Technical report

Agent Design with Goal Decomposition Trees

Bruno Mermet · Gaële Simon · Bruno Zanuttini

07/10/08

Abstract We present a formal model for specifying agents, called *Goal Decomposition Trees* (GDT). This model allows to specify the behaviour of the agent with operators and logical conditions for combining actions. Proof obligations, essentially in the logic used for the design, can be generated so as to formally prove that the behaviour of the agent is correct. Finally, automata implementing this behaviour can be automatically generated. Thus our method encompasses the whole design, from specification to implementation, with the ability to prove that the design and implementation are correct. This is made possible by providing GDTs, proof obligations and automata with a common semantics formulated in linear temporal logic.

We describe the whole model, from the specification to the implementation phase, and show the validity of automatic transformations and of proof schemas using the formal operational semantics. We also describe two extensions of the model, namely one to *external* goals, that is, goals which are meant to be executed by another agent in a multi-agent setting, and one to *parameterized* goals, which can serve as verified subGDTs which can be reused in various contexts.

Keywords Goal Decomposition Tree · Agent · Agent design · Formal method

1 Introduction

We present a global approach to specify and implement multiagent systems (MASs), which allows to perform a formal verification of each design step. We

Corresponding author: Bruno.Mermet@univ-lehavre.fr

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,
bruno.zanuttini@info.unicaen.fr

propose to specify the behaviour of each agent by a *Goal Decomposition Tree* (GDT).

A GDT is the specification of how the main goal of an agent is to be achieved, by a tree-structured combination of actions through operators (like sequence, choice, iteration...) and through logical conditions (allowing in particular to specifying goals and conditional branchings). This specification is very intuitive, since it uses the logic chosen for describing the environment and essentially graphical combinations of operators. The environment may be dynamic, which is modelled by descriptors which can change value independently from the agent, and nondeterministic, which is modelled in particular by actions which may fail.

From a GDT, proof obligations can be generated in essentially the same logic as that used for the design. If these obligations are verified, then the approach guarantees that the GDT indeed achieves its goal. This proof process is compositional, that is, proof obligations are generated independently for each part of the GDT, so that modifying a node in the tree does not require to verify the whole tree again. A theorem prover for the chosen logic is intended to perform the task of verifying the obligations, but this can be done by any means. Obviously and unavoidably, the more expressive the logic, the less easy automatic proofs.

Finally, from a GDT, a behaviour automaton can be generated, which provably executes the behaviour specified by the tree. In particular, if the proof obligations have been verified, then the automaton is guaranteed to achieve the specified goals. The automaton can be generated in, or translated to, any programing language of interest.

Our approach can be considered as a CASE tool[1]. Nevertheless, the model is such that the design and the proof process are as independent as possible. For instance, one can use a GDT for specifying the behaviour of an agent, thus using the intuitiveness of the model, and generate an automaton which (provably) executes the specified behaviour. This automaton can then be used for verifying the behaviour by, e.g., model checking, thus bypassing the verification of obligations. This may be of interest if, e.g., a convenient theorem prover is not available, or if some obligations turn out to be undecidable.

An important aspect of our approach is that the whole process is formally proven to be correct. To that aim, we attach an operational semantics to GDTs. This semantics is expressed in a logic built on top of linear temporal logic (LTL), and describes how a GDT is executed; it can be intended as a formal definition of a GDT and of operators. We use this semantics for proving that the obligations generated from a GDT are sufficient for establishing its correctness, and for proving that the automatically generated behaviour automata execute the behaviour specified by the GDT. Nevertheless, this validation of our approach is completely transparent to the designer of a MAS who uses our model.

[1] Computer Aided Software Engineering

By associating a formal operational semantics to GDTs, we also propose a formal specification of the behaviour of agents evolving in a dynamic environment. For instance, our semantics takes into account elements such as continual evolution of environment variables, delays during the execution of the different goals of an agent which lead to uncertainty on the current state of the environment, the possible failure of an action or of a plan, and side effects due to the evolution of the environment or to other agents, which can lead to the achievement of a goal even if the action or the plan which is supposed to achieve it has failed. All these characteristics are very general, and giving them a formalization is of interest for all kinds of MASs, even if they are not specified using GDTs.

Goal Decomposition Trees are of course inspired of existing agent models (Gaia [WJK00], Goals [dBHvdHM00]), agent languages (3APL [HBdHM99]), and general-purpose formal methods (B [Abr96]). For instance, the independant specification of the behaviour of an agent and of its properties is inspired from Gaia; the goal-oriented specification of behaviours comes from Goal; the principle of proof obligations is inherited from the B method, as well as the principle of automatically generating an implementation from a verified specification; the specification of the actions of agents by pre- and post-conditions can be found in 3APL. But many aspects of the method are completely new. This is for instance the case of the compositional aspect of the proof system, the nondeterministic outcome of an action, and the possibility to extend the method by defining new decomposition operators.

Parts of the GDT model have already been presented [MFS06,SMF06, MSSZ07]. In particular, two well-known testbed-applications have served as an illustration of GDTs: The prey-predator system [SM90] and robots collecting garbage on a grid [BFPW03]. In this article, we present the whole approach, including the design phase, generation of proof obligations and the implementation phase. The operational semantics and the associated validation of our approach is completely new, and of independent interest. Finally, we describe two extensions of the model: To external goals and to parameterized GDTs. We illustrate the concepts on simple examples, and we refer the reader to the articles mentioned above for applications of the model to larger systems. Likewise, we do not provide the details and proofs for every operator available; we refer the reader to the companion of this paper [MSZ08].

The paper is organized as follows. Section 2 gives the essential formal preliminaries about logic. Sections 3 to 6 describe the model in details, by describing our assumptions about agents and the environment, the notion of a GDT, the proof process and the implementation by automata, respectively. Section 7 describes the extensions to external goals and parameterized GDTs. Finally, Section 8 gives a detailed comparison of our approach to other existing ones for the design of MASs, and Section 9 presents a discussion and future work.

2 Logical setting

The GDT model makes use of logic at three levels: For satisfaction conditions of goals, for proof obligations, and for verifying the correctness of our approach.

2.1 Specification

Logic is used in the specification phase essentially for expressing satisfaction conditions of goals, preconditions of actions, and branching conditions. Thus the same logical language as the one used for describing the environment is to be used. We write $\mathcal{L}$ for this language. Predicate logic is a good example, where variables and constants represent objects in the worlds and predicates represent their properties.

We write $\models$ for the classical consequence relation associated with $\mathcal{L}$, that is, for any two formulae φ_1, φ_2 in $\mathcal{L}$, $\varphi_1 \models \varphi_2$ holds if every model of φ_1 is a model of φ_2. We also write $V(\varphi)$ for the set of all free variables occurring in a formula φ, and given a set of variables V, we write $\mathcal{L}_V$ for the restriction of $\mathcal{L}$ to those formulae in which all free variables belong to V.

The design phase generally requires the ability to express goals whose success depends on the *evolution* of some predicate, such as the goal of incrementing a value. Thus the designer may use an extension of the chosen logic in which two instants in time are referred to. The value of a variable v at the first instant will be denoted by v, whereas its value at the second instant will be denoted by v'. Observe that a similar notation is used, for instance, in TLA [Lam96]. If V is a set of (unprimed) variables, we also write V' for $\{v' \mid v \in V\}$.

Example 1 (logic) Consider a standard logic of arithmetics, and let v_1, v_2 be two variables with domain $\mathbb{N}$. The formula $v_1 + v_2 = 4$ is a formula in $\mathcal{L}$, while the formula $v_1' > v_1 + v_2$ is one in $\mathcal{L}'$. The former is true exactly when the sum of the values assigned to v_1 and v_2 is 4, and the latter is true when the value of v_1 is more than the sum of v_1 and v_2 was (see Section 2.2).

Importantly, we do not formally assign any temporally-dependent link between variables v and v'. Thus the formulae over $V \cup V'$ are simply formulae in $\mathcal{L}$ with an extended set of variables (in $\mathcal{L}_{V \cup V'}$). So as to avoid ambiguities, we write $\mathcal{L}'$ for the language of such formulas. Observe that primed variables cannot be themselves primed, that is, a formula in $\mathcal{L}'$ refers to at most two instants in time.

2.2 Proof obligations

Formulae in $\mathcal{L}'$ are evaluated with respect to couples of interpretations for $\mathcal{L}$, with the natural semantics. In particular, if ω is a world (interpretation) for $\mathcal{L}$, we write ω' for the world that assigns to each variable v' the value of v in

ω. If (ω_1, ω_2) is a couple of interpretations for a formula $\varphi \in \mathcal{L}'$, we refer to ω_2 as the *primed instant*.

We will sometimes need to extend these notions to a greater (but finite) number of instants in time. In particular, we will use formulae referring to three distinct instants in time, represented by plain variables, variables superscripted with *tmp* (for *temporary*), and primed variables, in this order, with a straightforward semantics over triples of interpretations.

Example 2 (continued) The couple $(\{v_1 = 1, v_2 = 2\}, \{v_1 = 4, v_2 = 0\}')$ satisfies the formula $v_1' > v_1 + v_2$ (regardless of the value assigned to v_2 at the primed instant), while $(\{v_1 = 1, v_2 = 2\}, \{v_1 = 2, v_2 = 0\}')$ does not. The formula $(v^{tmp} > v \wedge v' = v^{tmp})$ is satisfied by the triple of worlds $(\{v = 1\}, \{v = 2\}^{tmp}, \{v = 2\}')$.

In the process of generating proof obligations, we will sometimes need to refer to the same formula φ at various instants in time. For instance, some goals in GDTs will be *lazy*, that is, at execution time the agent will first decide whether the satisfaction condition φ is already true and only otherwise, execute some plan so as to make it become true *after* execution.

If φ is a formula, then $\varphi[\mathbf{V}/\mathbf{V}']$ denotes the formula obtained from it by substituting every occurrence v of an unprimed variable with its primed counterpart, and similarly with superscript *tmp*. We will use the following notation for translating formulas along time.

Notation 1 (translation of formulas) *Let $\varphi \in \mathcal{L}$. Then we write $At'(\varphi)$ for $\varphi[\mathbf{V}/\mathbf{V}']$ and $At^{tmp}(\varphi)$ for $\varphi[\mathbf{V}/\mathbf{V}^{tmp}]$.*

If $\varphi \in \mathcal{L}$, we also write $T'(\varphi)$ for $At'(\varphi)$, $T'_{tmp}(\varphi)$ for $At'(\varphi)$, and $T^{tmp}(\varphi)$ for $At^{tmp}(\varphi)$. If $\varphi \in \mathcal{L}' \backslash \mathcal{L}$, then we write $T'(\varphi)$ for φ, $T'_{tmp}(\varphi)$ for $\varphi[\mathbf{V}/\mathbf{V}^{tmp}]$, and $T^{tmp}(\varphi)$ for $\varphi[\mathbf{V}'/\mathbf{V}^{tmp}]$.

The mnemonics is that the superscript of At denotes the instant when a formula is translated to. The superscript of T denotes the instant at which the formula is evaluated, with respect to the (past) instant denoted by the subscript. In some sense, the formula is translated to between the instants denoted by the subscript and the superscript, respectively.

Example 3 (continued) Let φ be $v_1 + v_2 = 4$. Then $At'(\varphi)$ denotes $v_1' + v_2' = 4$ and $At^{tmp}(\varphi)$ denotes $v_1^{tmp} + v_2^{tmp} = 4$. Now $T'(\varphi)$ and $T'_{tmp}(\varphi)$ both denote $v_1' + v_2' = 4$, while $T^{tmp}(\varphi)$ denotes $v_1^{tmp} + v_2^{tmp} = 4$.

Now let φ be $v_1' > v_1 + v_2$. Then $T'(\varphi)$ denotes $v_1' > v_1 + v_2$, $T'_{tmp}(\varphi)$ denotes $v_1' > v_1^{tmp} + v_2^{tmp}$, and $T^{tmp}(\varphi)$ denotes $v_1^{tmp} > v_1 + v_2$.

Finally, we will often need to consider the *projection* of formulae along two dimensions: Sets of variables and time. Intuitively, projecting a formula onto a set of variables consists in forgetting (existentially quantifying) any other variable, and projecting a formula to an instant consists in existentially quantifying every variable at the other instants.

Definition 1 (projection) Let $\varphi \in \mathcal{L}$ and $W \subseteq V(\varphi)$; write U for $V(\varphi) \setminus W$. Then a *projection of φ onto W*, written $(\varphi)_W$, is any formula $(\varphi)_W \equiv \exists U \varphi$.

If $\varphi \in \mathcal{L}' = \mathcal{L}_{V \cup V'}$, then a *projection of φ onto $W \subseteq V$*, written $(\varphi)_W$, is any projection of φ onto $W \cup W'$. A *projection of φ onto W on the right*, written $(\varphi)_{rW}$, is any projection of φ onto $V \cup W'$. A *projection of φ onto W on the left*, written $(\varphi)_{\ell W}$, is any projection of φ onto $W \cup V'$. Finally, a *projection of φ onto the right*, written $(\varphi)_r$, is any projection of φ onto V'.

If $\varphi \in \mathcal{L}_{V \cup V^{tmp} \cup V'}$, then a *projection of φ onto the left and right*, written $(\varphi)_{\ell r}$, is any projection of φ onto $V \cup V'$.

Example 4 (projection) Let $V = \{v_1, v_2, v_3\}$ and $W = \{v_1, v_2\}$. With the notation of Definition 1, we have $U = \{v_3\}$. If φ is $(v_1 \geq 0) \wedge (v_3 \geq 0) \wedge (v_1 + v_3 = 4)(v_3 \leq v_2)$, then $(\varphi)_W$, defined to be $(v_1 \geq 0) \wedge (v_2 \geq 0) \wedge (v_1 + v_2 \geq 4)$, is a projection of φ onto W.

Now let φ be $(v_2 \geq 0) \wedge (v_3 \geq 2) \wedge (v'_1 \geq v_2 + v'_3) \wedge (v'_3 \geq v_3 + 1)$. Then $(\varphi)_W = (v_2 \geq 0) \wedge (v'_1 \geq v_2 + 3)$ is a projection of φ onto W. We also have that $(\varphi)_{rW} = (v_2 \geq 0) \wedge (v_3 \geq 2) \wedge (v'_1 \geq v_2 + v_3 + 1)$ is a projection of φ onto W on the right. Observe that, contrary to the previous case, all information about v_3 on the "left" instant is retained here. Finally, $(\varphi)_{ri} = (v'_3 \geq 3) \wedge (v'_1 \geq v'_3)$ is a projection of φ onto the right.

Finally, let φ be $(v_1 \geq 2) \wedge (v_1^{tmp} > v_1 + v_2) \wedge (v'_2 = v_1^{tmp} - 1)$. Then $(\varphi)_{\ell r} \equiv (v_1 \geq 2) \wedge (v'_2 \geq v_1 + v_2 - 1)$ is a projection of φ onto the left and right.

Projection φ onto a subset of variables and/or some instant can also be seen as computing the set of all consequences of φ which concern only these variables and/or this instant. In particular, for $\varphi \in \mathcal{L}'$ and $W \subseteq V(\varphi)$, we have $\varphi \models (\varphi)_{rW} \models (\varphi)_W$ and for $\varphi \in \mathcal{L}_{V \cup V^{tmp} \cup V'}$, we have $\varphi \models (\varphi)_{\ell r} \models (\varphi)_r$.

