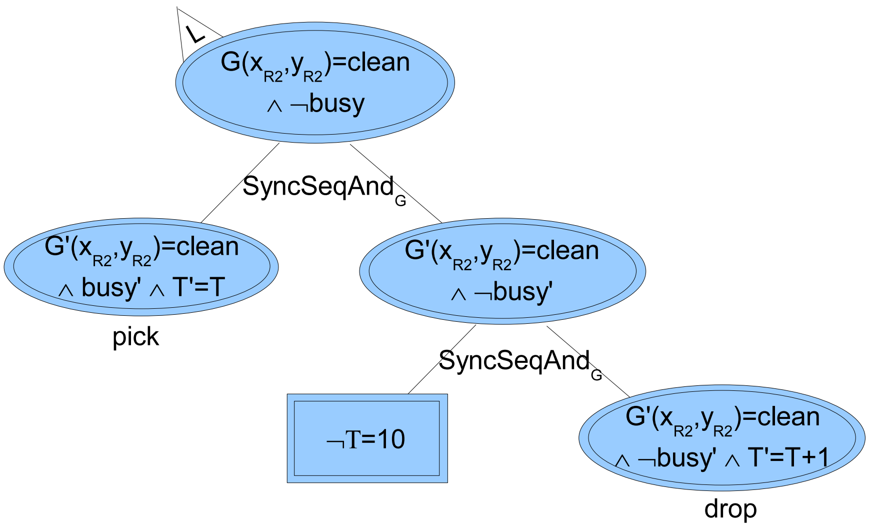
G(x_R2,y_R2)=clean ∧ ¬busy
SyncSeqAnd_G
G'(x_R2,y_R2)=clean ∧ busy' ∧ T'=T
pick
G'(x_R2,y_R2)=clean ∧ ¬busy'
SyncSeqAnd_G
¬T=10
G'(x_R2,y_R2)=clean ∧ ¬busy' ∧ T'=T+1
drop

Fig. 10 GDT of the arm of R2

nothing to do. Otherwise, the agent iterates on the action *convey* to empty the tank.

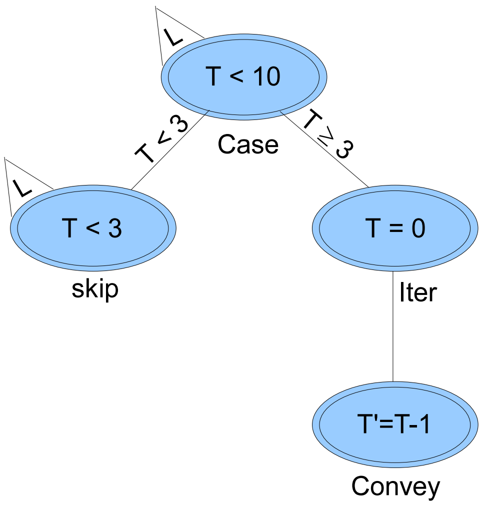

Fig. 11 GDT of the conveyor belt of R2

7.4 Verification of the behaviour of $R2$

To verify the correctness of the behaviour of $R2$, we have to apply the proof schemas on each content agent of $R2$ (the arm and the conveyor belt) but also on the main agent itself ($R2$).

7.4.1 R2

As the behaviour of $R2$ is reduced to a *parAND* operator, we just have to apply the proof schema associated with this operator and that is given in equations 11 and 12. Both of the following properties are obtained:

$$(G(x_{R2}, y_{R2}) = clean) \wedge T < 10 \rightarrow (G(x_{R2}, y_{R2}) = clean \wedge T < 10) \tag{17}$$

$$C_{G1} \rightarrow (true \wedge true) \tag{18}$$

The verification of both formulae is trivial. For the second one, the value of C_{G1} is not calculated because it is not required to verify the property.

