

Model-Checking Agent Refinement

Lăcrămioara Aștefănoaei
astefano@cwi.nl

Frank S. de Boer
F.S.de.Boer@cwi.nl

CWI, Amsterdam, The Netherlands

ABSTRACT

We present a proof-technique for reducing the nondeterminism of abstract agent specifications in a BDI framework by means of refinement. We implement the operational semantics of agent specifications in rewrite systems such that we can automatically check if refinement between (fair) executions of agents holds.

Categories and Subject Descriptors: I.6.5 Model Development: Modeling methodologies

General Terms: Languages, Theory, Verification.

Keywords: BDI, Non-determinism, Refinement, Rewriting, Simulation.

1. INTRODUCTION

In this paper we focus on *refinement* as reducing the nondeterminism of high-level agent specification languages. Such specifications are inspired by UNITY [5], a classical design methodology which emphasizes the principles:

- “specify little in early stages of design” and
- “program design at early stages should not be based on considerations of control flow”.

We place ourselves in the framework of BDI models [4]. As already a standard notion, an agent is defined in terms of *beliefs, desires, intentions*. Beliefs and desires (goals) usually represent the mental state of an agent, and intentions denote the deliberation phase of an agent (often concretizing the choice of executing a *plan*).

Our objective is to model-check refinement between different levels of abstraction of BDI agent-oriented specifications. The approach we take can be described in two steps: we first fix the levels of abstraction as being two particular languages, BUnity and BUPL (Belief Update programming Language); we then prove refinement by constructing a simulation relation (on the mental states of BUnity and BUPL agents) and by checking for the absence of deadlocks.

We introduce BUnity as an adaptation of the UNITY language to the BDI paradigm. It is meant to represent an agent in the first stage of design. One only needs to specify initial beliefs and actions (*what* an agent can do). We make the observation that “specify little” implies nondeterminis-

Cite as: Model-Checking Agent Refinement, Lăcrămioara Aștefănoaei and Frank S. de Boer, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 705-712.

Copyright © 2008, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

tic executions of BUnity agents (actions may be executed in any arbitrary order, for example).

On the other hand, BUPL enriches BUnity constructions with the notions of *plans* and *repair rules*. These are meant to refine the early stage nondeterminism by specifying *how* and *when* actions are executed. We have chosen BUPL as the representation of agents in the last stage of design.

We give operational semantics to our languages which we model into rewrite systems. In this way, we can prototype BUnity and BUPL agents in Maude [7], a rewriting logic software. We use the LTL model-checker [10] which comes with Maude in order to decide whether a BUPL agent refines a BUnity agent. Additionally, given that our languages allow nondeterministic choices, we explicitly model fairness as LTL formulas, and we check whether refinement holds under fairness assumptions.

We exploit the particular nature of nondeterminism in the BUnity language such that we can efficiently implement the simulation relation. This is possible because the operational semantics of BUnity is encoded in deterministic transition systems: though an agent makes arbitrary decisions regarding which action to execute, the mental state reflecting the effect of the chosen action is uniquely determined.

Our contribution consists of introducing a general framework for modeling, and not programming agent languages. We stress the importance of prototyping before implementing complex agent platforms. It is a quick method for proving that the semantics fulfills the initial requirements. In this setup, we emphasize simple but expressive agent languages by introducing BUnity and BUPL, which are inspired by the already standard GOAL [8] and 3APL [11] languages.

Related Works Considering verification techniques for multi-agent systems, there are already some notable achievements: [2] focuses on model-checking AgentSpeak systems, [3] proposes Temporal Trace Language for analyzing dynamics between agents, [14] refers to verifying deontic interpreted systems. However, we are not aware of existing proof-methods for model-checking refinement between abstraction levels of agent-oriented specifications.

As for prototyping executable semantics, there are already results ([17]) promoting Maude. The initiative of modeling agents in Maude is taken in [16] and this is, in fact, the starting point for our implementation.

Outline The paper is structured as follows: §2 presents the theoretical results and §3 details the syntax and the semantics of BUnity and BUPL languages. Implementation issues are discussed in §4 and §5 concludes the paper.

2. REFINEMENT AND SIMULATION

We consider labeled transition systems (LTSs) as tuples $(\Sigma, s_0, Act, \rightarrow)$, where Σ is a finite set of states, s_0 is an initial state, Act is a set of actions (labels), and \rightarrow describes all possible transitions. We denote by τ a special action called *silent* action. $Act - \{\tau\}$ is the set of *visible* actions.

We write $s \xrightarrow{a} s'$ when $(s, a, s') \in \rightarrow$, meaning “ s may become s' by performing an action labeled a ”. It also means that transition a is enabled on s . We say that a transition system is *deterministic* when for any state s , and for any action a , $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ implies $s' = s''$. We call $s \xrightarrow{\tau} s'$ an idling transition and we abbreviate it by $s \rightarrow s'$. The “weak” arrow \Rightarrow denotes the reflexive and transitive closure of \rightarrow , and \xRightarrow{a} stands for $\Rightarrow \xrightarrow{a} \Rightarrow$.

A *computation* in a transition system is defined to be a sequence of the form $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} s_2 \dots$, where $l_i \in Act, i \in \mathbb{N}$. It can be either finite (when there is no possible transition from the last state), or infinite.

For a computation σ , the corresponding *trace*, $tr(\sigma)$, is the sequence of visible actions (a word defined on $(Act - \{\tau\})^*$). The set of all traces of a system S (the traces corresponding to all computations starting with the initial state of the system) is denoted by $Tr(S)$.

The trace set defines the externally observable *behavior* of the system. Our focus on visible actions (and not states) is motivated by the fact that, in studying simulation, we are interested in *what we see* and not *how the agent thinks*. Take the case of a robot: one simulates his physical actions, lifting or dropping a block, for example, and not the mental states of the robot.