Importantly there are in general many formulae logically equivalent to the projection of φ. As a consequence, the precise one computed for, e.g., a proof obligation may have an impact on the possibility to automatically prove (or disprove) this obligation.

Moreover, in general it is a very hard problem to compute a projection, even in the propositional case [LLM03]. But there are efficient algorithms for computing some approximations of a projection, based on the following syntactic properties.

Proposition 1 (computing projections) *Let $\mathcal{L}$ be a subset of standard first-order logic. Let φ_1, φ_2 be two formulas in $\mathcal{L}_V$, and let $W \subseteq V$. Then:*

- $((\varphi_1 \vee \varphi_2))_W \equiv (\varphi_1)_W \vee (\varphi_2)_W$,
- $((\varphi_1 \wedge \varphi_2))_W \models (\varphi_1)_W \wedge (\varphi_2)_W$,
- *if* $V(\varphi_1) \cap V(\varphi_2) = \emptyset$, *then* $((\varphi_1 \wedge \varphi_2))_W \equiv (\varphi_1)_W \wedge (\varphi_2)_W$,
- $\neg((\varphi_1)_W) \models ((\neg\varphi_1))_W$,
- $(\exists X \varphi_1)_W \equiv \exists W((\varphi_1)_W)$ *($X \cap W = \emptyset$ holds by definition of projection),*
- $(\forall X \varphi_1)_W \models \forall W((\varphi_1)_W)$ *($X \cap W = \emptyset$ holds by definition of projection).*

For instance, if φ is a propositional formula in disjunctive normal form, then it can be seen that projecting it onto a subset of variables simply consists in removing any literal over another variable.

2.3 Validation of our approach

We use linear temporal logic (LTL) for giving an operational semantics to GDTs. Given a set of Boolean variables V, the LTL formulae over V are these formulae built from the variables in V, the usual propositional connectives (we will use $\neg, \wedge, \vee, \rightarrow$) and the standard modalities $\Box$, $\diamond$ and $\circ$.

LTL formulae are evaluated with respect to traces, that is, infinite sequences of worlds $(\omega_1, \omega_2, \omega_3, \ldots)$, where each world is labelled with an interpretation of the variables. Traces implicitly define an order on worlds. Thus, if the trace is clear from the context, we will use $\omega_a \prec \omega_b$ to denote that ω_a (properly) preceedes ω_b in this trace, that is, $\omega_a = \omega_k$ and $\omega_b = \omega_{k'}$ for some k, k' with $k < k'$. Moreover, in this case ω_a is said to be *before* ω_b. We will similarly use notation $\omega_a \preceq \omega_b$, $\omega_a \succ \omega_b$ (ω_a is *after* ω_b) and $\omega_a \succeq \omega_b$. Finally, we will write $\circ\omega_a$ for the world immediately following ω_a in the trace.

The validity of an LTL formula φ is evaluated with respect to a trace τ and a world ω_a in it, according to the standard semantics. We write $\tau, \omega_a \models \varphi$, or $\omega_a \models \varphi$ when the trace is clear.

Example 5 (LTL formulae) Let v_1, v_2 be two Boolean variables. Then $\varphi = \circ(v_2 \wedge \Box(\neg v_1))$ is an LTL formula. Assume ω_a, ω_b both assign v_1 to false and v_2 to true, and ω_c assigns both to false. Then the sequence $\tau = (\omega_a, \omega_b, \omega_c, \omega_c, \ldots)$ (ω_c infinitely repeated) is a trace. We have $\tau, \omega_a \models \varphi$ but $\tau, \omega_b \not\models \varphi$ (since $\circ\omega_b = \omega_c \models \neg v_2$).

3 Environment and agents

In this section, we present the assumptions we make about the environment and the agents, and introduce some notions and notation. Throughout the section (and the rest of the paper as well), an underlying logic $\mathcal{L}$ is assumed (think, e.g., of predicate logic).

3.1 Environment

The environment is assumed to be described through a set of *variables*, which are variables in $\mathcal{L}$. This set is denoted by $V_\mathcal{E}$.

The only assumptions about the environment are an *invariant* and a set of *stable properties*. The invariant is a formula $i_\mathcal{E} \in \mathcal{L}_{V_\mathcal{E}}$, which is always true. Consequently, it can always be assumed to be true when initiating an action, but it must also be preserved by the agents' behaviour. Proof obligations will enforce that. The semantics in temporal logic is simply $\Box i_\mathcal{E}$. Note that

invariants, like in formal methods such as B [Abr96] or Z [Spi87], give in particular a mean to formalize the domain of each variable, with properties such as $v \in \mathbb{N}$, provided the logic is expressive enough.

Stable properties are properties which, once true, are guaranteed to remain true, but which need not become true one day. Once again, they must be preserved by the agents. The set of stable properties is denoted by $S_{\mathcal{E}}$; it is just a finite set of formulae in $\mathcal{L}_{V_{\mathcal{E}}}$. The semantics in temporal logic is that for all $s_{\mathcal{E}} \in S_{\mathcal{E}}$, $\Box(s_{\mathcal{E}} \rightarrow \circ s_{\mathcal{E}})$ holds.

All these assumptions are intended to come from a modelisation of the environment which takes place prior to the use of our method. To sum up, the environment can be defined as follows.

Definition 2 (environment) An *environment* is a triple $\mathcal{E} = (V_{\mathcal{E}}, i_{\mathcal{E}}, S_{\mathcal{E}})$, where $V_{\mathcal{E}}$ is a set of variables, $i_{\mathcal{E}} \in \mathcal{L}_{V_{\mathcal{E}}}$, and $S_{\mathcal{E}} \subseteq \mathcal{L}_{V_{\mathcal{E}}}$; $S_{\mathcal{E}}$ is assumed to be finite. Formulae $\Box i_{\mathcal{E}}$ and $\Box(s_{\mathcal{E}} \rightarrow \circ s_{\mathcal{E}})$ (for all $s_{\mathcal{E}} \in S_{\mathcal{E}}$) hold.

Example 6 (environment) Consider the Robots on Mars Problem. The content of each cell (x, y) on the surface can be modelled by an environment variable, written G, assigning to each cell a value *dirty* if there is garbage on the cell, and *clean* otherwise. Then an invariant property could be: $G \in x_{min}..x_{max} \times y_{min}..y_{max} \rightarrow \{clean, dirty\}$.

3.2 Agents

Each agent is essentially described through a set of variables and a set of actions (which it can perform). Moreover, it is attached a behaviour using these variables and actions.

Each action is described by a precondition and a postcondition, as, e.g., in 3APL [HBdHM99]. Both are described in the underlying logic $\mathcal{L}$ (or $\mathcal{L}'$, see below). Nevertheless, we do not assume that actions always succeed. That is, the postcondition may not be established by the action, even if the precondition has been respected. In this case, a *guaranteed property on failure* (GPF) is established. The GPF of an action will most of the time assert that variables have not changed value, but it can assert more complex things (see Example 7).

Definition 3 (action) An *action* a is a 4-tuple $(name_a, pre_a, post_a, gpf_a)$, where $name_a$ is the name of the action, pre_a is a formula in $\mathcal{L}$ and $post_a$ and gpf_a are formulae in $\mathcal{L}$ or in $\mathcal{L}'$. Let α_a be a world at which an agent starts to execute such an action. Then $\alpha_a \models pre_a$ must hold.

Example 7 (action) Let a_1 be an action allowing a robot to move horizontally by one cell to the right on a 2D grid. This action may succeed, but it may also fail (if the target cell is not free), and then the robot stays in the same cell. Moreover, the robot cannot execute the action at all if it is in the rightmost column of the grid. Then this action could be described by

$name_{a_1} = moveRight$, $pre_{a_1} = (x < x_{max})$, $post_{a_1} = (x' = x+1 \wedge y' = y)$, and $gpf_{a_1} = (x' = x \wedge y' = y)$. The GPF is necessary to specify that the failure of the action does not move the robot anywhere.

As another example, consider an action a_2 allowing a robot to (try to) open a door. Then it could be described as follows: $name_{a_2} = open$, $pre_{a_2} = doorClosed$, $post_{a_2} = doorOpen$, and $gpf_{a_2} = doorLocked$. Observe that $post_{a_2}$ is a formula in $\mathcal{L}$, that is, it is evaluated only with respect to the current instant (that when the action finishes). Also observe that the GPF of the action allows to formalize the knowledge acquired from failure of the action.

So as to distinguish between actions which always succeed and actions which may fail, and to reason about them for specifying and verifying behaviours, we introduce the following definition.

Definition 4 (NS/NNS, actions) An action a is said to be *necessarily satisfiable* (NS) if for all worlds ω_a (resp. α_a) at which an agent ends (resp. starts) executing a, (α_a, ω'_a) satisfies $T'(post_a)$. For such an action a, we assume $gpf_a = \bot$.

Otherwise, a is said to be *nonnecessarily satisfiable* (NNS). In this case, for all worlds α_a, ω_a as above, either a succeeds, and $(\alpha_a, \omega_a)'$) satisfies $T'(post_a)$, or a fails and (α_a, ω'_a) satisfies $T'(gpf_a)$.

Importantly, every action is assumed to be either NS or NNS in the above meaning, that is, every action establishes its postcondition in case of success and its GPF in case of failure. It is also assumed that any execution of an action terminates (in a finite amount of time).

Actions are the means by which an agent can modify the values of variables. Each agent A knows a subset $V_{\mathcal{E}}(A)$ of the set of environment variables $V_{\mathcal{E}}$, and also owns a set of *internal variables*, written $V_i(A)$. The semantics is that only A is allowed to modify the values of the variables in $V_i(A)$.

Finally, like the environment, an agent A has to respect an *invariant* i_A and a set of *stable properties* S_A concerning its internal variables. At the beginning of its lifetime, it is initialized by an *initialization clause* $\texttt{init}_A$, which gives the initial values of its variables. Naturally, this clause must establish i_A.

Definition 5 (agent) Let $\mathcal{E}$ be an environment. An *agent* A (in $\mathcal{E}$) is a tuple:

$$(V_i(A), V_{\mathcal{E}}(A), \texttt{init}_A, i_A, S_A, Actions_A, \texttt{Beh}_A)$$

where $V_i(A)$ is a set of variables with $V_i(A) \cap V_{\mathcal{E}} = \emptyset$, $V_{\mathcal{E}}(A) \subseteq V_{\mathcal{E}}$, $\texttt{init}_A$ is a mapping from $V_i(A)$ to values, $i_A \in \mathcal{L}_{V_i(A)}$, $S_A \subseteq \mathcal{L}_{V_i(A)}$ (and is finite), and $Actions_A$ is a set of actions whose preconditions, postconditions, and GPFs concern only variables in $V_{\mathcal{E}}(A) \cup V_i(A)$. and $\texttt{Beh}_A$ is the behaviour of the agent [2]. It is assumed that the interpretation of $V_i(A)$ as defined by $\texttt{init}_A$ satisfies i_A.

[2] In our method, this behaviour is specified by a GDT, described in Section 4.

Again, this description comes as the result of agentifying the problem at hand, which takes place prior to the use of our method. However, some additional descriptors may be added by the designer in charge of proving the agent's behaviour.

Example 8 (agent) Mapping a mobile robot to an agent may induce descriptors for the current angle of its wheels. Then the invariant of the agent could enforce that this angle is never greater than 90 degrees. Other descriptors could concern the resources of the agent, such as an amount of gasoline.

However, so as to prove the correctness of some behaviour of this agent, for instance one aiming at reaching a given location, it could be useful to add a descriptor for its current distance to a goal.

We also wish to point out that our formalism is general enough for taking biased perceptions into account. For instance, assume an environment variable c which formalizes the color of an object as a tuple (r, g, b). A black-and-white perception of this object by an agent may be formalized by introducing a new Boolean environment variable, say bw, adding to the environment invariant $i_{\mathcal{E}}$ a property ensuring that bw is "black" when (r, g, b) is dark enough, and "white" otherwise, and finally letting bw but not (r, g, b) in $V_{\mathcal{E}}(A)$.

We now give an important assumption concerning internal variables. This assumption states that only actions modify their values. That is, we assume a kind of frame axiom for all internal variables of an agent.

Assumption 1 (frame axiom) *Let A be an agent, and let $\varphi \in \mathcal{L}_{V_i(A)}$. If ω is a world such that ω satisfies φ and $\circ\omega$ does not satisfy φ, then this is the result of A beginning an action a at world ω (and finishing it at world $\circ\omega$) so that $\omega \models pre_a$ and either $\circ\omega \models post_a \models \neg\varphi$ or $\circ\omega \models gpf_a \models \neg\varphi$.*

Finally, since internal variables of agents are of great importance in proof obligations, we will use the following notation.

Notation 2 (internal variables) *Let A be an agent. We write $(\varphi)_i$ for $(\varphi)_{V_i(A)}$, that is, for the projection of φ onto the set of all internal variables of agent A. Similarly, we will write $(\varphi)_{ri}$ for $(\varphi)_{rV_i(A)}$, $(\varphi)_{\ell i}$ for $(\varphi)_{\ell V_i(A)}$, $(\varphi)_{i,V}$ for $(\varphi)_{V_i(A)\cup V}$ and $(\varphi)_{r(i,V)}$ for $(\varphi)_{r(V_i(A)\cup V)}$.*

3.3 Parallelism model

As concerns the interactions between agents, we use the model of *interleaved parallelism*. That is, we assume that the behaviours of the agents may be interleaved in any manner, but that actions are supposed to be noninterruptible (intuitively, atomic).

More precisely, the evolution of the environment results from any interleaving of the traces of the agents, as defined in Section 4.3. Nevertheless, when an agent is executing an action, the environment is assumed to change only according to this action. This is similar to assuming a restricted frame axiom

on the values of variables, so that environment variables do not change values while an agent is executing an action unless explicitly stated by this action. Contrastingly, their value can *a priori* change without any restriction while no agent is executing any action.

4 Specification of the behaviour of agents

In this section, we present the design phase in details. In our approach, to each agent in the system a behaviour is attached. The agent will execute this behaviour when some conditions are met. The language which we propose the designer of the agent to use is that of *Goal Decomposition Trees* (GDTs). The GDT of an agent thus specifies how it can achieve its main goal, using its variables and actions.

A GDT is essentially a tree of goals, in which each goal is decomposed into simpler subgoals, combined via a decomposition operator. In order to formally define GDTs, we first define leaf and internal goals, and only then GDTs themselves.

Importantly, we formalize the behaviour of an agent as a single decomposition tree, aiming at achieving a main goal. Nevertheless, it is easy to see that the extension to a set of GDTs for each agent can be easily done, with mutually exclusive triggering contexts or not. In particular, this can be formalized as a single GDT whose root goal is decomposed into several $Case(\cdot, \cdot)$ branches (similar to the "switch" construct in imperative languages).

4.1 Goals

Each node in a GDT corresponds to a goal of the agent. In particular, the root corresponds to its main goal. Consequently, to each node a *satisfaction condition* is attached as well as a plan to achieve this condition.

Just as postconditions of actions, the satisfaction condition of a node may concern only the state of the universe at which it is evaluated, or relate the state when the execution of the node starts and that when it stops. Naturally, in the former case it is a formula in $\mathcal{L}$, whereas in the latter it is a formula in $\mathcal{L}'$.

Leaves of the GDT correspond to elementary goals, in the sense that in their context, an action of the agent allows to achieve the satisfaction condition. On the contrary, internal nodes need to be decomposed into subgoals. Consequently, to each internal node a *decomposition operator* is attached as well as an ordered set of nodes (whose number depends on the decomposition operator). The operator specifies how these children must be executed.