7.4.2 arm of R2

Verifications of both *SyncSeqAnd* decompositions are trivial and so are not presented here, as well as the verifications linked to the leaf goals. We briefly detail the verification associated with the achievement of the external goal (called E in the sequel). Given the proof schema presented in [MS09], we have to prove that there is in the system an agent with a leads-to property $A\, leads-to\, B$ with $C_E \wedge \neg SC_E \rightarrow A$ and $B \rightarrow SC_E$. Indeed, another agent in the system, the conveyor belt, has a leads-to property $T = 10 \rightarrow T = 0$. Moreover, $T = 10$ is a logical consequence of $\neg SC_E$ (as $SC_E = \neg(T = 10)$) and $T = 0$ implies SC_E. So the achievement of the external goal each time it is required is proven.

7.4.3 conveyor belt of R2

Verifying the behaviour of the conveyor belt is a standard proof of a GDT: verifying the case decomposition is trivial (as $T < 3 \rightarrow T < 10$ and $T = 0 \rightarrow T < 10$) and verifying the Iter decomposition is easy as the variant to choose is T (a natural), the subgoals makes T decrease, and the fact that the variant reaches its lower bound ($T = 0$) trivially implies the satisfaction condition of the parent goal ($T = 0$).

8 Comparison with other works

Several works dealing with recursive agents exist. In the sequel, we compare our proposal with other ones.

8.1 Motivation of the recursive model

Some work use recursive agents to model organisations. These organisations can be human organisations [ASM08] or organisations of agents [HDF08, DFH08]. The aim of the latter is to propose a very general but simple framework which can be used to specify most of the proposed models for organisational structuring in multiagent systems such as teams [CL91, Tid93], roles and organisations [FG98] or roles and permissions [NVSB$^+$07]. The aim is clearly different from ours: whereas they want to model organisations and groups, we want to model components of an agent. This explains a first difference between their work and ours: for them, an agent can be part of several context agents but for us, this is impossible. However, although the main aim is different from ours, our proposal is very close to this framework which is inspired from the group notion used in METATEM [Fis94]. Please notice that, although this framework is intended to be independent of the underlying language for agents, this language is restricted to a “BDI-like” language, contrary to our proposal.

In [GB04], the authors try to specify a model for an abstract recursive agent as a mean to integrate different pre-existing MAS. Using this model should help to consider each MAS as a whole, i.e. as an agent with individual goals, beliefs, actions... This possibility is essential to be able to use the MAS in a new context. Please note that in this proposal, an abstract recursive agent exists only during the analysis and design stages but has no implementation. Actually the authors claim that abstract agents are “modeling artifacts to represent MAS” and their behaviour. More precisely, as abstract agents are recursive ones, they contain agents from behaviour of which the behaviour of the abstract agent can be defined. An abstract recursive agent can be seen as a set of interacting agents when considering its internal structure but can also be seen as a single agent with its own properties resulting from those of its component agents. These concepts are illustrated with an example dealing with mobile physical cooperative robots. A robot can be modeled by an abstract agent containing multiple interacting agents, each of which being responsible for a part of the robot functionalities; this is comparable to one of the design patterns proposed in this paper (see section 4.2).

Holonic manufacturing systems [BG08] typically use recursive agents to model robots made of several actuators, which corresponds to one of the design patterns we propose to help a designer to use our recursive agents.

Finally, our work present some similarities with the proposals described in [JT02, GHG$^+$08]. Indeed, in these both articles, recursive agents are used to help to model complex systems.

8.2 Verification

Most recursive agent models do not deal with verification. Indeed, they have not been developed for that (see [ASM08] for instance).

In [GB04] for instance, ordered sequences of actions are used to obtain the set of goals of the abstract agent using a function f. Unfortunately, even if the authors give properties about this function, they do not provide any mean to determine this function in a given context, and so no verifcation can be performed.

In their papers, Fisher *et al.* explain that a context agent adds a kind of *glue* to its content agents thanks to a special specification that they call N_SPEC (this glue is a formal specification that makes all the content agents work together). However they do not give any key to write such a specification or guarantee its consistency with the specification of the content agents. And so, once again, proofs can no be performed.

