Using Rewrite Strategies for Testing BUpL Agents

Lăcrămioara Aştefănoaei*[1], Frank S. de Boer[1,2], M. Birna van Riemsdijk[3]

[1] CWI, Amsterdam, The Netherlands
[2] LIACS - Leiden University, The Netherlands
[3] TU, Delft, The Netherlands

Abstract. In this paper we focus on the problem of testing agent programs written in BUpL, an executable, high-level modelling agent language. Our approach consists of two main steps. We first define a formal language for the specification of test cases with respect to BUpL. We then implement test cases written in the formal language by means of a general method based on rewrite strategies. Testing an agent program with respect to a given test case corresponds to strategically executing the rewrite theory associated to the agent with respect to the strategy implementing the test case.

Keywords: Agent Languages, Testing, Rewriting, Strategies

1 Introduction

An agent is commonly seen as an encapsulated computer system that is situated in some environment and that is capable of flexible, *autonomous action* in that environment in order to meet its design objectives [12], or *dynamic goals*. An important line of research in the agent systems field is the design of agent languages [3] with emphasis on the use of formal methods. The guiding idea is that agent-specific concepts such as beliefs (representing the environment and possibly other data the agent has to store), goals (representing the desired state of the environment), and plans (specifying which sequences of actions and possibly compositions of other plans to execute in order to reach the goals) facilitate the programming of agents. Along these lines, we take as case of study in this paper a simple variant of 3APL [7], the agent language BUpL, which is introduced in [1]. There the authors advocate the use of the Maude language [4] and its supporting tools for *prototyping*, *executing*, and *verifying* BUpL agents. One of the main advantages of Maude is that it provides a single framework in which the use of a wide range of formal methods is facilitated. Namely, being a rewrite-based framework, it makes it is easy to prototype modelling languages with an operational semantics by means of rewrite theories [8], and it provides mechanisms for verifying programs and language definitions by means of LTL model-checking [6]. Furthermore, the inherent reflective feature of rewriting logic (and of Maude, in particular) offers an alternative to model-checking by means of rewrite strategies.

In this paper, we extend the results from [1]. More precisely, we investigate the problem whether a BUpL agent is conformant with respect to a given specification,

* **Email**: L.Astefanoaei@cwi.nl; **Address**: Centrum voor Wiskunde en Informatica (CWI),P.O. Box 94079,1090 GB Amsterdam, The Netherlands; **Tel.**: +31 (0)20 592 4368

however, from a different perspective. We understand conformance as the refinement relation in [1], that is, it holds when the set of traces of a BUpL agent is included in the set of traces of the specification. In a straight-forward approach, one solution is to look at each execution trace of the agent and to check whether it is also a trace of the specification. However, this is often practically unfeasible due to large (possibly infinite) sets of agent executions. A more clever way is to consider the trace inclusion problem in the opposite direction, that is, to look first at the traces of the specification and to check whether these are also traces of the agent. Usually, "check" is achieved by model-checking or inductive verification. However, both approaches have their disadvantages: with model-checking one might run into the state explosion problem, while inductive verification is not automatic. An orthogonal technique is to use *testing*.

In the literature, the very basic idea behind testing is that it aims at showing that the intended and the actual behaviour of a system differ by generating and checking individual executions. Testing object-oriented software has been extensively researched and there are many pointers in the literature with respect to manual and automated, partition and random testing, test case generation, criteria for test selection (please see [9] for an overview). In an agent-oriented setup, there are less references. A few pointers are [13, 10] for developing test units from different agent methodologies, however the direction is orthogonal to the one we consider.

Our testing methodology consists of the following steps. We see the traces of the specification as the basic constructions for *test cases*. Since specifications are meant to be "small", generating test cases is a much simpler task than exhaustively exploring possible agent executions. Either represented by regular expressions or by finite transition systems, specifications can be used to generate test cases by model-checking, for example. Traces are *deterministic*, and since we build test cases on top of traces, also test cases are deterministic, in contrast to specifications. This is an important feature which makes testing an efficient approach. We define test cases as pairs of tests on actions and tests on facts. The tests on actions are finite sequences of pairs (a, R) where a is the action to be executed and R is the set of actions which are allowed to be executed at a given state. Whenever the agent *cannot execute* the action specified by the test on actions, or whenever the agent *can execute* a *forbidden* action, the corresponding trace represents a nonconformant execution. Tests on facts are temporal formulae that are checked on the traces generated with respect to tests on actions. They can be further used to detect "bad" executions.