We distinguish two types of nodes in a GDT. If a node N is tagged *lazy* (L) by the designer, then when execution comes to N, the agent first evaluates the satisfaction condition of N. Then it executes the action or decomposition attached to N if and only if this satisfaction condition is not already true. If

N is tagged as *nonlazy* (NL), then the agent always executes the associated action or decomposition.

Intuitively, any node whose satisfaction condition is in $\mathcal{L}$, that is, concerns only the current state of the universe, ought to be lazy, whereas any node whose satisfaction condition is in $\mathcal{L}' \setminus \mathcal{L}$ cannot be lazy. However, for specific needs, other combinations are permitted.

Example 9 (laziness) A node with satisfaction condition $v_1 + v_2 = 4$ will typically be tagged as L, meaning it is worthless trying to achieve this condition if it is already true. On the contrary, a node with satisfaction condition $v_1' > v_1 + v_2$ is necessarily NL.

As a special case, consider the goal of lighting a candle with a match. The lazy fashion consists in checking first whether the candle is alight, and strike the match only if it is not. On the contrary, the nonlazy approach always strikes the match. Thus, in the lazy fashion, some wind may rise up while we are checking the candle state, potentially preventing us from striking the match. Contrastingly, with the nonlazy fashion, we are guaranteed to succeed. Thus, in this case it could be interesting to tag the node NL even if it could be tagged L (and this interest would show up when trying to prove the obligations).

Orthogonal to this distinction, we also distinguish *necessarily satisfiable* (NS) nodes from *nonnecessearily satisfiable* (NNS) ones. The notion is similar to that for actions (see Definition 4): If a node N is labelled NS, then its satisfaction condition is guaranteed to be true when its action or decomposition has been executed. On the contrary, if N is labelled NNS, then execution may fail to make its satisfaction condition true.

Nevertheless, contrary to the case of actions (and to laziness), the necessary satistiability of a node does not have to be known in advance nor specified by the designer. Instead, it can be automatically inferred from the decomposition operators and the necessary satisfiability of elementary goals (see Section 4). If the designer however labels some nodes, then these inference rules allow her to check whether the labels are correct.

Example 10 (necessary satisfiability) Consider the goal of moving by one square up and left on a 2D-grid, with an action allowing to move up by one square, and one allowing to move left by one square. Then achievement of the main goal can be decomposed into the execution of both actions once, in any order (operator *And*). If both execution succeed, then the main goal is achieved. But if any one fails, then the main goal is normally not achieved. Thus the node corresponding to the main goal can be soundly labelled NS if the two children nodes are labelled so, but it cannot *a priori* if one child is not.

Finally, like for actions, to each (NNS) node a guaranteed property on failure (GPF) is attached, which is guaranteed to be true in case the execution of the node fails. Typically, GPFs will state what variables have not changed value when execution fails. Once again these GPFs can be automatically inferred (see Section 5.1).

Definition 6 (leaf node) Let $\mathcal{E}$ be an environment, and let A be an agent. A *leaf node* N (of a GDT for A in $\mathcal{E}$) is a 6-tuple:

$$(name_N, a_N, sc_N, gpf_N, lz_N, nsat_N)$$

where $a_N \in Actions_A$, $sc_N, gpf_N \in \mathcal{L}'_{V_{\mathcal{E}}(A) \cup V_i(A)}$, $lz_N \in \{L, NL\}$, and $nsat_N \in \{NS, NNS\}$. Moreover, if $lz_N = L$, then we must have $sc_N \in \mathcal{L}$ and if $nsat_N = NS$, then $gpf_N = \bot$.

Definition 7 (internal node) Let A be an agent in an environment $\mathcal{E}$. An *internal node* N (of a GDT for A in $\mathcal{E}$) is a 7-tuple:

$$(name_N, Op_N, Children_N, sc_N, gpf_N, lz_N, nsat_N)$$

where $sc_N, gpf_N, lz_N, nsat_N$ are as in Definition 6, Op_N is a decomposition operator, and $Children_N$ is a sequence of internal and leaf nodes whose length matches the arity of Op_N.

4.2 Goal decomposition trees

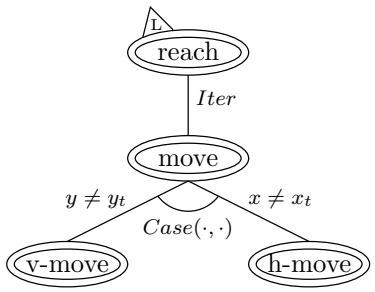
A GDT is essentially a tree built of internal and leaf nodes, but some more information is attached. First of all, the execution of a GDT is meant to begin only when some conditions are met. This is specified by a *trigerring context*, written tc_T, and a *precondition*, written pre_T.

The precondition pre_T formalizes the prerequisite for the GDT to be executed. In particular, if an agent executes its GDT several times, then it must preserve pre_T. Then, the semantics of T is that execution starts each time pre_T is true and tc_T becomes true. Note that this results in a potentially everlasting behaviour. However, a bounded number of executions can be enforced by an internal variable of the agent, whose value is incremented each time T is executed but is upper-bounded in tc_T.

Definition 8 (GDT) Let $\mathcal{E}$ be an environment, and let A be an agent. A *goal decomposition tree* (GDT) for A in $\mathcal{E}$ is a triple $(pre_T, tc_T, Root_T)$, where $pre_T, tc_T \in \mathcal{L}_{V_{\mathcal{E}}(A) \cup V_i(A)}$ and $Root_T$ is an internal or leaf node for A (and thus, implicitly, a tree of such nodes).

Example 11 (GDT) Figure 1 gives an example of a GDT for reaching a location on a grid. The agent has four internal (integer) variables, written x, y for those describing its position, and x_t, y_t for those describing the position of its target location (d denotes a distance between positions). Moreover, it has two actions, written *h-moveOneCell* and *v-moveOneCell*, which allow it to move one cell towards the target location along the x or y axis, respectively.

The GDT reads as follows: The root node *reach* is decomposed into the iteration of node *move*, which itself is decomposed into two (non mutually exclusive) subcases, one when a horizontal move is needed and one when a vertical move is needed. Here, all nodes are supposed to be necessarily satisfiable and nonlazy, except for the root node which is lazy.



$sc_{\text{reach}} = (x = x_t) \wedge (y = y_t)$

$sc_{\text{move}} = d((x', y'), (x'_t, y'_t)) < d((x, y), (x_t, y_t))$

$sc_{\text{h-move}} = |x' - x'_t| < |x - x_t|$
$a_{\text{h-move}} = \text{h-moveOneCell}$
$pre_{\text{h-moveOneCell}} = (x \neq x_t)$
$post_{\text{h-moveOneCell}} = |x' - x'_t| = |x - x_t| - 1$

Fig. 1 GDT for the goal of reaching a location on a grid (Example 11)

4.3 Operational semantics of GDTs

We now explain how GDTs are given a formal semantics. This semantics allows us to formally prove the correctness of proof obligations and generation of behaviour automata. More generally, it allows us to formally define how a GDT is intended to be executed, and to define each operator available.

We wish to emphasize that this semantics describe the intended behaviour of the agent, even if the proof obligations of the GDT are not verified. In this case, the satisfaction conditions of nodes may not be established by the execution, and the execution of some nodes may not terminate. This however allows to use our implementation by automata and, for instance, use model-checking techniques on unverified GDTs (see Section 6).

The operational semantics is given by rules in LTL. For an agent A, these rules are all evaluated in the same trace, corresponding to the lifetime of A. In particular, the LTL operator $\circ$ refers to the next instant relative to the internal clock of A. More precisely, for an instant ω, we define $\circ\omega$ (relative to A) to be the earliest instant after ω when A has performed exactly one atomic action or evaluated one formula of $\mathcal{L}$ or $\mathcal{L}'$. In particular, anything might have happened in the environment in the meantime (see our parallelism model in Section 3.3).

Our LTL rules are built on special Boolean variables. For simplicity, we say that at execution time, an agent *is executing* a node N in its GDT when it is executing the action or one of the children associated to N, or when it is deciding the satisfaction condition of N, etc.

Definition 9 (LTL atoms) Let N be a node in the GDT of an agent A, and let ω be an instant in the lifetime of A. Then $\omega \models init_N$ (resp. $\omega \models end_N$) if and only if A starts (resp. ends) executing N at instant ω, and $\omega \models in_N$ if and only if A starts, finishes, or is currently executing N.

Now if $\omega \models in_N$, write α_N for the latest world before or equal to ω and satisfying $init_N$. Then $\omega \models sc_N$ if and only if the couple of interpretations (α_N, ω') satisfies sc_N; if $\omega \not\models in_N$, the value of sc_N at it is not defined. Finally,

$\omega \models sat_N$ if and only if $\omega \models end_N \wedge sc_N$, and $\omega \models nonsat_N$ if and only if $\omega \models end_N \wedge \neg sc_N$.

Importantly, we view these atoms as signals sent between nodes in a GDT. For instance, we will consider sat_N as a signal sent by a child node to its father node in a GDT exactly when the corresponding goal is achieved. Thus we implictly assume that unless entailed otherwise by the rules, all atoms are always false. The only exceptions are in_N and sc_N, whose values are determined by whether N is being executed and whether its satisfaction condition is currently true, respectively.

In particular, with this assumption, and provided the operational semantics of operators respect the general rules given in Definition 10, we are guaranteed that the set of rules is consistent, since signals have only positive occurrences by construction.

Moreover, seeing LTL atoms as signals being sent between nodes makes it quite natural to envision the implementation of behaviours specified by GDTs using events. The implementation by automata (Section 6) is a particular type of this event-style programing.

Example 12 (LTL atoms) Assume an agent moving on a one-dimensional finite grid with an action for moving by one position to the left, and one for moving by one position to the right. Let N be a node in its GDT, modelling the goal of reaching the last but one square on the right, and assume the specified behaviour for achieving this goal consists of iterating moves to the right until failure, then moving once to the left. Write R for the iterated node, and L for the last one.

Assume a situation where the agent is three moves from the right border (denoted by $x = 3$), and executes the specified behaviour. Then the atoms true at each instant are as follows (atoms in_R and in_L are omitted).

Instant	**Node** N	**Node** R	**Node** L
ω_0 $(x = 3)$	$init_N, in_N$	$init_R$	
ω_1 $(x = 2)$	in_N	end_R, sc_R, sat_R	
ω_2 $(x = 2)$	in_N	$init_R$	
ω_3 $(x = 1)$	in_N, sc_N	end_R, sc_R, sat_R	
ω_4 $(x = 1)$	in_N, sc_N	$init_R$	
ω_5 $(x = 0)$	in_N	end_R, sc_R, sat_R	
ω_6 $(x = 0)$	in_N	$init_R$	
ω_7 $(x = 0)$	in_N	$end_R, nonsat_R$	
ω_8 $(x = 0)$	in_N		$init_L$
ω_9 $(x = 1)$	in_N, end_N, sc_N, sat_N		end_L, sc_L, sat_L

We wish to emphasize that at instants ω_3 and ω_4, the main goal is satisfied, but since the specified behaviour is to go on to the right (the agent does not even notice that its goal is satisfied), the end_N atom is not true, and thus the sat_N signal is not sent.

All nodes	For $1 \leq i < j \leq n, \Box(\neg in_{N_i} \vee \neg in_{N_j})$	$(G.1)$
	For $1 \leq i \leq n, \Box(in_{N_i} \rightarrow in_N)$	$(G.2)$
	$\Box(\neg init_N \vee \neg end_N)$	$(G.3)$
	$\Box((in_N \wedge \neg end_N) \rightarrow \circ\neg init_N)$	$(G.4)$
Leaf nodes	$\Box(init_N \rightarrow \circ end_N)$	$(G.5)$
L nodes	$\Box(init_N \rightarrow \circ(sc_N \rightarrow end_N))$	$(G.6)$
	$\Box(init_N \rightarrow \circ(\neg sc_N \rightarrow init_N^{NL}))$	$(G.7)$

Table 1 Operational semantics for GDTs (Definition 10)

We now give the operational semantics of GDTs, including rules common to all decomposition operators. These rules will serve as assumptions in proof obligations, but on the other hand, the decomposition operators and the designed GDT must obey them. For instance, a designer cannot use a node as a child of two different nodes nor introduce cycles in a GDT.

Definition 10 (well-formed GDT) A GDT is said to be *well-formed* if each of its nodes N satisfies the rules in Table 1, where $N_1, N_2, \ldots, N_n$ denote the children of N if it is an internal node.

Intuitively, the rules enforce the tree structure of a GDT (Rules (G.1), (G.2) and (G.4)), the fact that executing a node requires at least one step (Rule (G.3)), and that elementary goals (essentially, actions) terminate and determine the clock for the agent (Rule (G.5)). As for lazy nodes, the rules enforce that execution terminates immediately if the satisfaction condition is already true (Rule (G.6)), and otherwise is followed (at the next instant) by the execution of the NL version of the node ($init_N^{NL}$ is a shorthand for that).

4.4 Decomposition operators

We now describe the eight decomposition operators currently available. We concentrate on two of them so as to illustrate the model, and briefly present the other ones. We however want to emphasize that any decomposition operator can be introduced, as soon as it comes with an operational semantics respecting well-formedness of GDTs, with (validated) proof obligations and with a (validated) automatic transformation into behaviour automata.

Decomposition operators essentially specify which subgoals of a node will be executed, in which order and how many times, depending on the environment state and on the outcome of actions. Thus for each operator, we give the associated operational semantics. Moreover, we describe the schema for inferring the necessary satisfiability of the parent goal from the necessary satisfiability of its children.

As a general rule, we depict decompositions as trees, with the operator written on a link through children. Nodes are depicted into ellipses with an "L" attached if they are lazy, and double-circling if they are necessarily satisfiable.

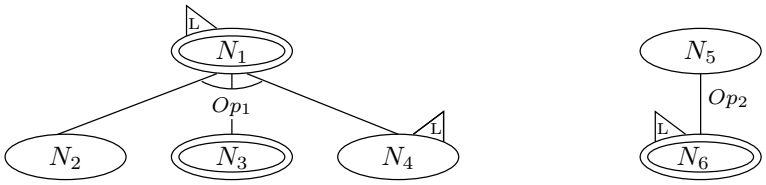


Fig. 2 Representations of decompositions (Example 13)

$SeqAnd$	$\Box(sat_{N_1} \rightarrow \circ init_{N_2})$	$(SA.1)$
	$\Box(nonsat_{N_1} \rightarrow \circ end_N)$	$(SA.2)$
	$\Box(sat_{N_2} \rightarrow (end_N \wedge sat_N))$	$(SA.3)$
	$\Box(nonsat_{N_2} \rightarrow \circ end_N)$	$(SA.4)$
NL	$\Box(init_N \rightarrow init_{N_1})$	$(SA.5)$

Table 2 Operational semantics for *SeqAnd* (Definition 11)

Example 13 (graphical representation of GDTs) Figure 2 gives two example decompositions. The one on the left uses a ternary operator Op_1, with parent node N_1 and children N_2 to N_4. The one on the right uses a unary operator. Nodes N_1, N_3, N_6 are NS while the others are not, and nodes N_1, N_4, N_6 are lazy, while the others are not.

The simplest operator, which we denote by *SeqAnd*, is a sequential "and". This operator specifies that in order to achieve the parent goal, both children must be executed in the specified order, and both must succeed. In particular, the parent goal can be soundly labelled NS if and only if both children are so.