In nondeterministic systems that abstract from scheduling policies, some traces are improbable to occur in real computations. In this sense, the operational semantics (given by transition rules) is too general in practice: if the actions an agent can execute are always enabled, it should not be the case that the agent always chooses the same action.

We want to cast aside such traces when modeling the systems. Moreover, we want a declarative, and not imperative solution. Our option is to follow the approach from [12]: we may constrain the traces by adding *fairness* conditions, modeled as LTL properties. Fairness is there expressed either as a weak, or as a strong constraint:

DEFINITION 2.1 (JUSTICE [12]). *A trace is just (weakly fair) with respect to a transition a if it is not the case that a is continually enabled beyond some position, but taken only a finite number of times.*

DEFINITION 2.2 (COMPASSION [12]). *A trace is compassionate (strongly fair) with respect to a transition a if it is not the case that a is infinitely often enabled beyond some position, but taken only a finite number of times.*

We will refer to these definitions when we detail our agent languages and the implementation. However, in this section, we consider the general case of constraints defined as predicates on computations. We say a constraint holds in an LTS if, and only if, it holds for any computation starting in the initial state. We denote by S_p an LTS with a constraint p , and we refer to $Tr(S_p)$ as the restricted set of traces: $Tr(S_p) = \{tr \mid tr = tr(\sigma) \wedge \sigma \models p\}$.

Reducing nondeterminism is *refinement*, usually defined in terms of trace inclusion. Being that we can specify con-

straints on computations, we consider *refinement* in terms of restricted trace inclusion.

DEFINITION 2.3 (REFINEMENT). *Let I_p, S_q be LTS with constraints. I_p refines S_q ($I_p \leq S_q$) iff $Tr(I_p) \subseteq Tr(S_q)$.*

Proving refinement by definition is not practically feasible: the set of traces may be infinite. Instead, refinement is proved by simulation, which has the advantage of locality of search: one looks for checks at the immediate (successor) transitions that can take place. Since we are interested in simulating only visible actions, we refer to weak simulation.

DEFINITION 2.4 (WEAK SIMULATION). *Let I, S be LTS with Σ, Σ' as sets of states, and let R be a relation, $R \subseteq \Sigma \times \Sigma'$. R is called a weak simulation if, whenever sRs' , if $s \xrightarrow{a} t$, then there exists t' such that $s' \xRightarrow{a} t'$ and tRt' .*

DEFINITION 2.5. *Let I, S be LTS with s_0, s'_0 as initial states. S weakly simulates I ($I \lesssim S$) if there exists a weak simulation R such that $s_0Rs'_0$.*

PROPOSITION 2.6 (SOUNDNESS). *Given the LTS I and S , we have that $I \lesssim S \Rightarrow I \leq S$.*

In general, simulation is not complete. Take, for example, the classical example described in Figure 1.

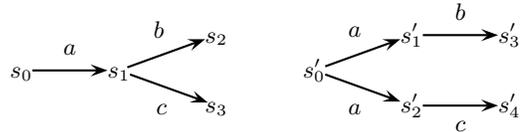


Figure 1: Refinement but not simulation

However, simulation is complete when the simulating system is deterministic:

PROPOSITION 2.7. *Given an LTS I and a deterministic LTS S , we have that $I \leq S \Rightarrow I \lesssim S$.*

COROLLARY 2.8. *$I \leq S \Rightarrow I \lesssim S'$, where S' is the deterministic system corresponding to S .*

The proofs are easy if one considers that two deterministic systems with the same traces simulate each other and that it is always possible to transform nondeterministic systems into deterministic ones by a method called “determinization” [15]. Nevertheless, “determinization” is computationally hard ($2^{O(n \log n)}$ in the number of states [15]).

The technique for proving refinement is to construct a *left synchronized product*, where the reachable states are pairs in a simulation relation. The absence of a *deadlock* pair implies that simulation holds, hence refinement.

DEFINITION 2.9 (LEFT SYNCHRONIZED PRODUCT). *The left synchronized product of two transition systems, $I = (\Sigma, s_0, Act, \rightarrow_1)$ and $S = (\Sigma', s'_0, Act, \rightarrow_2)$, is defined as:*

$$I \otimes S = (\Sigma \times \Sigma', (s_0, s'_0), Act, \rightarrow)$$

where if $s \xrightarrow{a}_1 t$ then $((s, s') \xrightarrow{a} (t, t') \text{ iff } s' \xrightarrow{a}_2 t')$.

REMARK 2.1. *We call it left product because a transition is possible only if the system I can fire an action. We say the first system drives the transition relation.*

DEFINITION 2.10 (DEADLOCK). Let \perp be the property ($s \xrightarrow{a}_1 t \wedge s' \not\xrightarrow{a}_2$). The state (s, s') has a deadlock when \perp holds. The product is deadlock-free if it has no deadlocks.

REMARK 2.2. We make the difference between (s, s') being a deadlock state, and (s, s') being a terminal one (when the only possible transition in s is the idling transition).

THEOREM 2.11. Given the LTS I, S , where S is deterministic, we have that $I \circ S$ is deadlock-free iff $I \lesssim S$.

Recalling that we restrict the behavior of transition systems by adding constraints, we extend the previous result such that these are also reflected.

THEOREM 2.12. Given the LTS I, S , where S is deterministic, with the constraints p , resp. q , $I \circ S \models p \rightarrow (q \wedge \neg \perp)$ iff $I_p \leq S_q$.

We anticipate the description of our agent languages in order to underline that the semantics of BUnity language is encoded in deterministic LTS (BUnity agents cannot have different mental states by performing the same action). This is important: on the one hand, we can efficiently implement the left product; on the other hand, by Proposition 2.7 weak simulation is a complete proof method in our framework.