Given that we define a formal language for expressing *what* a test case is, we then describe *how* to implement test cases. Namely, we provide a strategy-based mechanism to define *test drivers*. In a rewrite-based framework, strategies are meant to control nondeterministic executions by instrumenting the rewrite rules at a meta-level. Usually, in concrete implementations the nondeterminism is reduced by means of scheduling policies. While testing a concrete implementation, e.g., a multi-threaded Java application, there is no obvious distinction between testing the program itself and testing the default scheduling mechanism of the threads. We emphasise that the language we consider, BUpL, is a modelling language, where the *nondeterminism* in choices among plans, exception handling mechanisms and internal actions is a main aspect we deal with. Strategies give a great degree of flexibility which becomes important when the interest is in verification. For example, in our case, in order to analyse or experiment with a new testing formalism one only needs to change the *strategy* instead of changing *the semantics of the agent language* or *the agent program* itself.

Though test cases are deterministic, test drivers need to search all intermediary states that can be reached by nondeterministically executing internal BUpL computations. Defining test drivers by means of strategies is an elegant solution to the implicit nondeterminism in BUpL. However, it does not directly solve the problem of possibly divergent executions of internal steps. To avoid some divergent computations, we need to impose restrictions on the application of the strategies. This makes it less intuitive that test drivers are faithfully implementing test cases, and thus the last issue we focus upon is the correctness of our mapping between test cases and test drivers.

2 BUpL Agents By Example

In this section, we briefly present the syntax and semantics of BUpL for ease of reference and completeness. A BUpL agent has an initial belief base and an initial plan. A belief base is a collection of ground (first-order) atomic formulae which we refer to as beliefs. The agent is supposed to execute its initial plan, which is a sequential composition and/or a nondeterministic choice of actions or composed plans. The semantics of actions is defined using pre and post conditions. An action can be executed if the precondition of the action matches the belief base. The belief base is then updated by adding or removing the elements specified in the postcondition. When, on the contrary, the precondition does not match we say the execution of the action (or the plan of which it is a part) fails. In such a case repair rules are applied (if any), and this results in replacing the plan that failed.

Syntactically, a BUpL agent is a tuple ($\mathcal{B}_0$, p_0, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}$), where $\mathcal{B}_0$ is the initial belief base, p_0 is the initial plan, $\mathcal{A}$ is the set of internal and observable actions, $\mathcal{P}$ are the plans, and $\mathcal{R}$ are the repair rules. The initial belief base and plan form the initial mental state of the agent. To illustrate the syntax, we take as an example a BUpL agent that solves the Hanoi towers problem. We represent blocks by natural numbers. We assume that the initial configuration is of three blocks arranged on a table as follows: blocks 1 and 2 are on the table (0), and 3 is on top of 1. The agent has to rearrange them such that they form the tower 321 (1 is on 0, 2 on top of 1 and 3 on top of 2). The only action the agent can execute is $move(x, y, z)$ to move block x from block y onto z, if x and z are clear. Blocks can always be moved to the table, i.e., the table is always clear.

$$
\begin{aligned}
\mathcal{B}_0 &= \{\ on(3,1), on(1,0), on(2,0), clear(2), clear(3), clear(0)\ \} \\
p_0 &= build \\
\mathcal{A} &= \{\ move(x,y,z) = (on(x,y) \wedge clear(x) \wedge clear(z), \{on(x,z), \neg on(x,y), \neg clear(z)\})\ \} \\
\mathcal{P} &= \{\ build = move(2,0,1); move(3,0,2)\ \} \\
\mathcal{R} &= \{\ on(x,y) \leftarrow move(x,y,0); build\ \}
\end{aligned}
$$

Fig. 1. A BUpL Toy Agent

The BUpL agent from Figure 1 is modelled such that it illustrates the use of repair rules: we explicitly mimic a failure by intentionally writing a plan to move block 2 onto 1. This is not possible, since block 3 is already on top of 1. Similar scenarios can easily arise in multi-agent systems: imagine that initially 3 is on the table, and the agent decides to move 2 onto 1; imagine also that another agent comes and moves 3 on top of 1, thus moving 2 onto 1 will fail. The failure is handled by the repair rule $on(x, y) \leftarrow move(x, y, 0); build$. Choosing $[x/3][y/1]$ as a matcher enables the agent to move block 3 onto the table and then the initial plan can be restarted.