The formal definition is as follows. The rules first enforce that execution begins immediately with the first child (Rule (SA.5)); the case when N is lazy is derived using general rules (G.6) and (G.7). Then the rules enforce that if the first child fails, then execution terminates at the next step, and otherwise goes to the second child. Importantly though, observe that failure of one child does not automatically trigger signal $nonsat_N$ (nonsatisfaction of the parent goal); indeed, it can be the case that the decomposition fails but that the parent goal is achieved for some other reason. Rules (SA.2) and (SA.4) leave a delay for evaluating that.

Notice that due to the parallelism model described in Section 3.3, the environment may be modified between the execution of two subgoals of an agent, because, for instance, another agent acts in the meantime.

Definition 11 (SeqAnd) *SeqAnd* is the binary decomposition operator defined by the rules in Table 2, where N denotes the parent node with $Children_N = (N_1, N_2)$.

It is easily seen that, under the interpretation that atoms become true only if enforced by a rule, this semantics is consistent with the general rules for operators (Table 1).

$Iter$	$\Box(end_{N_1} \rightarrow \circ(\neg sc_N \rightarrow init_{N_1}))$ $\Box(end_{N_1} \rightarrow \circ(sc_N \rightarrow sat_N))$	$(I.1)$ $(I.2)$
NL	$\Box(init_N \rightarrow init_{N_1})$	$(I.3)$

Table 3 Operational semantics for $Iter$ (Definition 12)

Example 14 (SeqAnd) A *SeqAnd* decomposition can be used for specifying the behaviour of a robot which must take a photograph of some location. Indeed, first (successfully) going to the location and then (successfully) taking a photograph achieves this goal. On the contrary, if one of this goals fails, then the parent goal is normally not satisfied.

As another example, the goal of computing $(x' = \sqrt{x+1})$ can be achieved by a decomposition of N into N_1 and N_2 via *SeqAnd*, where $sc_N = (x' = \sqrt{x+1})$, $sc_{N_1} = (x' = x+1)$, and $sc_{N_2} = (x' = \sqrt{x})$.

We now turn to operator $Iter$. This operator is a unary one, and allows to specify iterating behaviours. More precisely, the child must be executed as long as the satisfaction condition of the parent node is not true. Typically, executing the child node allows for a progression to the parent goal, as in Example 11. Nevertheless, modifications in the environment may also realize the parent goal, in which case the iteration process also stops.

Termination is enforced by proof obligations, which essentially use a (hand-designed) variant to prove that there is a finite progression towards the parent goal. As a result, a verified decomposition via $Iter$ always terminates and thus, the parent goal can always be soundly labelled NS, whatever the necessary satisfiability of its child.

Definition 12 (Iter) $Iter$ is the unary decomposition operator defined by the rules in Table 3, where N denotes the parent node and N_1 denotes the child node.

Observe that the rules impose that the agent evaluates the satisfaction condition of the parent node N before deciding to iterate further or to stop. Noticeably, this entails that N never ends at the same time as its child.

Example 15 (Iter) A decomposition with $Iter$ can typically be used for a robot whose task is to collect a certain weight of ore. To achieve this task, it can iteratively collect pieces of ore until the expected weight is reached.

As another example, if the value of a (nonnegative) integer x can only be decreased one by one, then the behaviour specified by an $Iter$ decomposition with satisfaction condition of the child node $sc_{N_1} = (x' = x - 1)$ achieves the goal $x = 0$.

We now describe the remaining, currently available operators more briefly.

Besides the *SeqAnd* operator, we have defined the *And* operator. The semantics is like for *SeqAnd*, but the agent nondeterministically chooses one of the children to execute first, and in case of success proceeds to the other one.

The family of "or" operators allows to specify several ways to achieve a goal. All of them are binary (but the generalization is straightforward). The semantics of a plain *Or* decomposition is as follows. When executing the parent goal, the agent nondeterministically chooses one of the children and executes it. If this execution succeeds, then the parent goal is achieved (provided the proof obligations have been verified). On the contrary, if the execution of the first child fails, then the agent executes the other child. The parent goal is achieved if and only if this second execution succeeds (again, if the obligations have been verified).

Apart from the plain *Or*, *SeqOr* is an available operator. The semantics is as for *Or*, but the children are ordered, so that execution always starts with the first one, and only in case of failure proceeds to the second one. Observe that a *SeqOr* decomposition is useless if the first child is NS, since then execution will never come to the second child.

Example 16 ("or" operators) Operator *SeqOr* is typically used when an agent has two strategies for achieving a goal, one of which has little cost but may fail, and the other has a greater cost but is guaranteed to succeed. Then it is worth trying first the cheap strategy and, in case it fails, using the other.

This operator is also useful when failure of the first subgoal helps (or allows) to achieve the second one.

Operators *SeqAnd* and *SeqOr* also come in a synchronized version. Recall that between execution of the first and second children of such a node, environment variables may change value so that, for instance, the satisfaction condition of the first subnode is not true any more when entering the second subnode of a *SeqAnd* decomposition. When this is not desirable, the agent may use special operator $SyncSeqAnd(V)$. This operator is similar to *SeqAnd*, but (environment) variables in V are locked between the execution of the first and second node: Other agents cannot modify their value. A similar construct is available for *SeqOr*, resulting in operator $SyncSeqOr(\cdot)$.

Importantly, adding synchronization over environment variables may raise interblocking problems in the system. We do not address this specific issue here, since it is widely handled in the literature. Consequently, if such operators are used, our proof system for GDTs assumes that interblocking cannot occur at execution time.

Finally, the binary $Case(Cond_1,Cond_2)$ operator is similar to a "switch" construct. The semantics is that the agent first decides which of $Cond_1$ or $Cond_2$ is true. If $Cond_1$ is true, then it executes the first child, and otherwise it executes the second one. It is required that always one of $Cond_1$ and $Cond_2$ is true. Both can be true at the same time, resulting in a nondeterministic choice of the child to execute.

5 Proofs of GDTs

In this section, we describe the process for verifying a GDT. The general idea is that once the GDT is specified, proof obligations can be automati-

cally generated from (essentially) the satisfaction conditions of nodes and the decomposition operators, using schemas.

Proof obligations are entailment relations between formulas in $\mathcal{L}$ (in fact, in $\mathcal{L}_{V \cup V^{tmp} \cup V'}$). The semantics is that if all the obligations generated from a GDT are true (entailment can be proven), then executing the GDT achieves the satisfaction condition of each NS node, and achieves either the satisfaction condition or the GPF of each other node.

Importantly, the proof process of GDTs is compositional. That is, proof obligations associated to a node N do not depend on nodes other than N and its children. This allows for verifying parts of a GDT independently from each other, and for substituting subtrees for others without verifying the whole GDT again.

In Subsections 5.2 to 5.4, we present the proof schemas for nonlazy nodes. The schemas for lazy nodes can be derived from these, as presented in Subsection 5.5.

5.1 Information automatically inferred

As explained in Section 4, the designer of a GDT gives a precondition and a trigerring context for the whole GDT, as well as a satisfaction condition for each node. Moreover, to each leaf an action is attached, which comes with a satisfaction condition and a GPF.

From this information, other information can be automatically inferred for each node in the GDT, namely a *context* in which each node is executed and a GPF for each node. Thus the designer does not need to specify this information. Yet it can be used for generating easier proof obligations, since contexts and GPFs will come as additional hypotheses.

Definition 13 (context) Let T be a GDT, and let N be a node in T. The *context* of N, written c_N, is any formula in $\mathcal{L}$ which is logically equivalent to the conjunction of all formulae $\varphi \in \mathcal{L}$ such that for all worlds ω satisfying $init_N$, ω satisfies φ.

Example 17 (context) Consider the GDT on Figure 1. The context of the node labelled "move" contains $(x \neq x_t \vee y \neq y_t)$. Indeed, for any world ω in which the agent starts executing it ($\omega \models init_{\text{move}}$), we can infer that the satisfaction condition of its parent node is false (otherwise the iteration would stop).

Observe that contexts are defined as formulae in $\mathcal{L}$, not in $\mathcal{L}'$. That is, they only refer to the state of the world when execution begins.

In general, it is not possible to compute the exact context of a node. Nevertheless, since contexts will serve as hypotheses in proof obligations, it is enough to compute a logically weaker formula. Thus, in general, an ideal proof obligation will be of the form $c_N \wedge \mathit{premices} \models \mathit{conclusion}$, but in practice the obligation computed will be $\varphi_N \wedge \mathit{premices} \models \mathit{conclusion}$, with $c_N \models \varphi_N$. Clearly, the latter is harder to verify, but if it is verified, then the former is

also true. Weakening may occur because inference rules cannot take every information into account, and because some hard computations may be involved, like computing projections.

As an illustration of the inference process, we now give the rules for inferring contexts in two particular cases. The proof that the rules for *SeqAnd* are valid can be found in the companion paper, and is similar in spirit to the proof that proof schemas are correct (see Section 5.3). Using the rules for all decomposition operators allows to infer a (weakened) context for each node in a GDT, proceeding top-down from the root node.

Proposition 2 (context, root node) *Let T be a GDT. Then c_{Root_T} entails pre_T and tc_T.*

Proof Obvious from the definition of preconditions and trigerring contexts for GDTs (Definition 8). □

Proposition 3 (SeqAnd NL, context) *Let T be a GDT, and let N be a node in T with $Op_N = SeqAnd$, $Children_N = (N_1, N_2)$, and $lz_N = NL$. Then $c_{N_1} \models c_N$ and $c_{N_2} \models ((T'(sc_{N_1}))_{ri})_r$.*

Stated otherwise, when starting to execute N_1, from Proposition 3 we know that the current world satisfies c_N. Intuitively, this is just because since N is NL, execution of N_1 begins exactly at the same time as that of N (Rule (SA.5)). Now when starting to execute N_2, we know that the current world satisfies $((T'(sc_{N_1}))_{ri})_r$. Intuitively, this is because N_2 is executed only when N_1 has succeeded, at the time instant just following this success. Thus at this instant, the satisfaction condition of N_1 is true as far as internal variables are concerned (environment variables may have changed value). We only retain what $(T'(sc_{N_1}))_{ri}$ entails about the current world, that is, $((T'(sc_{N_1}))_{ri})_r$.

Dual to contexts are GPFs (of internal nodes). Inference goes bottom-up in the GDT, starting from the GPFs of actions (at the leaf nodes).

Example 18 (contexts and GPFs) Consider two agents A and A_t which are geographically situated. Assume that there is an environment variable pos_t corresponding to the position of A_t, and that A has the following three internal variables: *pos* denotes its own position, pos_{est} denotes the position where it estimates that A_t is, and *around* is true if it estimates that A_t is not too far, and false otherwise.

Let N be a node in the GDT of A, corresponding to the goal of meeting A_t if it is not too far, that is, sc_N is $around \rightarrow (pos = pos_{est})$. Assume N is NL and is decomposed with $Op_N = SeqAnd$ and $Children_N = (L, R)$. Node L corresponds to the NL goal of locating A_t, modelled by $sc_L = around \wedge (pos_{est} = pos_t)$ and $gpf_L = \neg around \wedge (pos'_{est} = pos_{est})$. Finally, Node R corresponds to the lazy goal of reaching the estimated position of A_t (in case locating has been possible), modelled by $sc_R = (pos = pos_{est})$ and $gpf_R = (pos' = pos)$.

Assume the context of N is $c_N = \neg around \wedge d(pos, pos_t) < 10$, where d computes a distance. Then we can infer that the context of L is the same,

since N is NL. Observe that if N was lazy, then we could only infer that the context of L entails (contains) $\neg around$, since $around$ is an internal variable of A; we could not infer $d(pos, pos_t) < 10$, since the distance may change while A is evaluating sc_N.

Now we can also infer that $around$ is true in the context of R, since if R is executed, then L has succeeded and $around$ has not changed value since then (being internal). On the other hand, we cannot infer that $(pos_{est} = pos_t)$ is true, since A_t may have moved since execution of L ended.

Finally, since the parent node N fails exactly when one of L or R fails, we can infer that gpf_N entails $\varphi_L \vee \varphi_R$ with $\varphi_L = \neg around \wedge (pos'_{est} = pos_{est})$ and $\varphi_R = around \wedge (pos' = pos)$. Observe that we can infer $(pos' = pos)$ using the frame axiom for internal variables (Assumption 1).

5.2 Proof schemas for leaf nodes

Recall that a leaf node is a tuple $N = (name_N, a_N, sc_N, gpf_N, lz_N, nsat_N)$. In particular, a_N is an action, of the form $a = (name_a, pre_a, post_a, gpf_a)$.

For verifying that executing such a node indeed establishes sc_N in case the execution is successful, and gpf_N otherwise, we first have to prove that the postcondition (resp. GPF) of the action entails sc_N (resp. gpf_N). Now since the action is meant to be executed, we also have to verify that the context of the (leaf) node entails its precondition. This results in the following proof schemas (recall that we assume here that N is NL). Recall that $i_{\mathcal{E}}$ (resp. i_A) denotes the invariant of the environment (resp. of agent A).

$$i_{\mathcal{E}} \wedge i_A \wedge c_N \models pre_a \tag{1}$$

$$i_{\mathcal{E}} \wedge i_A \wedge c_N \wedge T'(post_a) \models T'(sc_N) \tag{2}$$

$$i_{\mathcal{E}} \wedge i_A \wedge c_N \wedge T'(gpf_a) \models T'(gpf_N) \tag{3}$$

Example 19 (proof of leaf nodes) Consider the goal of moving a heavy load to some location pos_t (assumed to be constant) on a 2D-grid. Assume the current position of the load is described by variable pos, and its weight is described by a variable w. Assume the agent has one (NS) action, named *push*, with precondition $w < 100$ and postcondition $d(pos', pos_t)) = \max(d(pos, pos_t) - (100 + w), 0)$, where d computes a distance (the lighter the load, the closer to the target location it can be pushed at one time).

Assume w can vary through time, but respects the (environment) invariant $i_{\mathcal{E}} = (w < 50)$. Let N be an NL, NS leaf node with context $c_N = d(pos, pos_t) < 45$ and action *push*. Then the second proof obligation generated is:

$$\begin{pmatrix} w < 50 \\ \wedge\ d(pos, pos_t)) < 45 \\ \wedge\ d(pos', pos_t) = \max d(pos, pos_t) - (100 + w), 0 \end{pmatrix} \models (pos' = pos_t)$$

These rules are valid in the sense that we can prove the following. We refer the reader to the companion paper for the proof, but the technique is similar to that used for *SeqAnd* (see Section 5.3).

Proposition 4 (NL leaf nodes) *Let N be an NL leaf node, and let a be the associated action. Then if obligations 1 and 2 (resp. 1 and 3) are proven, execution of N succeeds (resp. fails) when a succeeds (resp. fails).*

Finally, we have to show that leaf nodes respect the invariant and stable properties of the agent and the environment. As we show right after, this will be enough to guarantee that the whole GDT does so. The proof schemas are as follows.

$$H_{elem} \models At'(i_{\mathcal{E}}) \tag{4}$$
$$H_{elem} \models At'(i_A) \tag{5}$$
$$H_{elem} \wedge s_{\mathcal{E}} \models At'(s_{\mathcal{E}}) \tag{6}$$
$$H_{elem} \wedge s_A \models At'(s_A) \tag{7}$$

with $H_{elem} = i_{\mathcal{E}} \wedge i_A \wedge c_N \wedge (T'(post_a) \vee T'(gpf_a))$.

The obligations generated from the two last proof schemas have to be proven for each stable property $s_{\mathcal{E}}$ of the environment (resp. s_A of the agent).