Note, however, that, when we say “BUnity agents are non-deterministic”, we refer to the fact that, at each step, they nondeterministically choose one among all the actions that could be executed.

3. FORMALIZING MENTAL STATES AND BASIC ACTIONS

In the current approach, the underlying logical framework of mental states is a fragment of Herbrand logic. We consider \mathcal{F} and Pred infinite sets of function, resp. predicate symbols, with a typical element f , resp. P . Variables are denoted by the symbol x . Each function symbol f has associated a non-negative integer n , its arity. Function symbols with 0-arity are also called *constants*.

Terms, usually denoted by the symbol t , are built from function symbols and variables. Formulas (denoted by the symbol φ) are built from predicates and the usual logical connectors. To sum up, the BNF grammar for terms and formulas is as follows:

$$\begin{aligned} t & ::= x \mid f(t, \dots, t) \\ \varphi & ::= P(t, \dots, t) \mid \neg \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \end{aligned}$$

An *atom* is any formula $P(t, \dots, t)$. A *literal* is either an atom or the negation of an atom. A term with no variables is called *ground*. A formula is either ground, or *open* (which has no quantifiers). All variables which appear in a formula are existential. The set of all ground atoms built upon \mathcal{F} and Pred is a *Herbrand model*.

Mental states are characterized in terms of *beliefs*. In the current framework, we consider beliefs as ground atoms, organized in the so-called *belief bases* (subsets of the Herbrand model), which we denote by \mathcal{B} .

Given \mathcal{B} a belief base, the satisfaction relation for ground formulas is defined using structural induction, as usually. For defining the satisfaction relation of open formulas, we consider the usual notion of *substitutions* as functions that replace variables with terms, denoted by $[x/t] \dots [x/t]$. Given

a syntactical expression e and a substitution θ , we denote the application of θ to e by $e\theta$, rather than by $\theta(e)$. The composition $\theta\theta'$ of two substitutions θ and θ' is defined as $\theta\theta'(x) := \theta(\theta'(x))$, and is an associative operation. The satisfaction relation for open formulas is as follows:

$$\mathcal{B} \models \varphi \text{ iff } \exists \theta \text{ s.t. } \mathcal{B} \models \varphi\theta$$

Such a substitution ($\exists\theta$) is obtained by solving a *matching problem*, since \mathcal{B} is ground. We say a term s *matches* a ground term t if there exists a substitution (called *matcher*) such that $s\theta$ is syntactically equal to t . The matching problem extends easily to formulas and belief bases. We say the matcher is the empty substitution ($\theta = \emptyset$) when φ is a ground formula satisfied in \mathcal{B} . We use the notation $\theta = \perp$ when the matching has no solution.

REMARK 3.1. We consider matching on its own, and not as a particular case of unification. A linear algorithm for matching is easy to implement, but for unification it is not.

Basic actions are functions of arity n , $a(x_1, \dots, x_n)$, defined as pairs (φ, ξ) , where φ are formulas which we call *preconditions*, and ξ are sets of literals which we call *effects*. The following inclusions are required:

$$\mathit{Var}(\xi) \subseteq \mathit{Var}(\varphi) = \{x_1, \dots, x_n\},$$

where $\mathit{Var}(e)$ denotes the set of variables in a syntactical expression e . We use the symbol \mathcal{A} for the set of basic actions. We refer to *Act* as the set of basic action *names*, with a typical element a , or $a\theta$.

Given a basic action $a = (\varphi, \xi)$, if matching φ to \mathcal{B} has a solution θ , then the effect of $a\theta$ is to update the belief base by adding or removing ground atoms from the set $\xi\theta$:

$$\begin{cases} \mathcal{B} \uplus l\theta = \mathcal{B} \cup l\theta, & l \in \xi \\ \mathcal{B} \uplus \neg l\theta = \mathcal{B} \setminus l\theta, & \neg l \in \xi \end{cases}$$

We write $\mathcal{B} \uplus \xi\theta$ to represent the result of an update operation, which is automatically guaranteed to be consistent since we add only positive literals.

3.1 BUnity Agents

BUnity language represents abstract agent specifications. Its purpose is to model agents at a coarse level, using a minimal set of constructions. A BUnity agent abstracts from specific orderings (for example, action planning). Hence her executions are highly nondeterministic.

The mental state of a BUnity agent is simply a belief base. On top of basic actions, BUnity language allows a finer type of construction, *conditional actions*, which are organized in a set denoted by \mathcal{C} .

A conditional action is built upon a basic action. It is syntactically defined by $\phi \triangleright do(a)$, where ϕ is a query on the belief base, and a is the name of an action. Intuitively, conditional actions are like *await* statements in imperative languages: **await** ϕ **do** a , action a can be executed only when ϕ matches the current belief base.

Both basic and conditional actions are enabled when a matching problem has a solution. However, note that the enabling conditions of basic actions are independent of the application, whereas those of conditional actions are application specific.

A BUnity agent is defined as a tuple, $(\mathcal{B}_0, \mathcal{A}, \mathcal{C})$, where \mathcal{B}_0 is a set of initial beliefs. For such a configuration, we define an operational semantics in terms of LTS.