Verification is however taken into considerations in a few works. In [JT02], the authors present a way to specify a multiagent system with a tree of properties dedicated to sub-processes. Moreover, examples of compositional proofs are presented. Our solution gives however many answers to questions raised by this article. For instance, Jonker *et al.* use a few decomposition operators without giving their semantics, and properties to verify are not justified whereas our proof schemas and their verification validate our method.

The verification is clearly the main goal in [GHG$^+$08], where examples of proofs are performed thanks to model checking and thanks to theorem proving by induction. However, verifications are performed on roles, which does not take into account compositional problems occuring when an agent implement several roles. Moreover, the notion of proof obligation is not integrated to the work, bringing no warranty on the correctness of the system verified.

8.3 Expressiveness

In several models, an agent can be a content agent of several subagents [GHG$^+$08, ASM08,HDF08]. Moreover, in these model, the recursive structure changes dynamically. This is well suited to model organisations, a goal of most of these models, but it makes proofs harder, and we chose a less expressive model to increase the success rate of proofs. To simplify proofs once again, our content agents are not visible outside their context, contrary to most holonic systems [ASM08,Fis99] and to the model presented in [HDF08]. This makes the model more compositional, maximizing reuse and compositionality (let recall that compositionality is a key concept to verify multi-agent sytems). However, we introduced environment variables and surface variables to express interactions between agents, a concept missing for instance in [JT02].

The problem of task interruption has been addressed in few articles, but never with recursive agents. The solution proposed in [THMYS07] propose repair plans when a task is aborted, a feature that our design pattern does not provide yet, but in this article, tasks cannot be interrupted anytime, and a task that must be interruptible must be modified, whereas our construct does not require to modify the task itself.

8.4 Gap between specification and implementation

The model we propose is the only one we know that, thanks to automata composition patterns associated to decomposition operators, allows a code generation from a verified specification.

9 Conclusion and Perspectives

In this paper, a new extension of the GDT model has been presented. The first aim of this extension, called "recursive GDT agents", is to allow agents to run parallel plans, that is to say to execute two subgoals at the same time. This was not really possible in the previous version of the GDT model: an agent had to finish the execution of a subgoal before beginning the execution of another one. The main principle of this proposal is to introduce the notion of "content agent". As a consequence, at different points in time during its execution, an agent can choose to manage the execution of subgoals resulting from a goal decomposition by assigning them to content agents. Content agents are GDT agents, which are not perceived by agents which are outside their context agent, and which can be controlled by the agent they are included in. As presented in the paper, this notion is very close to context and content notions proposed by Fisher et al. in [HDF08]. That is why the same terminology has been used for our proposal even if some differences exist.

Formal definitions of the notions of content agent and context agent have first been given. Decomposition operators called "parallel decomposition operators" have also been introduced. These operators help to decompose a goal into subgoals which are executed by different content agents, and to specify formally how a context agent and its content agents interact. Such subgoals are modeled by leaf nodes in the GDT of the context agent. The formal semantics of these operators is provided using logical variables and LTL rules as for other decomposition operators.

We have also shown that the recursive GDT notions give pretty solutions to very common behaviours by providing design patterns to build GDT agents, that is to say partial GDTs which can be instantiated and then integrated in effective GDTs. Another interesting aspect of this new proposal has also been shown: the correctness of the behaviour of GDT agents using content agents can be easily proven (contrary to the other recursive agent models we studied). Indeed, proof schemas associated with the new parallel operators are given. The main interest of this extension is that, with the same mechanism, various behaviours can be specified. More precisely, as far as we know, this is the only work until now that leads to prove in a compositional way the behaviour of agents with interruptible behaviours.