Proposition 5 (environment) *Let N be a leaf node in the GDT of an agent A, and assume that the obligation generated from Schema (4) is proven. Then for all worlds α_N such that $\alpha_N \models init_N$, if α_N satisfies $i_{\mathcal{E}}$, then so does $\circ\alpha_N$. Similarly, for all stable properties $s_{\mathcal{E}} \in S_{\mathcal{E}}$, if the obligation generated from Schema (6) is proven, then for all worlds α_N such that $\alpha_N \models init_N$, if α_N satisfies $s_{\mathcal{E}}$, then so does $\circ\alpha_N$.*

Proposition 6 (agent) *Assume that for all elementary goals N in the GDT of an agent A the obligation generated from Schema (5) is proven. Then for all worlds ω in the trace of A, ω satisfies i_A. Similarly, for all stable properties $s_A \in S_A$, if the obligation generated from Schema (7) is proven for all nodes in the GDT of A, then for all worlds ω in its trace, if ω satisfies s_A, then so does $\circ\omega$.*

5.3 Proof schemas for SeqAnd

We now consider the proof obligations for *SeqAnd* decompositions. Let N be an internal node in a GDT with $Op_N = SeqAnd$ and $Children_N = (N_1, N_2)$. Assume moreover that N is nonlazy. Then the proof schema for the satisfaction condition of N is as follows.

$$i_{\mathcal{E}} \wedge \Sigma_{\mathcal{E}} \wedge i_A \wedge \Sigma_A \wedge c_N \wedge (T^{tmp}(sc_{N_1}))_{ri} \wedge T'_{tmp}(sc_{N_2}) \models T'(sc_N) \tag{8}$$

with $\Sigma_{\mathcal{E}} = \bigwedge_{s_{\mathcal{E}} \in S_{\mathcal{E}}}(s_{\mathcal{E}} \rightarrow At'(s_{\mathcal{E}}))$ and $\Sigma_A = \bigwedge_{s_A \in S_A}(s_A \rightarrow At'(s_A))$. Intuitively, this amounts to considering three instants during the execution of N: The starting (unprimed) one, the final (primed) one, and an intermediate one (superscripted with *tmp*). The starting and final instants correspond to the execution of the parent node N, and the intermediate instant is the one

when execution of N_2 starts. The existence of these instants follows from the operational semantics of *SeqAnd*, as is proven in Proposition 7.

Thus the proof schema amounts to show that the success of N (that is, truth of $T'(sc_N)$ between the unprimed and primed instants) can be inferred from:

- what is known to be true when it starts, that is, $i_{\mathcal{E}} \wedge i_A \wedge c_N$,
- the fact that stable properties are preserved, as expressed by $\Sigma_{\mathcal{E}}$ and Σ_A,
- the success of N_1, that is, truth of sc_{N_1} between the unprimed and intermediate instants,
- the success of N_2, that is, truth of sc_{N_2} between the intermediate and final instants.

Right projection of $T^{tmp}(sc_{N_1})$ onto internal variables is necessary because the intermediate instant is when N_2 starts, which is not simultaneous to the end of N_1, when sc_{N_1} is evaluated. As a consequence, environment variables can change value between the two instants.

Example 20 (proof for SeqAnd, continued) Consider again Example 18, and assume all invariants are true and there are no stable properties. Then the proof obligation generated for N is:

$$\begin{pmatrix} \neg around \wedge d(pos, pos_t) < 10 \\ \wedge\ around^{tmp} \\ \wedge\ (pos' = pos'_{est}) \end{pmatrix} \models around' \rightarrow (pos' = pos'_{est})$$

Observe that only $around^{tmp}$ is retained from $T^{tmp}(sc_L)$, since every information about environment variable pos_t disappears with right projection; intuitively, A_t may change position freely between the instant when A finishes executing Node L and that when it starts executing Node R.

Also observe that the obligation can be verified since $(pos' = pos'_{est})$ is a hypothesis (as established by execution of Node R). However, $around'$ could also come as a hypothesis, since variable *around* is not modified by Node R and thus, $(around' = around^{tmp})$ could be soundly added to $T^{tmp}(sc_R)$.

Again, these schemas are valid in the sense that if the obligations generated from them are verified, then executing the decomposition terminates, and achieves the parent goal.

Proposition 7 (SeqAnd NL, termination) *Let N be a node with $Op_N =$ SeqAnd, $Children_N = (N_1, N_2)$, and $lz_N =$ NL. Then an execution of N terminates as soon as the corresponding executions of N_1 and N_2 do.*

Proof Let α_N be a world satisfying $init_N$. We have to show that α_N satisfies $\diamond end_N$.

From Rule (SA.5), we have $\alpha_N \models init_{N_1}$. Since N_1 terminates, there is a world ω_{N_1} such that $\omega_{N_1} \succeq \alpha_N$ and $\omega_{N_1} \models end_{N_1}$. Then if $\omega_{N_1} \models nonsat_{N_1}$, from Rule (SA.2) we get that it satisfies $\circ end_N$, which concludes. Otherwise we have $\omega_{N_1} \models sat_{N_1}$ (given the definition of atoms, see Definition 9), and

from Rule (SA.1) we get that ω_{N_1} satisfies $\circ init_{N_2}$; as above we get a world $\omega_{N_2} \succeq \alpha_N$ and satisfying end_{N_2}. If it satisfies sat_{N_2}, then it satisfies end_N by Rule (SA.3), and otherwise it satisfies $\circ end_N$ by Rule (SA.4), which concludes in both cases. □

Proposition 8 (SeqAnd NL, correctness) *Let N be a node with $Op_N = SeqAnd$, $Children_N = (N_1, N_2)$, and $lz_N = NL$. Assume that proof obligation (8) is verified for N. Then an execution of N succeeds as soon as the corresponding executions of N_1 and N_2 terminate and succeed.*

Proof Let ω_N be a world satisfying end_N, and α_N be the latest world before ω_N and satisfying $init_N$. We have to show that (α_N, ω'_N) satisfies $T'(sc_N)$.

We have $\alpha_N \models init_{N_1}$ by Rule (SA.5). Let ω_{N_1} be the earliest world after α_N satisfying end_{N_1}, which exists since N_1 terminates. Since N_1 succeeds, we have that $(\alpha_N, \omega'_{N_1})$ satisfies $T'(sc_{N_1})$, and $\omega_{N_1} \models sat_{N_1}$. Now let α_{N_2} be $\circ\omega_{N_1}$. By Rule (SA.1), we have $\alpha_{N_2} \models init_{N_2}$. Like for N_1, there is an earliest world ω_{N_2} after α_{N_2} which satisfies end_{N_2} and sat_{N_2}. Thus $(\alpha_{N_2}, \omega'_{N_2})$ satisfies $T'(sc_{N_2})$.

Now it follows from the tree semantics of GDTs (Definition 10) that ω_{N_2} is exactly ω_N. Finally, summing up and translating world ω_{N_1} to the intermediate instant, we have:

$$(\alpha_N, \omega^{tmp}_{N_1}) \models T^{tmp}(sc_{N_1}) \tag{9}$$

$$(\alpha^{tmp}_{N_2}, \omega'_{N_2}) \models T'_{tmp}(sc_{N_2}) \tag{10}$$

Now from (9) we get the stronger $(\alpha_N, \omega^{tmp}_{N_1}) \models (T^{tmp}(sc_{N_1}))_{ri}$. From the frame axiom for internal variables (Assumption 1) and from the fact that α_{N_2} is defined to be $\circ\omega_{N_1}$, we conclude $(\alpha_N, \alpha^{tmp}_{N_2}) \models (T^{tmp}(sc_{N_1}))_{ri}$. Finally, $(\alpha_N, \alpha^{tmp}_{N_2}, \omega'_{N_2})$ satisfies $(T^{tmp}(sc_{N_1}))_{ri} \wedge T'_{tmp}(sc_{N_2})$. Now by definition, α_N satisfies $i_{\mathcal{E}}$, i_A and c_N, $(\alpha_N, \omega'_{N_2})$ satisfies $\bigwedge_{s_{\mathcal{E}} \in S_{\mathcal{E}}}(s_{\mathcal{E}} \rightarrow At'(s_{\mathcal{E}}))$ and $\bigwedge_{s_A \in S_A}(s_A \rightarrow At'(s_A))$, and consequently, from the proof obligation, the above triple satisfies $T'(sc_N)$. Since this latter formula does not contain any variable of the form v^{tmp}, we finally have $(\alpha_N, \omega'_{N_2}) \models T'(sc_N)$, as desired since $\omega_N = \omega_{N_2}$. □

As a consequence, N can be labelled NS as soon as both N_1 and N_2 are, as said in Section 4.

Obviously, the reasoning is similar to that for leaf nodes concerning GPFs instead of satisfaction conditions. However, preservation of invariant and stable properties does not have to be verified here, since it can be inferred from preservation at leaf nodes (Propositions 5 and 6).

5.4 Proof schemas for Iter

In this section, let N be a node with $Op_N = Iter$ and $Children_N = (N_1)$. Clearly, the main role of the proof schema here is to ensure termination, since

success is guaranteed by the operational semantics of $Iter$, which merely say "try again until you succeed".

As is quite standard, so as to prove termination a *variant* is needed. In a general setting, a variant is a variable whose value decreases at each iteration of a loop and whose domain is a well-founded structure (i.e., a structure admitting no infinite decreasing sequence). As a consequence, the loop necessarily terminates. Observe that the variant needs not be part of the variables resulting from modelling the environment or from agentifying the problem. It can be added as a variable during the proof process. Nevertheless, so that its value is guaranteedly affected by the agent only, we require that it is an internal variable of the agent. We also restrict the definition of a well-founded structure a little, by requiring that all decreasing sequence have the same lower bound.

Definition 14 (variant) Let N be a node with $Op_N = Iter$ in the GDT of an agent A. Then a *variant* for N is a tuple $(v, <_v, v_0)$, where $v \in V_i(A)$ and $<_v$ is a total order on the values taken by v, such that every decreasing sequence of these values is lower-bounded by the value v_0.

Like for $SeqAnd$, the proof schema for $Iter$ (NL case) considers the execution of N at three distinct worlds: The initial one, the final one, and an intermediate one which corresponds to the last time when execution of N_1 starts. We nevertheless have to consider several cases, depending on whether we consider the first iteration and whether the last execution of N_1 is successful. In particular, when we consider iterations other than the first one, we know that N has just failed (otherwise the operational semantics of $Iter$ require the iteration to stop).

Thus, first of all, in all cases we know that the following hypotheses hold:

$$H_{NL} = i_{\mathcal{E}} \wedge \bigwedge_{s_{\mathcal{E}} \in S_{\mathcal{E}}} (s_{\mathcal{E}} \rightarrow At'(s_{\mathcal{E}})) \wedge i_A \wedge \bigwedge_{s_A \in S_A} (s_A \rightarrow At'(s_A)) \wedge c_N$$

Now when considering the first iteration, the following hypothesis holds, the precise disjunct depending on whether this first execution of N_1 has succeeded or failed:

$$H_1 = (T'(sc_{N_1}))_{ri} \vee (T'(gpf_{N_1}))_{ri}$$

Right projection is used because according to Rule (I.2), N ends at the instant after N_1 ends. Finally, when considering the other iterations, the following hypothesis holds, the precise disjunct depending on whether the last iteration of N_1 has succeeded or not. Indeed, let the intermediate world represent the instant when this last iteration has begun. Then since iteration went on, we know that the parent goal was not satisfied at this world.

$$H_2 = T^{tmp}(\neg sc_N) \wedge ((T'_{tmp}(sc_{N_1}))_{ri} \vee (T'_{tmp}(gpf_{N_1}))_{ri})$$

Then the proof schema is as follows (NL case). We first have to prove that when the variant reaches its lower bound, then the root goal is satisfied. This

guarantees that the iteration has succeeded if the subnode cannot be iterated further (because this would make the variant decrease lower than v_0). Observe that the iteration could also succeed earlier. Then we have to prove that the variant decreases at each iteration (may the subnode succeed or not). This guarantees that it will eventually reach its lower bound and thus that the iteration will stop (because of the first proof obligation).

$$H_{NL} \wedge (H_1 \vee H_2) \wedge (v' = v_0) \models T'(sc_N) \tag{11}$$
$$H_{NL} \wedge H_1 \wedge (v' \neq v_0) \models v' <_v v \tag{12}$$
$$H_{NL} \wedge H_2 \wedge (v' \neq v_0) \models v' <_v v^{tmp} \tag{13}$$

Example 21 (proof for Iter) Consider again Example 11. For the sake of this example, assume moreover that the root node *reach* is NL, with context $c_{\text{reach}} = (x \neq x_t) \vee (y \neq y_t)$, and that x_t and y_t are constant. Finally, assume that all invariants are true and that there are no stable properties.

We write $d(pos', pos_t)$ as a shorthand for $d((x', y'), (x_t, y_t))$ and $|pos' - pos_t|$ as a shorthand for $|x' - x_t| + |y' - y_t|$, and similarly for pos, pos^{tmp}. Then a convenient variant for the root node *reach* is the value $v = |pos - pos_t|$, with $<_{variant}$ being the natural order on integers and $v_0 = 0$.

Recall that Node *move* is NS, thus $gpf_{\text{move}} = \bot$). Then we have:

$$\begin{aligned} H_{NL} &= (x \neq x_t) \vee (y \neq y_t) \\ H_1 &= d(pos', pos_t) < d(pos, pos_t) \\ H_2 &= ((x^{tmp} \neq x_t) \vee (y^{tmp} \neq y_t)) \wedge d(pos', pos_t) < d(pos^{tmp}, pos_t) \end{aligned}$$

We get the following proof obligations:

$$\begin{aligned} H_{NL} \wedge (H_1 \vee H_2) \wedge (|pos' - pos_t| = 0) &\models (x = x_t) \wedge (y = y_t) \\ H_{NL} \wedge H_1 \wedge (|pos' - pos_t| \neq 0) &\models |pos' - pos_t| <_v |pos - pos_t| \\ H_{NL} \wedge H_2 \wedge (|pos' - pos_t| \neq 0) &\models |pos' - pos_t| <_v |pos^{tmp} - pos_t| \end{aligned}$$

Observe that these proof obligations can be verified only depending on the definition of the distance function d.

Recall that N has to be NS, so GPFs are not relevant. For more details we refer the reader to the companion paper.

5.5 Proof schemas for lazy nodes

In this section, let N with $lz_N = L$. We write N^{NL} for the node equal to N except for $lz_N = NL$. Recall that the satisfaction condition of a lazy goal is a formula in $\mathcal{L}$ (that is, with no primed variable).

From the schemas given above, one can get the proof obligations for N^{NL}, provided a context c_N^{NL} is given. We claim that $\neg sc_N \wedge (c_N)_i$, that is, the nonsatisfaction of N together with the projection of its context onto internal variables, is a convenient context. By that, we mean that if the obligations for

N^{NL}, as generated with $\neg sc_N \wedge (c_N)_i$ as c_N^{NL}, are proven, then executing N indeed establishes sc_N in case of success.

The intuition is simply that when executing a lazy node, if the satisfaction condition does not initially hold, then execution is exactly as in the nonlazy case, but it starts one instant later. Thus the (initial) context maybe is not true anymore as concerns environment variables (see the example of lighting a candle up with a match in Example 9).