We shortly describe (please see [1] for more details) the BUpL operational semantics. The states of BUpL agents are pairs of belief bases and plans, symbolically denoted by $(\mathcal{B}, p)$. These BUpL states change with respect to the transition rules in Figure 2.

$$\frac{p = (a; p') \qquad a = (\psi, \xi) \in \mathcal{A} \qquad \theta \in Sols(\mathcal{B} \models \psi)}{(\mathcal{B}, p) \xrightarrow{(\tau/a\theta)} (\mathcal{B} \uplus \xi\theta, p'\theta)} \; ((i/o)\text{-}act)$$

$$\frac{}{(\mathcal{B}, (p_1 + p_2)) \xrightarrow{\tau} (\mathcal{B}, p_i)} \; (sum_i, i \in \{1, 2\})$$

$$\frac{(\mathcal{B}, a; p) \overset{a}{\not\to} \qquad \phi \leftarrow p' \in \mathcal{R} \qquad \theta \in Sols(\mathcal{B} \models \phi)}{(\mathcal{B}, p) \xrightarrow{\tau} (\mathcal{B}, p'\theta)} \; (fail\text{-}act)$$

$$\frac{\pi(x_1, \ldots, x_n) := p}{(\mathcal{B}, \pi(t_1, \ldots, t_n)) \xrightarrow{\tau} (\mathcal{B}, p(t_1, \ldots, t_n))} \; (\pi)$$

Fig. 2. BUpL Rules

The rules $(i\text{-}act)$ and $(o\text{-}act)$[4] capture the effects of performing action a (either internal or observable), which is the head of the current plan. These rules basically say that for a given as a pair of a precondition (i.e., a first order formula) ψ and a postcondition (i.e., a set of literals) ξ, if θ is a solution (i.e., a substitution) such that ψ matches[5] $\mathcal{B}$ (i.e., $\mathcal{B} \models \psi\theta$), then the current mental state changes to a new one, where the belief base is updated by adding/removing the positive/negative literals from ξ. It is also the case that the current plan becomes $p'\theta$, that is, the "tail" of the previous plan p instantiated with respect to θ. The transition rule $(fail\text{-}act)$ handles exceptions. If the head of the current plan is an action that cannot be executed (the set of solutions for the matching problem is empty) and if there is a repair rule $\phi \leftarrow p'$ such that the new matching problem $\mathcal{B} \models \phi$ has a solution θ then the plan is replaced by $p'\theta$. The transition rule (π) implements "plan calls". If the abstract plan $\pi(x_1, \ldots, x_n)$ defined as $p(x_1, \ldots, x_n)$ is instantiated with the terms $t_1, \ldots, t_n$ then the current plan becomes $p(t_1, \ldots, t_n)$ which stands for $p[x_1/t_1] \ldots [x_n/t_n]$. The transition rule (sum_i) replaces a choice between two plans by either one of them.

[4] For simplicity, they are denoted by the same transition $((i/o)\text{-}act)$. Syntactically, the only difference between them is that the label for *i-act* is τ.

[5] Belief bases are sets of *ground* positive literals, thus we solve a generalisation of the *matching* and not unification problem.

2.1 Prototyping BUpL Agents as Rewrite Theories

In [1] it is shown how the operational semantics of BUpL can be implemented and executed as a *rewrite theory* in Maude. The main advantage of using Maude for this is that the translation of operational semantics into Maude is direct [11], ensuring a *faithful implementation*. Thanks to this, it is relatively easy to experiment with different kinds of semantics, making Maude suitable for rapid *prototyping*.

We do not explain here the way BUpL is prototyped in Maude but we briefly illustrate at a more generic level how BUpL transition rules map into rewrite rules. A rewriting logic specification or rewrite theory is a tuple $\langle \Sigma, E, R\rangle$, where Σ is a signature consisting of sorts (types) and function symbols, E is a set of equations and R is a set of rewrite rules. The signature describes the *terms* that form the state of the system. These terms can be rewritten using equations and rewrite rules. Rewrite rules are used to model the dynamics of the system, i.e., they describe transitions between states. Equations form the functional part of a rewrite theory, and are used to reduce terms to their "normal form" before they are rewritten using rewrite rules. The application of rewrite rules is intrinsically nondeterministic, which makes rewriting logic a good candidate for modelling concurrency.