DEFINITION 3.1 (BUNITY SEMANTICS). Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{C})$ be a BUnity configuration. The associated LTS is $(\Sigma, \mathcal{B}_0, L, \rightarrow)$, where:

- Σ is a set of states (belief bases)
- \mathcal{B}_0 is the initial state (the initial belief base)
- L is a set of ground action names
- \rightarrow is the transition relation, given by the rule:

$$\frac{\phi \triangleright do(a) \in \mathcal{C} \quad a = (\varphi, \xi) \in \mathcal{A} \quad \mathcal{B} \models (\phi \wedge \varphi)\theta}{\mathcal{B} \xrightarrow{a\theta} \mathcal{B} \uplus \xi\theta} \text{ (Act)}$$

We take, as an illustration, a known problem, which we first found in [11], of an agent building a tower of blocks. An initial arrangement of three blocks A, B, C is given there: A and B are on the floor, and C is on top of A . The goal of the agent is to rearrange them such that A is on the floor, B on top of A and C on top of B . The only action an agent can execute is to move one block on the floor, or on top of another block, if the latter is free.

$$\begin{aligned} \mathcal{B}_0 &= \{ on(C, A), on(A, floor), on(B, floor), \\ &\quad free(B), free(C), free(floor) \} \\ \mathcal{A} &= \{ move(x, y, z) = (on(x, y) \wedge free(x) \wedge free(z), \\ &\quad \{ on(x, z), \neg on(x, y), \neg free(z) \}) \} \\ \mathcal{C} &= \{ \neg(on(B, A) \wedge on(C, B)) \triangleright do(move(x, y, z)) \} \end{aligned}$$

Figure 2: A BUnity Toy Agent

The example from Figure 2 is taken in order to underline the difference between enabling conditions (for basic actions) and triggers (for conditional actions): on the one hand, it is possible to move a block x on top of another block z , if x and z are free; on the other hand, given the goal of the agent, moves are allowed only when the configuration is different than the final one.

3.2 BUPL Agents

The BUnity agent described in Figure 2 is highly non-deterministic. She can, for example, move C on the floor, B on A , and C on B , which is the shortest execution to achieve her goal. She can also pointlessly move C from A to B and then back from B to A .

BUPL language allows the construction of *plans* as a way to order actions. We refer to \mathcal{P} as a set of plans, with a typical element p , and to Π as a set of plan names, with a typical element π . Syntactically, a plan is defined by the following BNF grammar:

$$p ::= a(t, \dots, t) \mid \pi(t, \dots, t) \mid a(t, \dots, t); p \mid p + p$$

with ‘;’ being the *sequential composition* operator and ‘+’ the *choice* operator, with a lower priority than ‘;’.

The construction $\pi(x_1, \dots, x_n)$ is called *abstract plan*. It is a function of arity n , defined as $\pi(x_1, \dots, x_n) = p$. Abstract plans should be understood as procedures in imperative languages: an abstract plan calls another abstract plan, as a procedure calls another procedure inside its body.

BUPL language provides a mechanism for handling the failures of actions in plans through constructions called *repair rules*. A plan fails when the current action cannot be

executed. Repair rules replace such a plan with another. Syntactically, they have the form $\phi \leftarrow p$, and it means: if ϕ matches \mathcal{B} , then substitute the plan that failed for p .

A BUPL agent is a tuple $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$, where $\mathcal{B}_0, \mathcal{A}$ are the same as for a BUnity agent, p_0 is the *initial plan*, \mathcal{P} is a set of plans and \mathcal{R} is a set of repair rules.

Plans, like belief bases, have a dynamic structure, and this is why the mental state of a BUPL agent incorporates both the current belief base and the plan in execution. The operational semantics for a BUPL agent is as follows:

DEFINITION 3.2 (BUPL SEMANTICS). Let $(\mathcal{B}_0, \mathcal{A}, \mathcal{P}, \mathcal{R}, p_0)$ be a BUPL configuration. Then the associated LTS is $(\Sigma, (\mathcal{B}_0, p_0), L, \rightarrow)$, where:

- Σ is a set of states, tuples (\mathcal{B}, p)
- (\mathcal{B}_0, p_0) is the initial state
- L is a set of labels, either ground action names or τ
- \rightarrow represents the transition rules given in Figure 3:

$$\begin{aligned} &\frac{p = (a; p') \quad a = (\varphi, \xi) \in \mathcal{A} \quad \mathcal{B} \models \varphi\theta}{(\mathcal{B}, p) \xrightarrow{a\theta} (\mathcal{B} \uplus \xi\theta, p'\theta)} \text{ (Act)} \\ &\frac{(\mathcal{B}, p) \not\models \phi \leftarrow p' \in \mathcal{R} \quad \mathcal{B} \models \phi\theta}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \text{ (Fail)} \\ &\frac{\pi(x_1, \dots, x_n) = p}{(\mathcal{B}, \pi(t_1, \dots, t_n)) \xrightarrow{\tau} (\mathcal{B}, p(t_1, \dots, t_n))} \text{ (\pi)} \\ &\frac{(\mathcal{B}, p_i) \xrightarrow{\mu} (\mathcal{B}', p')}{(\mathcal{B}, (p_1 + p_2)) \xrightarrow{\mu} (\mathcal{B}', p')} \text{ (+)} \end{aligned}$$

Figure 3: BUPL Rules

where in Rule (π) $p(t_1, \dots, t_n)$ stands for $p[x_1/t_1] \dots [x_n/t_n]$.

Note that in the rules for choice μ can be either a ground action name or a silent step τ , in which case $\mathcal{B}' = \mathcal{B}$, and p' is a valid repair plan (if any).

We take as an example a BUPL agent that solves the *tower of blocks* problem. She has the same initial belief base and the same basic action as the BUnity agent.

$$\begin{aligned} \mathcal{B}_0 &= \{ on(C, A), on(A, floor), on(B, floor), \\ &\quad free(B), free(C), free(floor) \} \\ \mathcal{A} &= \{ move(x, y, z) = (on(x, y) \wedge free(x) \wedge free(z), \\ &\quad \{ on(x, z), \neg on(x, y), \neg free(z) \}) \} \\ \mathcal{P} &= \{ p_0 = move(B, floor, A); move(C, floor, B) \} \\ \mathcal{R} &= \{ on(x, y) \leftarrow move(x, y, floor); p_0 \} \end{aligned}$$

Figure 4: A BUPL Toy Agent

The BUPL agent from Figure 4 is modeled such that it illustrates the use of repair rules: we explicitly mimic a failure by intentionally telling the agent to move B on A . Similar scenarios can easily arise in multi agent systems: imagine that initially C is on the floor, and the agent decides she can move B on A ; imagine also that another agent comes and moves C on top of A , thus moving B on A will fail.