The case of GPFs is a bit more involved. Indeed, contrary to satisfaction conditions, the GPF of a lazy node may be a formula in $\mathcal{L}'$, that is, relate the initial instant of the execution to the final one. Now consider N and its nonlazy counterpart N^{NL}, as above. The proof obligations for GPFs will show that the GPF of N is established starting from the initial instant for N^{NL}, but we want it established relative to the initial instant for N. Thus we have to show the proof obligations for N^{NL} as above, but we moreover have to show that $(gpf_N)_{\ell i}$ is enough for establishing gpf_N.

Proposition 9 (proof schemas for lazy nodes) *Let N be a lazy node, and let N^{NL} be the same node except for $lz_{N^{NL}} = NL$. Define c_N^{NL} to be $\neg sc_N \wedge (c_N)_i$. Then if all proof obligations generated for N^{NL} with this context, and the additional obligation $(T'(gpf_N))_{\ell i} \models T'(gpf_N)$, are proven, N terminates and succeeds (resp. establishes its GPF) when its satisfaction condition is initially true or the corresponding execution of N^{NL} terminates and succeeds (resp. establishes it GPF).*

Observe that in case GPFs are automatically inferred, the proposition can be used to infer $(T'(gpf_N))_{\ell i}$ for N, where gpf_N is inferred as if N were nonlazy.

Example 22 (proof schemas for lazy nodes) As concerns lazy leaf nodes, the proof obligations become the following.

$$\begin{aligned} i_{\mathcal{E}} \wedge i_A \wedge \neg sc_N \wedge (c_N)_i &\models pre_a \\ i_{\mathcal{E}} \wedge i_A \wedge \neg sc_N \wedge (c_N)_i \wedge T'(post_a) &\models T'(sc_N) \\ i_{\mathcal{E}} \wedge i_A \wedge \neg sc_N \wedge (c_N)_i \wedge T'(gpf_a) &\models T'(gpf_N) \\ (T'(gpf_a))_{\ell i} &\models T'(gpf_a) \end{aligned}$$

To conclude, as concerns preservation of invariant and stable properties by leaf nodes, the proof schemas are the same as those given in Section 5.2, but with $H_{elem} = i_{\mathcal{E}} \wedge i_A \wedge \neg sc_N \wedge (c_N)_i \wedge (T'(post_a) \vee T'(gpf_a))$.

6 Implementation of behaviours

In this section, we describe how the behaviour of an agent, as specified by a GDT, can be automatically implemented. The implementation is as an automaton, which can be easily generated in (or translated into) any programing language of interest.

The semantics of this implementation is that executing the automaton generated is (provably) the same as executing the behaviour specified by the GDT, as defined by the operational semantics. The automaton are anyway essentially a translation of this operational semantics, though there are some subtleties for the implementation of, e.g., synchronization.

Importantly, executing the automaton is the same as executing the GDT, even if the proof obligations generated from the latter have not been verified. As a consequence, techniques alternative to theorem proving can be used for verifiyng the correctness of the behaviour specified, like first generating the automaton and then model-checking it (the only problem might be unverified termination for operators like $Iter$).

6.1 Behaviour automata

We propose an implementation of GDTs by *behaviour automata*. Such automata are an adaptation of string-to-string automata. The automaton implementing a node N in a GDT is essentially obtained from automata implementing the children of N, combined according to the decomposition operator of N and modified according to its laziness and necessary satisfiability. The behaviour automaton for a GDT is that for its root node.

A behaviour automaton for a node N essentially works as follows. First of all, it receives signals of the form $init_N$ from an automaton for the parent of N in the GDT. It also receives signals of the form sat_{N_i} or $nonsat_{N_i}$ from each of its children N_i. Such receptions are modelled as symbols read by the automaton, thus labelling its transitions. Other symbols which can be read by an automaton are its own satisfaction condition (sc_N or $\neg sc_N$), and symbols specific to each operator, like conditions for a $Case(Cond_1, Cond_2)$ decomposition.

Dually, a behaviour automaton for N sends signals to its parent and children nodes ($sat_N/nonsat_N$ and $init_{N_i}$, respectively). Finally, it prescribes the execution of actions. Signals sent and actions executed are modelled by symbols written by the automaton[3]. Other symbols model the action of saving the value of variables so as to be able to evaluate the satisfaction condition of the node later on, and locking/unlocking environment variables for synchronized operators.

Clearly, the implementation or translation of such an automaton in a programing language is straightforward (provided, obviously, that the implementation of actions is).

Definition 15 (behaviour automaton) Let T be a GDT for an agent A, and let N be a node in T. A *behaviour automaton* for N is a tuple $(Q, \Sigma, \Gamma, q^0, \delta)$, where:

[3] In a previous work [SF06], we represented the different actions as performed inside the states of the automaton. Here, like in the model of Mealy transducers [HU79], we have chosen to represent them on transitions. However, the difference between the two representations is only technical.

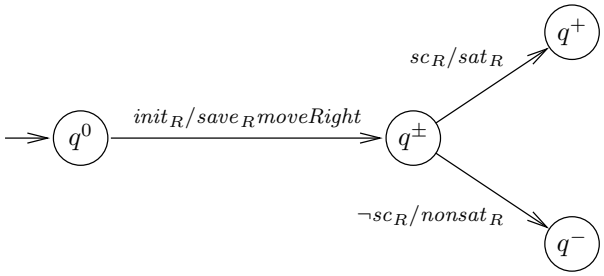


Fig. 3 Automaton for moving by one position to the right (Example 23)

- Q is a set of states, including two special states written q^+ and q^-,
- Σ is an input alphabet, including $init_N$, sc_N, $\neg sc_N$, as well as sat_{N_i} and $nonsat_{N_i}$ for all nodes $N_i \in Children_N$,
- Γ is an output alphabet, including sat_N and $nonsat_N$, $init_{N_i}$ for all nodes $N_i \in Children_N$, $name_a$ for all actions $a \in Actions_A$, as well as $save_N$ and $lock_V, unlock_V$ for all sets of variables $V \subseteq V_\mathcal{E}$,
- $q^0 \in Q$ is an initial state,
- $\delta \subseteq Q \times \Sigma \times (\Gamma \cup \{\varepsilon\}) * \times Q$ is the transition relation.

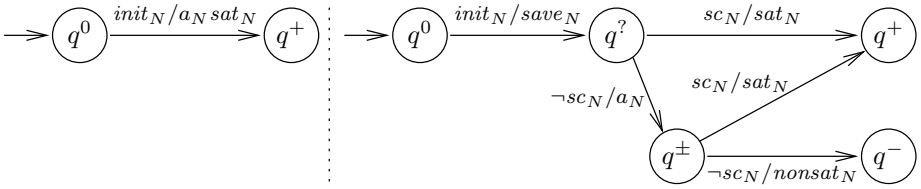
There can be only one transition from q^0, on reading $init_N$. Dually, the last symbol written on transiting to q^+ (resp. q^-) is restricted to be sat_N (resp. $nonsat_N$).

Example 23 (behaviour automaton) Consider again Example 12, and the node R corresponding to the goal of moving by one position to the right (which fails if the agent is already at the rightmost position). Let sc_R be $(x' = x - 1)$ and gpf_R be $(x' = x)$. Finally, let *moveRight* be the action of moving by one position to the right.

The behaviour automaton for Node R is depicted on Figure 3. The automaton is initially in state q^0. When execution comes to Node R, the signal $init_R$ is received (from the parent node). Then the agent saves the value of all relevant variables (here that of x) and performs the action *moveRight*. This leads it to state $q^\pm$. Then the agent evaluates sc_R, that is, decides whether the current value of x equal the one saved minus 1. If yes, then atom sc_R is true and the automaton moves to state q^+, sending the signal sat_R (to its parent node). Otherwise, atom sc_R is false (symbol $\neg sc_R$ is read) and the automaton moves to state q^-, sending the signal $nonsat_R$ (to its parent node).

Remark that behaviour automata give an implementation of the behaviour of an agent, but do not claim to define all elements of this behaviour. For instance, they do not specify how messages are waited for, how locking of variables is achieved, etc. Note however that this can be specified by the implementation of specific actions available to the agent.

Moreover, for specific purposes (in particular, new decomposition operators), new symbols can be added to the input or ouput alphabets. For instance,



(a) NL and NS node (b) L and NNS node

Fig. 4 Behaviour automata for leaf nodes

the nondeterministic choice of the first child to be executed in an *Or* decomposition can be modelled by an input signal.

6.2 Automata for leaf nodes

Recall that a leaf node N is a tuple $(name_N, a_N, sc_N, gpf_N, lz_N, nsat_N)$. To such a node a behaviour automaton is associated, depending on its laziness and necessary satisfiability.

The simplest case is that of an NL and NS node. The corresponding automaton only prescribes, on receiving the $init_N$ signal, to save the values of variables and execute the action, and then send the sat_N signal. The other cases (L and/or NNS leaf nodes) are a bit more complex, and can be derived from the former by using the L and NNS patterns (see Section 6.4).

All four combinations give rise to a behaviour automaton. The simplest and the most complex ones are shown on Figure 4. Each transition is labelled with the symbol σ read and the string $\boldsymbol{\gamma}$ written in the form $\sigma/\boldsymbol{\gamma}$. We formally define only the NL and NS case, since the other cases can be derived with patterns.

Definition 16 (automaton for leaf node) Let N be a leaf node in a GDT with $lz_N = NL$ and $nsat_N = NS$. The *behaviour automaton for* N is the behaviour automaton with $Q = \{q^0, q^+\}$ and $\delta = \{(q^0, init_N, a_N sat_N, q^+)\}$.

6.3 Composition patterns for internal nodes

The automaton for a node is built from automata for its children in the GDT. This is done using composition patterns, that is, patterns describing how the children automata are combined according to the decomposition operator. As for leaf nodes, L nodes are built using patterns, so we describe the case when the parent node is NL.

We first consider the case of a node N with decomposition operator $Op_N = SeqAnd$, and we write $Children_N = (N_1, N_2)$. As the operational semantics states (Rule (SA.5)), when execution of N begins, then simultaneously does

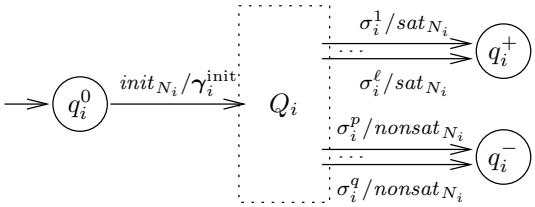


Fig. 5 General notation for the automaton of N_i

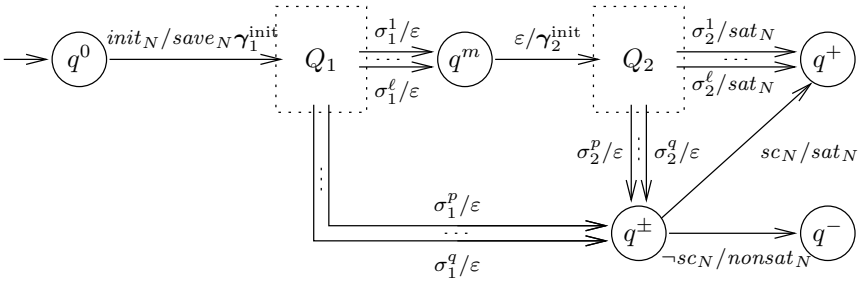


Fig. 6 Composition pattern for *SeqAnd* (NL)

execution of N_1. Thus the composition pattern prescribes to identify the initial state of the automaton for N to that of the automaton for N_1. On reception of signal $init_N$, the automaton for N is defined to behave exactly as that for N_1 on reception of $init_{N_1}$.

Dually, when execution of N_1 ends, two cases are to consider. If this execution is successful, then the automaton for N_1 is in state q^+. As prescribed by Rule (SA.1), we identify this state with the initial one for the automaton of N_2 (and the state is not a "q^+" state for N). On the contrary, if execution of N_1 fails, then as prescribed by Rule (SA.2) we replace the "q^-" state of N_2 with an intermediate state $q^\pm$ to which we attach an NNS pattern (see Section 6.4).

Finally, the final states q^+ and q^- of the automaton for N_2 are identified with the q^+ state of the automaton for N and with the $q^\pm$ state above, respectively. In the end, assume the automata for N_1 and N_2 are as on Figure 5. Then for N, we get the pattern given on Figure 6 and formally defined as follows.

Definition 17 (pattern for SeqAnd) Let N be a node with $Op_N = SeqAnd$, $lz_N = NL$, and $Children_N = (N_1, N_2)$. Write Q_i for the set of states of the behaviour automaton for N_i, and similarly for q_i^0, q_i^+, etc. Then the *behaviour automaton for* N is built as follows. The set of states Q is defined to be $(Q_1 \setminus \{q_1^0, q_1^+, q_1^-\}) \cup (Q_2 \setminus \{q_2^0, q_2^+, q_2^-\}) \cup \{q^0, q^m, q^+, q^-, q^\pm\}$. The transition relation δ contains the following transitions:

$(q^0, init_N, save_N\gamma_1^{\text{init}}, q_1)$ where $(q_1^0, init_{N_1}, \gamma_1^{\text{init}}, q_1) \in \delta_1$

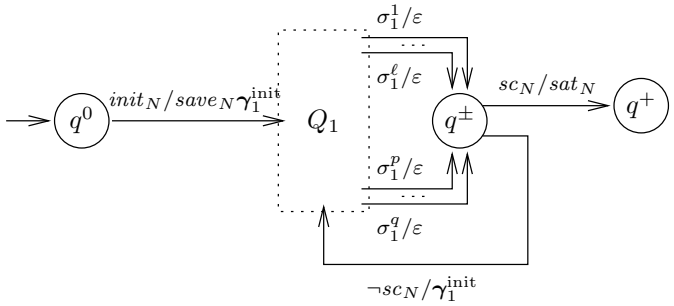


Fig. 7 Composition pattern for *Iter* (NL)

$$
\begin{aligned}
(q_1, \sigma_1^i, \varepsilon, q^\pm) &\text{ for all } (q_1, \sigma_1^i, nonsat_{N_1}, q_1^-) \in \delta_1\\
(q_1, \sigma_1^i, \varepsilon, q^m) &\text{ for all } (q_1, \sigma_1^i, sat_{N_1}, q_1^+) \in \delta_1\\
(q^m, \varepsilon, \boldsymbol{\gamma}_2^{\text{init}}, q_2) &\text{ where } (q_2^0, init_{N_2}, \boldsymbol{\gamma}_2^{\text{init}}, q_2) \in \delta_2\\
(q_2, \sigma_2^i, sat_N, q^+) &\text{ for all } (q_2, \sigma_2^i, sat_{N_2}, q_2^+) \in \delta_2\\
(q_2, \sigma_2^i, \varepsilon, q^\pm) &\text{ for all } (q_2, \sigma_2^i, nonsat_{N_2}, q_2^-) \in \delta_2\\
(q^\pm, sc_N, sat_N, q^+)&\\
(q^\pm, \neg sc_N, nonsat_N, q^-)&\\
(q_1, \sigma_1, \boldsymbol{\gamma}_1, q_1') &\text{ for all } (q_1, \sigma_1, \boldsymbol{\gamma}_1, q_1') \in \delta_1, q_1, q_1' \neq q_1^0, q_1^+, q_1^-\\
(q_2, \sigma_2, \boldsymbol{\gamma}_2, q_2') &\text{ for all } (q_2, \sigma_2, \boldsymbol{\gamma}_2, q_2') \in \delta_2, q_2, q_2' \neq q_2^0, q_2^+, q_2^-
\end{aligned}
$$

The fact that this pattern is correct follows straightforwardly from the fact that it is built according to the rules which define the operational semantics of *SeqAnd*.