Several perspectives derive from the work presented in this article. At first, we wish to extend thes set of design patterns presented here to design a kind of library which could be associated with a MAS design method using the GDT model. This library can be also completed by parameterised GDT's presented in [MSSZ07]. We also aim at completing the set of design patterns with new ones helping to model classical communication protocols. The set of parallel decomposition operators introduced in this article is reduced. A future work will also consist in specifying new PDOs. CASE-tools to support the GDT model are limited to prototypes. A great effort has to be performed to develop other tools. Indeed, one of the main interest of the GDT model is that it has been designed to make the verification process automatic, but the implementation has not not yet been fulfilled.

References

[Abr96] J.-R. Abrial. *The B-Book.* Cambridge Univ. Press, 1996.

[ALW04] N. Alechina, B. Logan, and M. Whitsey. A complete and decidable logic for resource-bounded agents. In *Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, 2004.

[ASM08] E. Adam, E. Grislin-Le Strugeon, and R. Mandiau. Flexible hierarchical organisation of role based agents. In *2nd IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops*, pages 186–191, 2008.

[BED$^+$06] A. Bracciali, U. Endriss, N. Demetriou, T. Kakas, W. Lu, and K. Stathis. Crafting the mind of PROSOCS agents. *Applied Artificial Intelligence*, 20(2–4):105–131, 2006.

[BFVW03] R.H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifiable multi-agent programs. In M. Dastani, J. Dix, and A. Seghrouchni, editors, *ProMAS*, 2003.

[BG08] V. Botti and A. Giret. *ANEMONA: A multi-agent methodology for Holonic Manufacturing Systems.* Springer, 2008.

[BvET97] F.M.T. Brazier, P.A.T. van Eck, and J. Treur. *Simulating Social Phenomena*, volume 456, chapter Modelling a Society of Simple Agents: from Conceptual Specification to Experimentation, pages pp 103–109. Lecture Notes in Economics and Mathematical Systems, 1997.

[CHSS01] Scott A. Deloach Clint H. Sparkman and Athie L. Self. Automated derivation of complex agent architectures from analysis specifications. In *Proceedings of AOSE'01*, 2001.

[CL91] P.R. Cohen and H.J. Levesque. Teamwork. *Noûs*, 25(4):487–512, 1991.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[dBHvdHM07] F. de Boer, K. Hindriks, W. van der Hoek, and J-J Meyer. A Verification Framework for Agent Programming with Declarative Goals. *Journal of Applied Logic*, (5):277–302, 2007.

[DdBDM03] M. Dastani, F. de Boer, F. Dignum, and J.-J. Meyer. Programming agent deliberation: An approach illustrated using the 3apl language. In *Proceedings of the Second International Conference on Autonomous Agents and MultiAgent Systems (AAMAS'03)*, 2003.

[DFH08] L. Dennis, M. Fisher, and A. Hepple. Language constructs for multi-agent programming. In *Computational Logic in Multi-Agent Systems: 8th International Workshop, CLIMA VIII, Porto, Portugal, September 10-11, 2007. Revised Selected and Invited Papers*, pages 137–156, 2008.

[FG98] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organisations in multi-agent systems. In *Third international Conference on Multi-Agent Systems (ICMAS'98).* IEEE Computer Society, 1998.

[Fis94] M. Fisher. A survey of concurrent METATEM – the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic - Proceedings of the First Intemational Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Heidelberg, Germany, 1994.

[Fis99] K. Fischer. Holonic multiagent systems - theory and applications. In *Proceedings of the 9th Portuguese Conference on Progress in Artificial Intelligence*, volume 1695 of *LNAI*. Springer Verlag, 1999.

[GB04] A. Giret and V. Botti. Towards an abstract recursive agent. *Integrated Computer-Aided Engineering*, 11:165–177, 2004.

[GHG$^+$08] N. Gaud, V. Hilaire, S. Galland, A. Koukam, and M. Cossentino. A verification by abstraction framework for organizational multi-agent systems. In *AT2AI: Froma Agent Theory to Agent Implementation*, 2008.