In our case, the signature (the set of terms) maps the mental states of the agents and the rewrite rules map BUpL transitions, thus they describe how BUpL mental states change. There is a natural encoding of transition rules as *conditional rewrite rules*. The general mathematical format of a conditional rewrite rule is as follows:

$$l : t \to t' \; if \; (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \to q_k)$$

It basically says that l is the label of the rewrite rule $t \to t'$ which is used to "rewrite" the term t to t' when the conditions on t are satisfied. Such conditions can be either equations like $u_i = v_i$, memberships like $w_j : s_j$ (that is, w_j is of type s_j) or other rewrites like $p_k \to q_k$. For example, the corresponding rewrite rule for transition (act) in the case of observable actions is:

$$\begin{gathered} \textit{o-act} : (\mathcal{B}, p) \to (update(\mathcal{B}, \xi\theta), p'\theta) \; \textit{if} \; p = \textit{o-a}; p' \wedge \textit{o-a} = (\psi, \xi) \wedge \\ \theta = match(\mathcal{B}, \psi) \wedge \textit{o-a} : A^o \end{gathered}$$

where A^o denotes the sort of observable actions. As it will be clear in the next sections, we need the distinction between internal and observable actions for testing, in order to have a more expressive framework.

All other transition rules are encoded as rewrite rules in a similar manner and we do not further explain them. In what follows, we only need to remember that each transition has a corresponding rewrite rule labelled with the same name.

2.2 Meta-controlling BUpL Agents with Rewrite Strategies

In this section we make a short overview of the strategy language presented in [5] with illustrations of how strategies can be used to control the execution of BUpL agents. We denote the rewrite theory that implements the operational semantics of BUpL by

T. Given a BUpL agent, we denote by ms terms corresponding to BUpL mental states $(\mathcal{B}, p)$. These terms can be rewritten by the rewrite rules from T. We further denote by S the strategy language from [5]. The strategy language S can be viewed as a transformation of the rewrite theory T into $S(T)$ such that the latter represents the execution of T in a controlled way. Given a strategy expression E in the strategy language S, the application of E to ms is denoted by $E@ms$. The semantics of $E@ms$ is the set of successors which result by rewriting ms using the rewrite rules from $S(T)$.

The simplest strategies we can define in the strategy language S are the constants *idle* and *fail*: *idle* $@\ ms = \{ms\}$, *fail* $@\ ms = \emptyset$. Another basic strategy consists of applying to a BUpL agent state ms a rule identified by one of the labels: *i-act*, *o-act*, *fail-act*, or *sum*, possibly with instantiating some variables appearing in the rule. The semantics of $l@ms$, where l is one of the above rule labels, is the set of all terms to which ms rewrites in one step using the rule labelled l. For example, applying the strategy *o-act* to the initial state $(\mathcal{B}_0, \mathit{build})$ of the BUpL builder from Figure 1 has as result $\emptyset$ because initially the only possible observable action $move(2, 0, 1)$ fails. However, applying the strategy *fail-act* has as result the set $\{(\mathcal{B}_0, (move(3, 1, 0); \mathit{build})),$ $(\mathcal{B}_0, (move(1, 0, 0); \mathit{build})),\ (\mathcal{B}_0, (move(2, 0, 0); \mathit{build}))\}$, thus the set of all possible states reflecting a solution to the matching problem $\mathcal{B}_0 \models on(x, y)$. Of course, some of these resulting states are meaningless in the sense that there is no point in moving a block from the table to the table. A much more adequate strategy is *fail-act*$[\theta \leftarrow [x/3][y/1]]$, that is, to explicitly give the value we are interested in to the variable θ which appears in the rewrite rule *fail-act*. This results in a set containing only the state $(\mathcal{B}_0, (move(3, 1, 0); \mathit{build}))$.