The failure is handled by $on(x, y) \leftarrow move(x, y, floor); p_0$. Choosing $[x/A][y/C]$ as a matcher, enables the agent to move C on the floor and after she can restart her plan.

3.3 The product of BUpL-BUnity agents

We recall that we focus on the nondeterminism of agent models, and this boils down to studying the refinement between abstraction levels of agent specifications. We have proposed BUnity as a typical abstract specification language, and BUpL as an implementation language.

The semantics of agents designed in these languages relates to labeled transition systems. For LTS, we have presented in Section 2 the technique for proving refinement: Theorem 2.11 states that deadlock freedom in a left synchronized product implies refinement between its components.

Applying the technique to the agent framework is straightforward, we just instantiate such components with the LTSs associated to BUnity and BUpL. If $S_1 = (\Sigma, (\mathcal{B}_0, p_0), Act \cup \{\tau\}, \rightarrow_1)$ and $S_2 = (\Sigma', \mathcal{B}_0, Act, \rightarrow_2)$ correspond to a BUpL, resp. BUnity agent, then their left synchronized product is $S_1 \circ S_2 = (\Sigma \times \Sigma', ((\mathcal{B}_0, p_0), \mathcal{B}_0), Act, \rightarrow)$. The semantics is given by the following transition rule:

$$\frac{\mathcal{B} \xrightarrow{a_2} \mathcal{B}'}{\langle (\mathcal{B}, p), \mathcal{B} \rangle \xrightarrow{a} \langle (\mathcal{B}', p'), \mathcal{B}' \rangle}, \quad \text{if } (\mathcal{B}, p) \xrightarrow{a_1} (\mathcal{B}', p')$$

Note that we consider the same initial belief base and the same set of actions for both agents. This is not a condition, but a choice for simplifying the notation.

Recall that the first component of a left product “drives” the simulation. We express this by conditioning the transition rule: only when the BUpL agent executes a , the transition can take place. If the BUnity agent can execute the same action, the product reaches a “good” state. Otherwise, the product is in a deadlock state.

Take, for example, the BUpL and BUnity agents building the *ABC* tower. Any visible action that BUpL executes can be mimicked by the BUnity agent, thus in this case BUnity simulates BUpL and refinement is guaranteed.

3.4 Call for Justice (or Compassion)

We imagine a scenario illustrative for cases where modeling fairness constraints is a “must”. For this, we slightly complicate the “tower” problem, by giving the agents an extra assignment to clean the floor, if it is dirty. The agents have two alternatives: either to clean or to build.

For both BUnity and BUpL agents we add a basic action, *magic-cleaning* = $(\neg cleaned, \{cleaned\})$. We enable the BUnity agent to execute this action at any time, by adding a conditional action $\top \triangleright do(magic-cleaning)$.

We assign a *mission* plan to the BUpL agent, *mission* = $clean + rearrange(B, A, C)$, where *clean* is a tail-recursive plan, *clean* = *magic-cleaning*; *clean*.

The plan *rearrange* generalizes the previously defined p_0 : $rearrange(x, y, z) = move(x, floor, y); move(z, floor, x)$. It consists of reorganizing free blocks placed on the floor, such that they form a tower. This plan fails if not all the blocks are on the floor, and the failure is handled by the already defined repair rule, which we call r_1 .

We add a repair rule, $r_2, \top \leftarrow mission$, which simply makes the agent restart the execution of the plan *mission*.

Note that it is possible that both agents always prefer cleaning the floor instead of rearranging blocks, though this is useless when the floor has already been cleaned. Never-

theless, such cases are disregarded if one requires that executions are fair.

We model fairness as LTL formulas. Basically, we need only two future operators, \diamond (*eventually*) and \square (*always*). The corresponding satisfaction relation is defined as follows:

$$\begin{aligned} \sigma \models \diamond \phi & \text{ iff } (\exists i > k) s_i \models \phi \\ \sigma \models \square \phi & \text{ iff } (\forall i > k) s_i \models \phi, \end{aligned}$$

where s_0, \dots, s_k, \dots are the states of a computation σ .

In function of the agent language, we distinguish between different types of fairness. In the case of the BUnity agent, it suffices to consider weak fairness: any enabled basic action a such that there is a conditional one $\phi \triangleright do(a)$, which can continuously often trigger a , should be infinitely often taken:

$$just_1 = \bigwedge_{a \in \mathcal{A}} (\diamond \square enabled(\phi \triangleright do(a)) \rightarrow \square \diamond taken(a)).$$

where *enabled* and *taken*, predicates on the states of the left product, are defined as:

$$\begin{aligned} \langle (\mathcal{B}, p), \mathcal{B} \rangle \models enabled(\phi \triangleright do((\varphi, \xi))) & \text{ iff } \mathcal{B} \models \phi \wedge \varphi \\ \langle (\mathcal{B}, p), \mathcal{B} \rangle \models taken(a) & \text{ iff } \mathcal{B} \xrightarrow{a_2} \mathcal{B}' \end{aligned}$$

For the BUpL agent, we consider two scenarios for defining fairness with respect to choices in repair rules and plans:

The execution of *rearrange* has failed. Both repair rules r_1 and r_2 are enabled, and always choosing r_2 makes it impossible to make the rearrangement. This would not be the case if r_1 were triggered. It follows that the choice of repair rules should be weakly fair:

$$just_2 = \bigwedge_{p \in \mathcal{P}} (\diamond \square enabled(\phi \leftarrow p) \rightarrow \square \diamond taken(p)).$$

The repair rule r_1 has been applied, and all three blocks are on the floor. Returning to the initial mission and being in favor of cleaning leads again to a failure (the floor is already clean). The only applicable repair rule is r_2 which simply tells the agent to return to the mission. Thus, it can be the case that, though rearranging the blocks is enabled, it will never happen, since the choice goes for magic-cleaning (which always fails). Therefore, because plans are not continuously enabled, their choice has to be strongly fair:

$$compassionate = \bigwedge_{p \in \mathcal{P}} (\square \diamond enabled(p) \rightarrow \square \diamond taken(p))$$

In the above scenarios *enabled* and *taken* are defined similarly as in the case of BUnity language: a repair rule is enabled when its precondition is satisfied in the belief base; a plan is enabled when the precondition of its first action is satisfied; a plan is taken when its first action is taken.