The pattern for *Iter* decompositions follows the same lines. We do not detail it here, and only give it on Figure 7. Observe that nodes with operator *Iter* can always be labelled NS, thus there is no (reachable) "q^-" state in the resulting automaton.

6.4 Patterns for L and NNS nodes

When a node N is lazy, the behaviour automaton for N can be built as in the NL case, using an additional modification with a specific pattern.

This pattern implements the operational semantics for L nodes. The former (NL) initial state is replaced with one corresponding to the test of the satisfaction condition (denoted by $q^?$). If the satisfaction condition is true, then transition takes place directly to q^+, and otherwise the transition is the initial one from the nonlazy automaton. The new initial state has a single transition to $q^?$. Moreover, observe that no saving of values is needed, since the satisfaction condition has to be a formula in $\mathcal{L}$ (that is, a nonprimed one), and thus it can be evaluated with the current values of variables only.

The pattern is depicted on Figure 8 (with the notation of Definition 18).

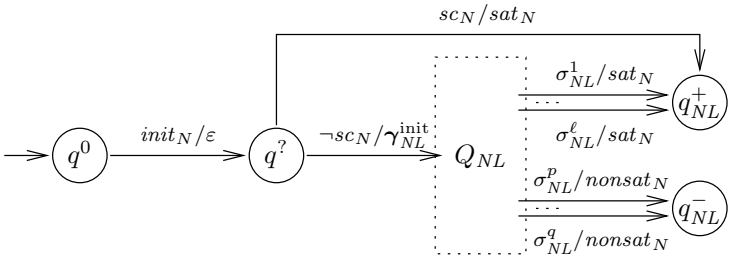


Fig. 8 Patterns for L nodes

Definition 18 (lazy pattern) Let N be a lazy node, and let N_{NL} be the node equal to N except for $lz_{N_{NL}} = NL$. Write Q_{NL} for the set of states of the behaviour automaton for N_{NL}, and similarly for its other components. Then the *behaviour automaton for* N is built as follows. The set of states Q is defined to be $(Q_{NL} \setminus \{q^0_{NL}\}) \cup \{q^0, q^?\}$, states q^+ and q^- are defined to be q^+_{NL} and q^-_{NL}, respectively, and the transition relation δ contains the following transitions:

$$
\begin{aligned}
&(q^0, init_N, \varepsilon, q^?)\\
&(q^?, sc_N, sat_N, q^+)\\
&(q^?, \neg sc_N, \boldsymbol{\gamma}^{\text{init}}_{NL}, q_{NL}) \text{ where } (q^0_{NL}, init_{N_{NL}}, \boldsymbol{\gamma}^{\text{init}}_{NL}, q_{NL}) \in \delta_{NL}\\
&(q_{NL}, \sigma_{NL}, \boldsymbol{\gamma}_{NL}, q'_{NL}) \text{ for all } (q_{NL}, \sigma_{NL}, \boldsymbol{\gamma}_{NL}, q'_{NL}) \in \delta_{NL}, q_{NL} \in Q_{NL} \setminus \{q^0_{NL}\}
\end{aligned}
$$

Similarly, the automata for NNS nodes can be derived from those for NS nodes. The pattern is used for composition patterns (see Section 6.3), but it can also be used for making any NS node deliberately NNS.

The need for a specific pattern, as opposed to simply reachability of state q^-, comes from the fact that a node whose decomposition fails may nevertheless succeed, due to other conditions like evolution of environment variables. As already discussed in Section 4.4, this is reflected by, e.g., Rules (SA.2) and (SA.4), according to which the end_N signal is sent only at the instant after the decomposition fails.

The NNS pattern thus replaces state q^- of the automaton by an intermediate state $q^\pm$, which corresponds to the test of the satisfaction condition. Naturally, there are two transitions from this state, one to q^+_{NS} and one to (the new) q^-. The pattern is depicted on Figure 9 (with obvious notation).

7 Further types of nodes

In this section, we present two natural extensions of the method as presented in Sections 4 to 6. These extensions allow to take into account types of nodes different than nodes associated with a decomposition or an action.

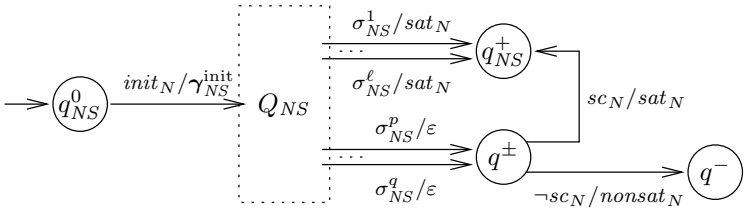


Fig. 9 Patterns for NNS nodes

7.1 Parameterized GDTs

Quite naturally, when specifying behaviours with GDTs, there is the need for using the same part of a tree several times. For instance, if the agent models a mobile robot, then it is quite likely that its behaviour will consists in going from one place to another at different moments and in different contexts. It is then quite natural to use the same GDT for achieving this goal, without reverifying its correctness.

As specified by the general operational semantics of GDTs (see Section 4.3), several nodes cannot share a child, for this would result in a DAG instead of a tree (thus preventing, e.g., from efficiently propagating contexts and GPFs). Moreover, it can be the case that the designer wants to reuse the same subGDT in different GDTs for agents of the same kind.

For these reasons, we described here how a (sub)GDT can be used in different contexts. The idea is to consider a so-called *parameterized GDT* which is verified once and for all, independently of the GDT in which it is used, and then to use instances of this pattern with the only requirement to verify that the usage is correct. Thus a paremeterized GDT is much like an action coming with a proof that it achieves the claimed goal (postcondition) in a given context (precondition).

The technical definition is as follows. A parameterized GDT is exactly like a GDT, except that its variables are not instantiated; they are just placeholders, some for internal variables of the agent using the GDT and some for environment variables. The GDT is defined relative to a set of actions which are used at the leaf nodes.

Definition 19 (parameterized GDT) Let *Actions* be a set of actions. A *parameterized GDT* is a 5-tuple $T = (P^P, Root_T, c_T, i^P, S^P)$ where P^P is a set of variables, $Root_T$ is the root node of a GDT over variables P^P and using only actions from *Actions*, c_T, i^P are formulas in $\mathcal{L}_{P^P}$, and S^P is a finite set of formulas in $\mathcal{L}_{P^P}$. Moreover, P^P is partitioned into $P^P_i \cup P^P_{\mathcal{E}}$.[4]

Observe that in particular, $Root_T$ defines a satisfaction condition sc_T, a necessary satisfiability $nsat_T$ and, if needed, a GPF gpf_T for T.

[4] An invariant over placeholders for internal variables and one over placeholders for environment variables could be defined instead of only one, and similarly for stable properties, but we chose to keep the presentation simple here.

A parameterized GDT comes with all the necessary information for verifying that it is correct. Namely, proof obligations can be generated treating the variables in P_i^P as being internal variables (of some imaginary agent) and variables in $P_{\mathcal{E}}^P$ as being environment variables. The context of the root node as well as invariants and stable properties used in proof schemas are those coming with the paramaterized GDT. If all proof obligations generated are verified, then the parameterized GDT itself will be called *verified*.

Intuitively, if a parameterized GDT is verified, this means that it will achieve the claimed satisfaction condition or GPF each time it is used in a GDT with the variables in P_i^P (resp. $P_{\mathcal{E}}^P$) replaced (instantiated) by internal (resp. environment) variables, and the context, invariant and stable properties are respected.

Definition 20 (use of a parameterized GDT) Let A be an agent in an environment $\mathcal{E}$. A *use* of a parameterized GDT $T = (P^P, Root_T, c_T, i^P, S^P)$ is a node N in T of the form

$$(name_N, T, \mathcal{I}_{P_i^P}, \mathcal{I}_{P_{\mathcal{E}}^P}, sc_N, gpf_N, lz_N, nsat_N)$$

where $name_N, sc_N, gpf_N, lz_N, nsat_N$ are as in Definition 6, $\mathcal{I}_{P_i^P}$ is a mapping from P_i^P to the set of internal variables $V_i(A)$ of A, and $\mathcal{I}_{P_{\mathcal{E}}^P}$ is a mapping from $P_{\mathcal{E}}^P$ to $V_{\mathcal{E}}(A)$.[5]

We finally give proof schemas for uses of paramterized GDTs. These schemas are similar to those for leaf nodes (see Section 5.2), but one also has to check preservation of the invariants and stable properties of the agent and of the parameterized GDT. Then the intuition is that if the corresponding obligations are verified, then the parameterized GDT is correctly used.

So as to simplify notation, we write, e.g., c_T, but this denotes the formula obtained from c_T by replacing every parameter in P^P with its image under $\mathcal{I}_{P_i^P}$ or $\mathcal{I}_{P_{\mathcal{E}}^P}$. Moreover, so as to keep the presentation simple, we do not give the obligations referring to invariant and stable properties.

$$i_{\mathcal{E}} \wedge i_A \wedge c_N \models c_T \tag{14}$$

$$i_{\mathcal{E}} \wedge i_A \wedge c_N \wedge T'(sc_{Root_T}) \models T'(sc_N) \tag{15}$$

$$i_{\mathcal{E}} \wedge i_A \wedge c_N \wedge T'(gpf_{Root_T}) \models T'(gpf_N) \tag{16}$$

Proposition 10 (use of PGDTs) *Let N be a use of a parameterized GDT T. Assume moreover that T is verified, and $lz_N = NL$. Then if obligations 14–16 are proven and all invariants and stable properties are correctly preserved, execution of N terminates (resp. succeeds, fails) as soon as the corresponding execution of T does.*

As a consequence, N can be labelled NS as soon as $Root_T$ is NS. The lazy case is handled as in Section 5.5.

[5] Note that, since assumptions over environment variables are less restrictive than over internal ones, a placeholder for an environment variable could also be instanciated with an internal one.

7.2 External goals

In a multi-agent setting, one can distinguish two types of interaction between agent. The first one is implicit, through modification of the environment, which we model by unpredictible evolution of environment variables as seen by an agent. Interactions of the second type are explicit, for instance, agent helping each other to achieve part of their goals.

External goals are a first step towards handling this second kind of interaction when designing an agent. We do not claim to handle such interactions exhaustively, but we present some preliminary work towards representing and verifying them.

An external goal N in the GDT of an agent A is one which A cannot achieve, for instance because it depends on variables that it does not control. Thus a satisfaction condition is associated to N, but instead of an action or decomposition, other agents are expected to establish it true.

Thus external goals are a way to express dependences between (collaborative) agents. They can be seen as a specialization of TAEMS "nonlocal tasks" [VHL01]. Observe however that there is no contracting with our notion. As a special case, an external goal as the first operand of a *SeqAnd* decomposition can be seen as a specification of an "enables" interrelationship in TAEMS. For a more detailed comparison with TAEMS, we refer the reader to [SMF06].

In this preliminary report, we make the following simplifying assumptions:

- exactly one goal N of one agent is specified as the one satisfying N for A,
- N is necessarily satisfiable,
- there are no cyclic dependencies between agents (where A is said to depend on A^E if A has an external goal satisfied by a goal of A^E).

Moreover, implicitly we assume that all agents have a behaviour specified by a GDT, and that this behaviour is known when external goals are specified (see Definition 5). This makes sense when the whole MAS is designed at a time, but clearly leaves space for further improvements.

Then an external goal is defined as follows. The notion is similar to that of a leaf node, but another goal of another agent, instead of an action, is attached.

Definition 21 (external goal) Let A be an agent in an environment $\mathcal{E}$. An *external goal* in the GDT T of A is a node N in T of the form

$$(name_N, A^E, N^E, sc_N, lz_N)$$

where $name_N, sc_N, lz_N$ are as in Definition 6, A^E is another agent in $\mathcal{E}$, and N^E is an NS node in the GDT of A^E.

Example 24 (external goal) Let us consider two agents in a university where classrooms are locked for security reasons. The first agent, the *teacher*, has a course to provide in a classroom. The second agent is the *mace bearer*: it has a pass key to open classrooms. Then a subgoal of the teacher agent is

to enter into the right classroom. This subgoal is decomposed thanks to a *SeqAnd* operator into two subgoals: Unlocking the door, then opening it. So, the first subgoal is an external goal that will be achieved by the mace bearer, the second agent.

Even with our simplifying assumptions, external goals make the proof process much more complex. So as to prove that executing an external goal N will indeed establish it satisfaction condition, one has to prove as usual that execution terminates and succeeds.

In case execution terminates, we propose the following proof schema for the verification whether the satisfaction condition of N is established (in the nonlazy case, referring to Section 5.5 for the lazy case). Recall that the goal of A^E is assumed to be necessarily satisfiable.

$$i_{\mathcal{E}} \wedge i_A \wedge i_{A^E} \wedge c_N \wedge T'_{tmp}(sc_{N^E}) \models T'(sc_N) \tag{17}$$

Importantly, observe that the starting instant for execution of N^E is not the same as that for N, as reflected by the "tmp" instant in the obligation.[6]

Now as concerns termination, we propose to prove that whenever execution of N starts, at whatever point in its GDT A^E is, A^E will eventually reach N^E. Clearly, this is sufficient (but not necessary) for establishing that A does not wait forever, provided that A^E is already executing its GDT (see Proposition 12).

We first define, given a node N_0 in the GDT of an agent A, what other nodes in its GDT execution is guaranteed to eventually reach starting from execution of N_0 (provided the GDT is verified, and in particular that execution of all nodes in between terminates). We will then require to show that N^E is such a node for A^E, starting from whatever N_0 in its GDT (the one A^E is executing when A comes to its external goal N).

Note that our characterization is syntactic (and thus, easy to compute) but only partial.

Definition 22 (eventually reached) Let T be a GDT for an agent A, and let N_0 be a node in T. The nodes *eventually reached* from N_0 in T are N_0 and all nodes obtained by recursive application of the following rules, where N is required to be NL and eventually reached in all cases:

- if Op_N is one of $\{SeqAnd, SeqOr, SyncSeqAnd(\cdot), SyncSeqOr(\cdot)\}$ and $Children_N = (N_1, N_2)$, then N_1 is eventually reached from N,
- if Op_N is $Iter$ and $Children_N = (N_1)$, then N_1 is eventually reached from N,
- if Op_N is either $SeqAnd$ or $SyncSeqAnd(\cdot)$, $Children_N = (N_1, N_2)$ and $nsat_{N_1} = NS$, then N_2 is eventually reached from N.

The following proposition is quite straightforward given the operational semantics of operators.

[6] Equivalently, we could consider only the projection of T'(sc_{N^E}) onto the right.

Proposition 11 (eventually reached) *Let T be a verified GDT for an agent A, and let N_0, N be two nodes in T such that N is eventually reached from N_0. Then for all world ω in the trace of agent A^E, if $\omega \models init_{N_0}$, then $\omega \models \diamond init_N$.*

We finally get the following proof schema :

$$\forall N_0^E \in T^E,\ N^E \text{ is eventually reached from } N_0^E \tag{18}$$

So as to get rid of the case when A^E is not executing its GDT, one can simply prove that this cannot occur, that is, that its precondition and triggering context are always true. Clearly, this is stronger than needed, but guarantees the correctness of the proof. Moreover, if one can prove by any means that A^E is executing its GDT when A comes to its external goal, then this is also sufficient for the proof to be correct. For formal details see Proposition 12.