Since matching is one of the basic steps that take place when applying a rule, another strategy one can define is *match* T *s.t.* C. When applied to a given state term ms, the result of this strategy is $\{ms\}$ if ms matches the pattern T and the condition C is satisfied with the substitutions for the variables obtained in the matching, otherwise $\emptyset$. For example, applying *match* $(\mathcal{B}, p)$ *s.t.* $on(2, 1) \in \mathcal{B}$ to $(\mathcal{B}_0, \mathit{build})$ has as result $\emptyset$ because $on(2, 1)$ is not in $\mathcal{B}_0$. The language S allows further strategies definitions by combining them under the usual regular expression constructions like concatenation (";"), union ("|") and iteration ("*", "+"). Thus, given E, E' as already defined strategies, we have that $(E; E')@ms = E'@(E@ms)$, meaning that E' is applied to the result of applying E to ms. The strategy $(E \mid E')@ms$ defined as $(E@ms) \cup (E'@ms)$ means that both E and E' are applied to ms. The strategy $E^+@ms$ is defined as $\bigcup_{i \geq 1}(E^i@ms)$ with $E^1 = E$ and $E^n = E^{n-1}; E$, $E^* = idle \mid E^+$, thus it recursively re-applies itself. It is also possible to define *if-then-else* combinators. The strategy $E\ ?\ E' : E''$ defined as (if $(E@ms) = \emptyset$ then $E'@(E@ms)$ else $E''@ms$ fi) has the meaning that if, when evaluated in a given state term, the strategy E is successful then the strategy E' is evaluated in the resulting states, otherwise E'' is evaluated in the *initial* state. The *if-then-else* combinator is further used to define the following strategies. The strategy *not*(E) = E ? *fail* : *idle* which reverses the result of applying E. The strategy *try*(E) = E ? *idle* : *idle* changes the state term if the evaluation of E is successful, and if not, returns the initial state. The strategy *test*(E) = *not*(E) ? *fail* : *idle* checks the success (resp. the failure) result of E but it does not change the initial state. The strategy $E!$ = E^* ; *not*(E) "repeats until the end", that is, it applies E until no longer possible.

3 Formalising Test Cases

Our test case format is based on two main concepts: observable actions and facts as appearing in belief bases. Our test case format is a kind of black box testing, aimed at testing the observable behaviour of agents. For this reason, we have made a distinction between internal and observable actions. The idea is that the execution of observable actions is visible from outside the agent. Observable actions can be actions the agent executes in the environment in which it operates. In the sequel, we will sometimes omit the adjective "observable" if it is clear from the context.

We introduce a general test case format that allows to express that certain sequences of observable actions are executed, and that the belief bases of the corresponding trace satisfy certain properties. That is, we consider that a test case $\mathcal{T}$ is a pair consisting of a test on actions $\mathcal{T}_a$ and a test on facts $\mathcal{T}_f$. Tests on actions are finite sequences of pairs $(a_0, R_0); \ldots ; (a_n, R_n)$. Each pair (a_i, R_i) consists of a ground observable action a_i to be executed and a set of actions R_i which are allowed to be executed from the current state. The idea is that a test on actions controls the execution of the agent in the sense that only those actions are executed that are in conformance with the action expression. Furthermore, the sets R can be used to identify "bad" traces. If, at a certain state of execution, the agent can perform a *forbidden* action, i.e., which is not allowed by the test case, then the corresponding trace is seen as a counter-example. If no restriction is imposed on the enabled actions we simply use the notation a instead of the pair (a, R). It is then the case that a counter-example can be generated when the agent cannot execute the action indicated by the test. Tests on actions can be derived from a given specification by means of model-checking, for example. We stress that though the specification may be nondeterministic, tests on actions should be deterministic. This is crucial for reducing the state space and makes this approach essentially different from search techniques since it is more efficient. Tests on facts are specified like LTL formulae. For ease of presentation, we work only with a subset of basic formulae:

$$\mathcal{T}_f ::= true \mid fact \mid \neg fact \mid \Box(\neg \bigcirc true \rightarrow fact) \mid fact \wedge fact \mid \Box fact \mid \Diamond fact$$

with $fact$ being a ground atomic formula. Observe that the syntax allows also test cases consisting of tests on actions only, $(\mathcal{T}_a, true)$ which we write shortly as $\mathcal{T}_a$. The LTL formula $\Box(\neg \bigcirc true \rightarrow fact)$ can be used to check if $fact$ holds in the last states, that is, in the states reachable after executing the test on actions. Tests on facts are meant to provide additional counter-examples besides those reflecting forbidden actions. While tests on actions can be automatically derived from the specification (where the tester needs only to choose adequate test cases), using tests on facts requires more effort and intuition from the tester. For illustration purposes, we provide an example of an adequate test on facts by the end of the paper.