We pose the question whether fair executions of the BUpL agent refine fair executions of the BUnity agent. Conforming to Theorem 2.12, the answer is positive if the formula $(compassionate \wedge just_2) \rightarrow (just_1 \wedge \neg \perp)$ is satisfied in the left product.

It is possible to prove by hand that this is the case, by analyzing all possible executions. However, we consider it is of interest to prove it automatically. This is the reasoning underlying our implementation, which constitutes the subject in the next section.

4. EXECUTABLE BUNITY-BUPL

Our approach in implementing the operational semantics of BUnity and BUPL languages is to map them into rewrite theories, which are rewriting logic [13] specifications.

Rewriting logic is a logic of *becoming* and *change*, in the sense that it reasons about the evolution of the systems: a rewrite theory is mainly a signature and a set of rewrite rules; the signature describes the states of a system, and the rewrite rules are executions of the transitions.

We choose Maude [6] as a rewriting logic language implementation. The main objective of Maude is to support executable specifications, formal method applications, and reflective computations. The reflective capabilities of Maude make it possible to extend it to an user-definable system. Furthermore, Maude comes with an LTL model-checker.

Maude's basic statements are equations and rules. Maude programs containing only equations are called *functional modules*, while those containing in addition rules are called *system modules*. We use functional modules to define the syntax of our agent languages, and system modules to define the semantics.

For example, assume a functional module SYNTAX where we write constructions common to both languages. The top-level syntax will then be:

```
fmod SYNTAX is
...
endfm
```

with '...' standing for actual declarations and statements.

The first thing to declare in a specification are the types (*sorts*) which can be ordered via a *subsort* relation. For example, we declare **Belief**, **BeliefBase** at once using the keyword **sorts**, and that **Belief** is a subsort of **BeliefBase**:

```
sorts Belief BeliefBase .
subsort Belief < BeliefBase .
```

A belief base is a set of beliefs, and this definition corresponds to the declaration of an operator ';':

```
op _;- : BeliefBase BeliefBase -> BeliefBase
[assoc comm] .
```

where ';' is in mixfix form, and **assoc**, **comm** are attributes declaring that the set is associative and commutative.

In some cases, the notation in the code differs from the one used previously (for example the above ';' instead of ','). This is not a discrepancy, but a natural consequence since certain symbols have a standard meaning in Maude, not compatible with ours. However, the correspondence between notations should be clear from the context.

Using **op** $\langle Name \rangle : \langle Sort_1 \rangle \dots \langle Sort_k \rangle \rightarrow \langle Sort \rangle$ as the general statement for declaring operators, a query is:

```
subsort Term < Query .
op ~_ : Query -> Query .
op _/\_ : Query Query -> Query .
op _\/_ : Query Query -> Query .
```

where **Term** is a metalevel construction, and in order to use it, we need to add **protecting META-LEVEL .** after the declaration **fmod SYNTAX is**.

Predicates are a particular type of terms, and because we want to simplify the notation, we consider any term as a query (by declaring **Term** as a subsort of **Query**).

The matching problem is defined inductively on the structure of queries. We present only the case of terms. At this point, the reflective features of Maude come in handy.

The reflective kernel of Maude is the built-in **META-LEVEL** module, where Maude terms are reified as elements of a type **Term** of terms, and Maude modules are reified as terms in a type **Module** of modules. The **META-LEVEL** provides the capacity of representing theories (including itself) as data.

Matching a term to a belief means simply invoking the **metaMatch** function:

```
op match : Query BeliefBase Nat -> Substitution .
var T : Term . var B : Belief . var N : Nat .
eq match(T, B, N) =
  metaMatch(upModule('AGENT-EXAMPLE, false),
    upTerm(T), upTerm(B), nil, N) .
```

Note that functions are defined by statements **eq** $\langle Term_1 \rangle = \langle Term_2 \rangle$. and variables are declared with the keyword **var**.

The **metaMatch** tries to match at the top the metarepresentations of the terms **T** and **B** in the metarepresentation of the module **AGENT-EXAMPLE**, where **T** and **B** are instantiated. The result is either an element of sort **Substitution** or the constant **noMatch**, already defined at the metalevel. The matching might have multiple solutions, and **N** indicates which one should be returned.

We could have implemented our own matching function, but there is no point "reinventing the wheel", since Maude provides us with a very efficient matching. Besides, it is a good occasion to advocate the elegance of *metaprogramming*, combining up and down moves between reflection levels.