Proposition 12 (external goals) *Let a MAS containing agents $A_1, \ldots, A_n$. If the three following conditions are verified:*

- *there are no cyclic dependencies between the agents;*
- *each agent A_i^E supposed to help another agent A_j to satisfy one of its external goals is always executing its GDT;*
- *obligations 17 and 18 are proven for each relevant external goal of each agent.*

Then the external goals of each agent always terminate and succeed.

8 Related work

In this section, we survey work related to ours. We first review work in formal verification (but not necessarily concerned with agents and MASs) and in MAS design (but not necessarily concerned with validation). We then compare our model to the closest ones in the literature, namely models for MAS design which integrate a test or proof system.

Note that we do not propose a classification of the methods described, but that we simply analyze them under points of view relevant to the comparison.

8.1 General-purpose verification methods

Methods for validating software have been extensively studied mainly for critical systems, that is, application fields where security is necessary (e.g., in the transportation domain). Two main methods exist: Test and proof. Since our model is oriented towards theorem-proving, we will not discuss (general-purpose) test methods here.

Methods and models for proof rely on a formal specification written in a formal language. Moreover, most methods are supported by tools allowing to perform the proofs themselves.

Numerous such methods have been proposed, but they can be grouped into a limited number of categories: Abstract Data Types [GH78], Process Algebras (LOTOS [FL94], π-calculus [MPW92]), Dynamic Distributed models (Unity [CM88], Back's Action Systems [Bac93]), and Model-Oriented methods (Z [Spi87],VDM [Jon90], B [Abr96], TLA+ [Lam96]).

However, all these methods are too general to be directly used for specifying and verifying MASs, which are massively distributed and dynamic systems.

8.2 Models and methods dedicated to MASs

The first aim of models for agents and MASs was to help developers to design MASs. The most famous one is certainly the BDI model [RG95], though there are numerous other ones [SDB02].

The BDI architecture has become a standard model, and most recent works on multiagent models are based on it. For instance, the BOID architecture adds the notion of *obligation* to the belief, desire and intention notions of BDI agents [BDH$^+$01]. However, the BDI architecture and its extensions lack a strong structuration and a method.

Two early formal methods dedicated to MASs are MetateM [Fis94] and Desire [BvET97]. Nevertheless, neither allows to specify properties that the system must guarantee.

On the contrary, methods relying on the role notion introduce an abstract notion that helps to perform the requirements engineering task. This kind of methods allows to reason at first at the system level, and not directly at the agent level. For instance, Wooldridge *et al.* developed the Gaïa method [WJK00]. In Gaïa, a MAS is specified twice: in terms of its behaviour (through liveness properties) and in terms of its invariant properties. Thus the bases for proving MASs are parts of this method. Neverthess, using directly Gaïa to prove MASs or agent behaviours is not possible, in particular because properties are assigned to roles, not to agents, and the method does not provide any formal semantics to role composition. So, adding a role proof mechanism to Gaïa could be easily performed, but it would not provide an agent verification mechanism [7].

Another family of methods, closest to ours, is the family of goal-oriented methods. Most of these methods are at the agent level rather than at the system level, and so the agentification task must be performed first [8]. Nevertheless, two exceptions can be found: Moise [HSB02] and PASSI [CP02]. For instance, with PASSI, agent types are produced by grouping *use cases* identified during the analysis step. There are however no guidelines for grouping use cases not associating them to agents.

Now among the goal-oriented methods at the agent level, we can distinguish declarative and procedural models. Methods with a declarative model

[7] For these reasons, the method is essentially dedicated, as their authors claim, to systems with "a one-to-one mapping between roles and agents types".

[8] This is also the case of our approach.

allow to formally specify goals and to reason about them. This is mainly the case of the Goal method [dBHvdHM00] or of the work by van Riemsdijk *et al.* [vRDDM04]. An advantage of such models is that they often introduce the notion of a goal decomposition into subgoals, allowing a top-down, progressive specification mechanism. Among all these methods, TAEMS [VHL01] uses the task and subtask notions (similar to our goals and subgoals) to simulate MASs and to check at runtime if an implementation satisfies a theoretic model of tasks dependencies. We refer the reader to [SMF06] for more details.

Procedural models aim at producing agent descriptions which are easier to implement. For that reason, most procedural models for MASs are associated to languages dedicated to agent programming, such as 3APL [DdBDM03] and AgentSpeak [Rao96]. These languages give a formal model of the behaviour of the system, making a proof *theoretically* possible, since it is possible to directly prove the correctness of programs. However, there are three limits to such approaches. First of all, proving a program is much more difficult than proving a specification. Then, proving a programimplies means than the program has already been developed, and thus the verification step occurs very late in the design process. Finally, in a language such as AgentSpeak, an important part of the agent behaviour is not directly expressed in AgentSpeak. Thus it is impossible to perform complete proofs.

To overcome some of these limits, Winikoff *et al.* [WPHT03] propose a goal model allowing to express both declarative and procedural views of goals: The declarative view is specified by a satisfaction and a failure condition for each goal, and the procedural view is given by a plan. However, the semantics of actions is not specified, which weakens the expressiveness of the procedural view.

For more details about the numerous models and methods for MAS development, we refer the reader to [JSW98,IGG99,SDB02,DW03].

8.3 Comparison with the closest approaches

As evoked in Section 8.1, there are essentially two ways to prove the correctness of a specification, namely model checking and theorem proving.

Recently there have been many works on model-checking agents (see for instance [BFVW03,BFPW03,ALW04,RL04,KLP04,KP04]. However, all these works share the same limit: The complexity is reduced, but is still here, making verification of very complex systems difficult if not unfeasible. Among these works, the one by Alechina *et al.* [ALW04] is interesting because it allows to take time explicitly into account in the proof. However, proofs are limited to propositional logic. Similarly, Raimondi and Lomuscio [RL04] clearly explain the difficulties of theorem proving and the advantage of using Binary Decision Diagrams, but the logical world which they propose is rather limited (more limited than Linear Temporal Logic, which, they claim, is not rich enough). Finally, Kacprzak and Penczek [KP04] propose an interesting unbounded model checking method for alternating-time temporal logic, an exten-

sion of the branching time logic CTL where operations can be parameterised by sets of agents. However, once again, proofs are limited to propositional logic.

As opposed to model checking, there are not a lot of works which deal with using theorem proving for verifying MASs, as we propose to do. The main reason for that is that theorem provers cannot perform all the proofs of a system whose properties are expressed with predicates (essentially because first-order logic is undecidable). However, many theorem provers can now prove very complex systems automatically, like PVS [OSR92] or krt (the prover of the atelier B) [Abr96]. These provers can also use model checking when useful.

A recent and very interesting work is the one by Bracciali *et al.* [BED+06] about PROSOCS agents. A detailed comparison of our work with PROSOCS can be found in [MSSZ07]. The main drawback of PROSOCS is that it relies on propositional logic.

Other models exist, in particular relying on logic programming. Actually, these models look well suited to perform verification by theorem proving. Among them, one can find CaseLP [MMZ97] and DCaseLP [BBG+05]. However, proofs are absent from the CaseLP model. Since the extension to DCaseLP presented in [BBG+05], proofs have been integrated, but they only verify the implementation of interaction protocols.

Congolog [GLL00] and CASL [SLL02] are also two interesting languages, relying on the situation calculus. Moreover, they both allow to perform proofs. However, these proofs only concern the sequence of actions, not their semantics.

The Goal method [dBHvdHM00] allows to formally define goals of an agent. Goals are described in propositional logic, limiting the expressiveness of the language, in comparison with systems allowing a specification in predicate logic. The method also defines a proof mechanism allowing to prove temporal properties expressed in a Unity-like language [CM88]. However, the essential temporal property which allows to express the liveness of a program, namely $leads - to$, cannot be verified by the proof system. This strongly limits the usage of the method. Moreover, the weak fairness assumption made by Goal on the action selection of each agent also makes the MAS difficult to implement.

To conclude, Dastani *et al.* have proposed the 2APL language [Das08]. Though there are some similarities with our approach, like a formal transition-system semantics, a multi-agent model, and the notion of a dynamic environment, their approach does not embed a proof system. Moreover, 2APL is not compositional, which makes the system more monolithic from the validation point of view. On the other hand, 2APL allows to formalize communication actions, which we cannot do so far with GDTs.

9 Discussion and future work

We have presented the model of goal decomposition trees, which allows to specify the behaviour of agents in an intuitive, goal-based fashion, using one's

favourite logic. The model also allows to formally verify that these behaviours achieve the desired goals, by verifying automatically generated proof obligations, and/or to generate automata which (provably) execute this behaviour.

An important point of our model is that it allows to take nondeterminism and dynamicity into account. First, the environment is both dynamic and nondeterministic, in the sense that the value of an environment variable may change at any moment in time, independently from the current state of the world and nonnecessarily according to rules known to the agents (or to any rules at all). The only restriction is that environment variables cannot change value while an agent is executing an atomic action. Second, the actions which our method can model may also be nondeterministic. For instance, an action with satisfaction condition $v' > 3$ may assign an integer value to v nondeterministically. More importantly, actions may fail to achieve their postcondition.

We believe that our study of how these aspects of the environment and of actions impact the proof system for our model is interesting *per se*. In particular, projection of satisfaction conditions and contexts for lazy or NNS goals, and the need for GPFs prove very important.

As for the specified behaviours, they essentially correspond to policies. Our model allows these policies themselves to be nondeterministic, using operators like *And* and *Or*. For instance, parallism with nondeterministic scheduling can be modelled by an *Iter* decomposition, with one child itself decomposed into the *Or* of two threads. The resulting execution will be a concurrent execution of the two threads.

A perspective for further work in this direction is to take stochasticity into account, with features such as actions with a given probability to fail or variables with a given probability not to change values. The resulting policies would come themselves with probabilities to succeed or fail. We believe that our model could be extended in this direction, with proof obligations allowing for computing (or verifying) the probability of a decomposition to succeed or fail, etc. In the same vein, numerical utilities could be attached to goals and satisfaction conditions, and the expected utilitiy of a decomposition computed from the subgoals using tools like proof obligations.

Another perspective is to take more interactions between agents into account. The model is currently agent-centered, though it allows to take into account interactions via the environment. We have presented a first step towards handling more complex interactions, through the use of external goals. Our current work focuses on extending the modelling capabilities of our method for such dependencies between agents. Handling communication explicitly is also an interesting perspective in this direction. Our ultimate goal in this direction is to be able to verify behaviours of the whole system (as defined by the interleaved execution of individual behaviours), may these goals be specified explicitly, or correspond to observed emerging behaviours. Clearly, our event-based operational semantics serves these purposes, since the behaviour of an agent is not seen as a sequence of tests and actions, but as behaviours fired by trigerring events.

Finally, we aim at extending the GDT editor and code generator that we have already developed and which are presented in [MSSZ07]. One of the extensions will be to automatically generate proof obligations attached to a GDT in order to make them be proven by a theorem prover.

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.

[Bac93] R.J.R. Back. Atomicity refinement in a refinement calculus framework. Technical Report 141, Åbo Akademi, 1993.

[BBG+05] M. Baldoni, C. Baroglio, I. Gungui, A. Martelli, M. Martelli, V. Mascardi nd V. Patti, and C. Schifanella. Reasoning About Agents' Interaction Protocols Inside DCaseLP. In *Declarative Agent Languages and Technologies II*, volume 3476 of *LNCS*, pages 112–131. Springer, 2005.

[BDH+01] J. Broersen, M. Dastani, Z. Huang, J. Hulstijn, and L. Van der Torre. The BOID architecture: Conflicts between beliefs, obligations, intentions and desires". In *Proceedings of the Fifth International Conference on Autonomous Agents (AA2001)*, pages 9–16. ACM Press, 2001.

[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.

[BFPW03] R.H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model-checking AgentSpeak. In *AAMAS-03*, Melbourne, Australia, 2003.

[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.

[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.

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

[CP02] M. Cossentino and C. Potts. A CASE tool supported methodology for the design of multi-agent systems. In *SERP*, 2002.

[Das08] Mehdi Dastani. 2APL: a practical agent programming language. *Journal of Autonomous Agents and Multi-Agent Systems*, 16:214–248, 2008.

[dBHvdHM00] F.S. de Boer, K.V. Hindriks, W. van der Hoek, and J.-J.Ch. Meyer. Agent programming with declarative goals. In *7th International Workshop on Intelligent Agents. Agent Theories Architectures and Language*, pages 228–243, 2000.

[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.

[DW03] K.H. Dam and M. Winikoff. Comparing agent-oriented methodologies. In *Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems*, 2003.

[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 International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Heidelberg, Germany, 1994.

[FL94] M. Faci and L. Logrippo. Specifying Features and Analysing Their Interactions in a LOTOS Environment. In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, 1994.

[GH78] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[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.

[HBdHM99] K.V. Hindriks, F.S. De Boer, W. Van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

[HSB02] J.F. Hubner, J.S. Sichman, and O. Boissier. Spécification structurelle, fonctionnelle et déontique d'organisations dans les SMA. In *Journees Francophones Intelligence Artificielle et Systemes Multi-Agents (JFIADSM'02)*. Hermes, 2002.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusets, USA, 1979.

[IGG99] C. Iglesias, M. Garrijo, and J. Gonzalez. A survey of agent-oriented methodologies. In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.

[Jon90] C.B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 1990.

[JSW98] N.R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agents Systems*, 1(1):275–306, 1998.

[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.

[KP04] M. Kacprzak and W. Penczek. Unbounded model checking for alternating-time temporal logic. In *Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, 2004.

[Lam96] L. Lamport. The syntax and semantics of tla$^+$. Part 1: Definitions and Modules, June 1996.

[LLM03] J. Lang, P. Liberatore, and P. Marquis. Propositional independence — formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.

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

[MMZ97] M. Martelli, V. Mascardi, and F. Zini. CaseLP: a Complex Application Specification Environment base on Logic Programming. In *Proc. of ICLP'97 workshop on Logc Programming and Mult-Agents*, pages 35–50, 1997.

[MPW92] R. Milner, J. Parrow, and D. Wlaker. A calculus of mobile processes. *Journal of Information and Computation*, 100, 1992.

[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.

[MSZ08] 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/~zanutti/data/articles/msz08companion.pdf`.

[OSR92] S. Owre, N. Shankar, and J. Rushby. Pvs: A prototype verification system. In *CADE 11*, 1992.

[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.

[RG95] A. Rao and M. Georgeff. BDI agents from theory to practice. In *Technical note 56*. AAII, 1995.

[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.

[SDB02] Arsne Sabas, Sylvain Delisle, and Mourad Badri. A comparative analysis of multiagent system development methodologies: Towards a unified approach. In Robert Trappl, editor, *Cybernetics and Systems*, pages 599–604. Austrian Society for Cybernetics Studies, 2002.

[SF06] G. Simon and M. Flouret. Implementing validated agents behaviours with automata base on goal decomposition trees. In *Agent Oriented Software Engineering VI*, volume 3950 of *LNCS*, pages 124–138. Springer Verlag, 2006.

[SLL02] S. Shapiro, Y. Lesprance, and H. J. Levesque. The Cognitive Agents Specification Language and Verification Environment for Multiagent Systems. In *AAMAS*, pages 19–26. ACM Press, 2002.

[SM90] L.M. Stephens and M.B. Merx. The effect of agent control strategy on the performance of a DAI pursuit problem. In *10th International Workshop on Distributed Artificial Intelligence*, Bandera, Texas, 1990.

[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.

[Spi87] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1987.

[VHL01] 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.

[vRDDM04] 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.

[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.

[WPHT03] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *8th International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, 2003.