To define formally when a BUpL agent satisfies a test we use induction on the structure of test cases. We denote the application of a test $\mathcal{T}$ on an initial configuration (an initial BUpL mental state) ms_0 as $\mathcal{T}@ms_0$. The (set) semantics is defined such that it yields the set of final states reachable through executing the agent restricted by the test, i.e., only those actions are executed that comply with the test. This means that an agent with initial mental state ms_0 satisfies a test $\mathcal{T}$ if $\mathcal{T}@ms_0 \neq \emptyset$, in which case we say that a test $\mathcal{T}$ is *successful*.

$$\mathcal{T}@ms_0 = \begin{cases} \{ms \mid ms_0 \overset{a}{\Rightarrow} ms\}, & \mathcal{T} = (a, R) \wedge R(ms_0) \subseteq R \\ \emptyset, & \mathcal{T} = (a, R) \wedge R(ms_0) \not\subseteq R \\ \mathcal{T}_a^2@(\mathcal{T}_a^1@ms_0), & \mathcal{T} = \mathcal{T}_a^1; \mathcal{T}_a^2 \\ \{ms \mid ms \in \mathcal{T}_a@ms_0 \wedge \Pi_{ms_0}^{\mathcal{T}_a}(ms) \models \mathcal{T}_f\}, & \mathcal{T} = (\mathcal{T}_a, \mathcal{T}_f) \end{cases}$$

The arrow $\overset{a}{\Rightarrow}$ stands for $\Rightarrow\xrightarrow{a}\Rightarrow$, where $\Rightarrow$ denotes the reflexive and transitive closure of $\xrightarrow{\tau}$, and $R(ms)$ denotes the set of actions ready to be executed from ms, i.e., $R(ms) = \{a \mid \exists ms'$ s.t. $ms \overset{a}{\Rightarrow} ms'\}$. The idea behind the definition of the semantics of $(a, R)@ms_0$ is that the test should be successful for ms_0 if action a can be executed in ms_0, while the enabled actions from the states reached by doing a should be a subset of R (defined by $R(ms) \subseteq R$). The result is then the set of mental states resulting from the execution of a, as defined by $\{ms \mid ms_0 \overset{a}{\Rightarrow} ms\}$. We need to keep those mental states to allow a compositional definition of the semantics. In particular, when defining the semantics of $\mathcal{T}_a^1; \mathcal{T}_a^2$ we need the mental states resulting from applying the test $\mathcal{T}_a^1$, since those are the mental states in which we then apply the test $\mathcal{T}_a^2$, as defined by $\mathcal{T}_a^2@(\mathcal{T}_a^1@ms_0)$. In the definition of the semantics of $(\mathcal{T}_a, \mathcal{T}_f)$, by abuse of notation, we use $\Pi_{ms_0}^{\mathcal{T}_a}(ms)$ to denote the paths from ms_0 to ms which are taken while executing $\mathcal{T}_a$. These paths are with respect to observable actions, that is, we abstract from intermediary states reached by doing τ steps. More specifically, each state in a path is reached from the previous by executing an observable action and then executing a number of τ steps until an observable action is again about to be executed (or no transitions are possible). In the initial state, first τ steps can be executed before the first observable action is executed. Tests on facts are thus checked in states resulting from the execution of an observable action and as many τ steps as possible. We call these states *stable*. The definition says that the result of applying the test $(\mathcal{T}_a, \mathcal{T}_f)$ is a subset of $\mathcal{T}_a@ms_0$, namely, those states ms which are reachable after executing $\mathcal{T}_a$ and the corresponding path LTL satisfies $\mathcal{T}_f$.

Our language is such that tests on facts can be omitted. By design, they are meant to provide more expressivity and to give more freedom to the tester. One might raise the issue that inspecting facts classifies our method as white-box testing. However, since facts can be deduced from the effects of actions, our method lies at the boundary between black-box and gray-box testing. In order to define test cases, there is no need to understand the way BUpL agents work (i.e., the internal mechanism for updating states or the structure of repair rules and plans), but only to look at basic actions, which we see as the interface of BUpL agents.