A basic action is an element of sort **B-Act**, defined as a pair of a query and an effect, where effect is a set of literals, and a literal is a term or its negation:

```
sorts Lit Effect . subsort Lit < Effect .
subsort Term < Lit . op neg : Term -> Lit .
op _;- : Effect Effect -> Effect [assoc comm] .
sort B-Act . op [_,-] : Query Effect -> B-Act .
op effect : B-Act -> Effect .
eq effect ([Q:Query, E:Effect]) = E .
```

The effect of a basic action is to add (remove) literals to (from) the belief base, where **add**, **remove** are basic operations on sets. The function **update** is defined recursively:

```
var T : Term . var L : Lit . var R : Effect .
var BB : BeliefBase .
eq update(BB, neg(T)) = remove(BB, T) .
eq update(BB, T) = add(BB, T) .
eq update(BB, L; R) = update(update(BB, L), R) .
```

The syntax of BUnity language is defined in the module **BUNITY-SYNTAX**, where, besides including the already defined constructions from **SYNTAX**, we declare BUnity specifics: conditional actions (elements of sort **C-Act**) and mental states (elements of sort **ByMentalState**):

```
fmod BUNITY-SYNTAX is
protecting SYNTAX .
sorts C-Act ByMentalState LabelA .
op _do(_ ) : Query B-Act -> C-Act .
op <<-> : BeliefBase -> ByMentalState .
op [_]- : LabelA ByMentalState
-> ByMentalState [frozen] .
endfm
```

Note the declaration **[_]-** with the use of the attribute **frozen**. We need this construction in the implementation of the left synchronized product.

The BUnity semantics is encoded in system modules (note the **mod ... endm**). BUnity has only one transition rule, which we map into a conditional rewrite rule, using the statement **cr1** $[\langle Label \rangle] : \langle Term_1 \rangle \Rightarrow \langle Term_2 \rangle$ **if** $\langle C_1 \rangle \wedge \dots \wedge \langle C_k \rangle$.

```

mod BUNITY-SEMANTICS is
protecting BUNITY-SYNTAX .
op eqS : -> EquationSet .
eq eqS = upEqs('AGENT-EXAMPLE, false) .
var BB : BeliefBase . var A : B-Act .
crl [act1]: < BB > => < update(BB, effect(A)) >
  if (Q:Query do(A)) in iniC(eqS, BB, 0) .
crl [act2] : < BB > => [name(A)]
  < update(BB, effect(A)) >
  if (Q:Query do(A)) in iniC(eqS, BB, 0) .
endm

```

where `iniC` returns the set of instantiated conditional actions such that the preconditions are true in the belief base. It analyzes recursively all equations from the module **AGENT-EXAMPLE**. For each equation defining a conditional action, it matches the precondition to the belief base and it returns all possible solutions:

```

var BB : BeliefBase . vars Q1 Q2 : Query .
var L : LitSet . var N : Nat .
vars T T' : Term . var S : Substitution .
ceq iniC(eq T = T' [label('c-act)] ., BB, N)
  = downTerm(substitute(T, S), err)
  iniC(eq T = T' [label('c-act)] ., BB, N + 1)
  if Q1 do([Q2, L]) := downTerm(T', err)
  /\ S := match(Q1 /\ Q2, BB, N)
  /\ S /= noMatch .

```

We provide two rewrite rules: the first one `[act1]`, is used when model-checking BUnity agents on their own, the last one is needed when model-checking left synchronized products. In the latter case we need only one step successors, and we enforce such a result by freezing the BUnity agent after performing one step (configurations like `[]_` have no applicable rewrite rules). It is up to the left product to “defrost” the agent.

The procedure for implementing BUPL language is similar, although longer. Due to lack of space we do not include it here. In exchange, we present it for the left product. The syntax is simply a triplet of a label and the mental states of a BUnity and a BUPL agent:

```

sort Label . sort LeftProduct .
op <_,_,> : BpMentalState ByMentalState Label
  -> LeftProduct [frozen] .

```

We use the attribute `frozen` in the declaration of the left product such that it is forbidden to rewrite the BUPL (resp. the BUnity agent) on its own. This is because the left product is synchronous (one agent executes an action only if the other one does):

```

vars Bp Bp' : BpMentalState . var LA : LabelA .
vars By By' : ByMentalState . var LP : LabelP .
crl [act] :
  < Bp, By, L:Label > => < Bp', By', LA >
  if Bp => [LA] Bp' /\ By => [LA] By' .

crl [tau] :
  < Bp, By, L:Label > => < Bp', By, LP >
  if Bp => [LP] Bp' .

```

Note that in Maude conditions are evaluated from left to right, and, therefore, the order in which they appear, although mathematically inessential, is very important operationally [7]. This suits perfectly our setup, since the left product is driven by its first component.

Note also how defrosting works: by doing one step each agent reaches a configuration from where no rewrite is possible. However, in the right hand side of the Rule `[act]` the agent is enabled to do one step again.

Recall Th. 2.11 states that no deadlock under fairness in the left product implies refinement between the components of the product. Deadlock and fairness are modeled as predicates on the states of the left product.

A state predicate is an operator of sort `Prop`. Its meaning is given by: `op _|=_ : State Prop -> Bool`. For example, we present the predicates needed for `just1`, like in §3.4. For the rest, the procedure is similar.

```

subsort LeftProduct < State .
op taken : B-Act -> Prop .
ceq < Bp, By, L > |= taken(A) if name(A) == L .
op enabled : C-Act -> Prop .
ceq < Bp, < VB >, L > |= enabled(Q do(A))
  if C in iniC(eqS, VB, 0) .
op just1 : C-Act -> Prop .
ceq just1(C) = <>[] enabled(C) -> []<> taken(A)
  if Q do(A) := C .

```

The variables in the fragment of code are the usual ones defined in previous examples. The function `just1` is extended inductively for the set of all conditional actions.

The examples presented in this section are fragments from the implementation, hence not executable on their own. The complete code, available for download, can be used for further refinement testing of BUnity and BUPL instances. Given the existing implementation, all that is left to the user is to write Maude code for BUnity and BUPL agents, and to use the Maude command `red modelCheck(...)`.