[GLL00] G. De Giacomo, Y. Lesperance, and H. J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[HDF08] A. Hepple, L. Dennis, and M. Fisher. A Common Basis for Agent Organisation in BDI Languages. In *Languages, Methodologies and Development Tools for Multi-Agent Systems: First International Workshop, LADS 2007, Durham, UK, September 4-6, 2007. Revised Selected Papers*, pages 71–88, 2008.

[JT02] C. M. Jonker and J. Treur. Compositional Verification of Multi-Agent Systems: A Formal Analysis of Pro-activeness and Reactiveness. *Int. J. Cooperative Inf. Syst.*, 11(1-2):51–91, 2002.

[KLP04] M. Kacprzak, A. Lomuscio, and W. Penczek. Verification of multiagent systems via unbounded model checking. In *Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, 2004.

[KW05] J. Khallouf and M. Winikoff. Towards goal-oriented design of agent systems. In *Proceedings of ISEAT'05*, 2005.

[MFS06] B. Mermet, D. Fournier, and G. Simon. An agent compositional proof system. In *From Agent Theory to Agent Implementation (AT2AI'06)*, 2006.

[MS09] B. Mermet and G. Simon. GDT4MAS: an extension of the GDT model to specify and to verify MultiAgent Systems. In *AAMAS*, pages 505–512, 2009.

[MSSZ07] B. Mermet, G. Simon, A. Saval, and B. Zanuttini. Specifying, verifying and implementing a MAS: A case study. In M. Dastani, A. El Fallah Segrouchni, A. Ricci, and M. Winikoff, editors, *Post-Proc. 5th International Workshop on Programming Multi-Agent Systems (ProMAS'07)*, number 4908 in Lecture Notes in Artificial Intelligence, pages 172–189. Springer, 2007.

[MSZ08a] Bruno Mermet, Gaële Simon, and Bruno Zanuttini. Agent design with Goal Decomposition Trees. Technical report, 2008. Available at `http://www.info.unicaen.fr/~bmermet/GDT/publications/gdt/msz08.pdf`.

[MSZ08b] Bruno Mermet, Gaële Simon, and Bruno Zanuttini. Agent design with Goal Decomposition Trees: Companion paper. Technical report, GREYC, CNRS, Université de Caen Basse-Normandie, ENSICAEN, 2008. Available at `http://www.info.unicaen.fr/~bmermet/GDT/publications/gdt/-msz08companion2.pdf`.

[NVSB$^+$07] P. Noriega, J. Vazquez-Salceda, G. Boella, O. Boissier, V. Dignum, N. Fornara, and E. Matson, editors. *Coordination, Organization, Institutions and Norms in agent systems II (COIN)*, volume 46386. Springer, 2007.

[Rao96] 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.

[RL04] F. Raimondi and A. Lomuscio. Verification of multiagent systems via orderd binary decision diagrams: an algorithm and its implementation. In *Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, 2004.

[SMF06] G. Simon, B. Mermet, and D. Fournier. Goal Decomposition Tree: An agent model to generate a validated agent behaviour. In Matteo Baldoni, Ulle Endriss, Andrea Omicini, and Paolo Torroni, editors, *Declarative Agent Languages and Technologies III: Third International Workshop, DALT 2005*, volume 3904 of *LNCS*, pages 124–140. Springer Verlag, 2006.

[THMYS07] J. Thangarajah, J. Harland, D. Morley, and N. Yorke-Smith. Aborting Tasks in BDI Agents. In *Proceedings of the 6th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-07)*, pages 8–15, Hawaii, may 2007.

[Tid93] G. Tidhar. Team-oriented programming: Prelimaniry report. Technical Report 1993-41, Australian Artificial Intelligence Institute, Melbourne, Australia, 1993.

[TPW03] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and Exploiting Positive Goal Interaction in Intelligent Agents. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003)*, pages 401–408, Melbourne, Australia, 2003.

[WJ95] M. J. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: a survey. In *Intelligent Agents*, volume 890 of *LNCS*, pages 1–39. Springer-Verlag, 1995.

[WJK00] 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.