If, for example, the user may want to input the building agents from §3.4, then the fragment corresponding to the BUnity agent would look like:

```

sorts Block . ops a b c fl : -> Block .
op on : Block Block -> Belief .
op free : Block -> Belief .
op bb : -> BeliefBase . vars X Y Z : Block .
eq bb = on(c, a) ; free(c) ; on(a, fl) ;
  on(b, fl) ; free(b) ; free(fl) .

op move : Block Block Block -> B-Act .
eq [b-act] : move(X, Y, Z) =
  [on(X, Y) /\ free(X) /\ free(Z),
  neg on(X, Y) ; on(X, Z) ; free(Y);
  neg free(Z) ; free(fl)] .

op c : Block Block Block -> C-Act .
eq [c-act] : c(X, Y, Z) = ~ (on(b, a) /\
  on(c, b)), do(move(X, Y, Z)) .
op by : -> ByMentalState . eq by = < bb > .

```

Assuming the same is done for the BUPL agent called `bp`, and that `lp` is the left product of `bp` and `by`, the user may check for refinement under fairness assumptions. The result would look like:

```

Maude> red modelCheck(lp, (just2 /\ compassion)
  -> (just1 /\ []~ deadlock)) .
reduce in AGENT-EXAMPLE : modelCheck(lp, (just2
  /\ compassion) -> (just1 /\ []~ deadlock)) .
rewrites: 38617 in 670ms cpu (2103ms real)
(57560 rewrites/second)
result Bool: true

```

The result of model-checking in Maude is either the boolean value `true`, or a counterexample, which is defined as a pair consisting of a finite path and the cycle which can be reached (for any unsatisfiable LTL formula such a counterexample exists, [9]). We obtained `true`, which means that the BUPL agent refines the BUnity agent, as it was expected.

4.1 What else can model-checking do?

Though we have focused only on model-checking refinement, note, however, that model-checking LTL properties for single agents is derived within. Moreover, it is known that refinement preserves safety properties, thus safety properties are intrinsically true for a BUpL agent if they are true for the BUnity specification.

As for liveness properties, the most we can do is to conclude that under fairness assumptions something good eventually will happen. For example, we can define the predicates $goal_1$, $goal_2$ on the states of BUnity:

```
eq < on(a, f1) ; on(b, a); on(c, a);
  BB: BeliefBase > |= goal1 = true .
eq < cleaned; BB: BeliefBase > |= goal2 = true .
```

meaning that if, eventually, the rearrangement succeeds then $goal_1$ is achieved, in which case the implementation is correct. Model-checking $\diamond goal_1$ returns a counter-example: the trace consisting of an infinite loop *magic-cleaning*. However, justice implies correctness: model-checking $just_1 \rightarrow \diamond goal_1$ returns true. The same reasoning applies for $\Box \diamond (goal_1 \vee goal_2)$: if the BUnity agent is just, then it is always the case that she will eventually achieve one or the other goal.

5. CONCLUSIONS AND FUTURE WORK

We have addressed the problem of model-checking agent refinement. We have first presented the underlying theory, which we have implemented in Maude. The complete implementation is available for public download at <http://homepages.cwi.nl/~astefano/agents>.

Note that our proof technique is semi-decidable. A sufficient condition for being decidable is to allow only variables, constants and predicates in the model. If one allows even one functional symbol, the Herbrand model is infinite. To be precise, with just one function we can encode natural numbers: consider a belief base with one belief $p(0)$, and a basic action $inc(X) = (p(X), \{\neg p(X), p(s(X))\})$, where s is the usual successor function. The plan $flyToInfinite(X) = inc(X)$; $flyToInfinite(X)$ is not terminating and, moreover, gives rise to infinite reachable states. In this case, model-checking fails, but one can still try the Maude `search` command, which is based on a breadth first search.

However, forbidding any functional terms as parameters of the predicates in the belief base keeps the system on the safe side. When the restriction is too strong, abstraction techniques are of great interest. In such cases, refinement evolves on two axes: one is the refinement of nondeterministic choices (which has been considered in this paper), and the other is the refinement of data (which needs to be investigated). The basic idea is to make BUnity agents finite state systems such that they faithfully represent the behaviors of infinite state BUpL agents. Future work consists of implementing such abstraction techniques.

Though we have considered only single agents, our results extend to multi agent systems by incorporating an action-coordination mechanism. Theoretically, we have that refinement is compositional: if the agents in a MAS refine the agents in another MAS, then the refinement relation between the two MAS is preserved in the presence of a coordinator. Practically, we have already experimented with Reo [1] as a platform for implementing coordinators. Future work concerns extending the current implementation, which is available at the above mentioned location.

6. REFERENCES

- [1] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
- [2] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.
- [3] T. Bosse, C. M. Jonker, L. van der Meij, A. Sharpanskykh, and J. Treur. Specification and verification of dynamics in cognitive agent models. In *IAT*, pages 247–254, 2006.
- [4] M. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, 1987.
- [5] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley Publishing Company, Inc., Reading, Massachusetts, 1988.
- [6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.
- [7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
- [8] F. S. de Boer, K. V. Hindriks, W. van der Hoek, and J.-J. C. Meyer. A verification framework for agent programming with declarative goals. *J. Applied Logic*, 5(2):277–302, 2007.
- [9] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
- [10] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software: Proc. 10 th Intl. SPIN Workshop*, volume 2648 of *LNCS*, pages 230–234. Springer, 2003.
- [11] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
- [12] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
- [13] J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools.
- [14] F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *J. Applied Logic*, 5(2):235–251, 2007.
- [15] S. Safra. *Complexity of automata on infinite objects*. PhD thesis, Rehovot, Israel, 1989.
- [16] M. B. van Riemsdijk, F. S. de Boer, M. Dastani, and J.-J. C. Meyer. Prototyping 3apl in the maude term rewriting language. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1279–1281, New York, NY, USA, 2006. ACM Press.
- [17] A. Verdejo and N. Martí-Oliet. Two case studies of semantics execution in maude: Ccs and lotos. *Form. Methods Syst. Des.*, 27(1/2):113–172, 2005.