4 Using Rewrite Strategies to Define Test Drivers

In this section we describe how to define *test drivers* for test cases by means of the strategy language S. To give some intuition and motivation, we consider the way one would implement the basic test case a. By definition, the application of this test case to a BUpL mental state ms is the set of all mental states which can be reached from ms by executing the observable action a *after eventually executing τ steps corresponding to internal actions, applying repair rules or making choices*, i.e., after computing closure sets of particular types of rewrite rules. It thus represents a *strategic* rewriting of ms. We are only interested in those rewritings which finally make it possible to execute a. To achieve this at the object-level means to have a procedure implementing the computation of the closure sets. However, the semantics of the application of the test a is independent of the computation of closure sets. Following [5], we promote the design principle that automated deduction methods (e.g., closure sets of τ steps) should be specified *declaratively* as nondeterministic sets of inference rules and not *procedurally*. Depending on the application, specific algorithms for implementing the specifications should be given as *strategies* to apply the inference rules. This has the implication that there is a clear separation between *execution* (by rewriting) at the object-level and *control* (of rewriting) at the meta-level.

In what follows, for ease of reference, we denote by $\mathbb{S}$ (resp. $\mathbb{T}$) the set of strategies (tests) and by s the mapping from tests to test drivers, i.e., $s : \mathbb{T} \rightarrow \mathbb{S}$. Since the definition of tests is inductive, so is the definition of s. We first consider the test drivers for tests on actions:

$$s(\mathcal{T}) = \begin{cases} allow(R)\ ; do(a), & \mathcal{T} = (a, R) \\ s(\mathcal{T}_1)\ ; s(\mathcal{T}_2), & \mathcal{T} = \mathcal{T}_1\ ;\ \mathcal{T}_2 \end{cases}$$

thus sequences of tests map to sequences of strategies. We describe the basic test driver $do(a)$ in more detail. Observe that though tests on actions are deterministic, there are still possibly many executions due to internal actions, choices in plans and repair rules. Thus the test driver must search "all" possible intermediary states which can be reached by doing τ steps. By means of strategies, this is an easy process. By definition, the transitive closure of τ steps, $\Rightarrow$, is $\xrightarrow{\tau}^*$, with τ being one of the label *sum*, *i-act*, or *fail-act* and the corresponding being maximal, in the sense that no τ steps are possible from the last state. Thus, in a naive approach, we could simply consider the following test driver:

$$tauClosure = (\textit{sum} \mid \textit{i-act} \mid \textit{fail-act})!$$

which is clearly implementing $\Rightarrow$. However, though the order of application of the τ steps does not matter when the computation paths are finite, this is no longer the case when considering infinite paths. Consider an extraneous agent program with a plan $p = \textit{i-a} + \textit{i-b}$ where *i-a* is always enabled and *i-b*, on the contrary, is never enabled and a repair rule $(true \leftarrow \textit{i-b})$ which says that whenever there is a failure repair it by executing *i-b*. Applying $tauClosure$ as defined above we obtain two solutions corresponding to a finite path reflecting the choice for executing *i-a* and a divergent path reflecting the choice for executing *i-b* then failing all the time. As long as we are only interested in the "first" solution, then $tauClosure$ is fine, however, if we want to generate also the "next" solution then the computation will not terminate. From this we conclude that we may lose termination if any application order is allowed while we may be able to achieve it if we impose a certain order. Since one source of non-termination is mainly in a sort of "unfairness" with regard to enabled internal actions, a much more adequate test driver is implemented if we enforce the execution of internal actions after eventually applying the sequence (*sum*; *fail-act*). That is, $tauClosure$ becomes:

$$tauClosure = (\textit{try}(\textit{sum}); \textit{try}(\textit{fail-act}); \textit{i-act})!; \textit{try}(\textit{sum}); \textit{try}(\textit{fail-act})$$

We make a few observations with respect to the new definition of $tauClosure$. First, since one might expect multiple sum and fail applications before an internal action is executed, it is no longer immediately clear that $tauClosure$ faithfully implements $\Rightarrow$. We present a correctness proof by the end of the section. Second, because we use the sequential strategy, we need to surround both *sum* and *fail-act* by *try* blocks. Otherwise, if either one of them were not applicable, i.e., the current plan is not a sum and the "head" action is enabled, then the strategy (*sum* ; *fail-act*; *i-act*) fails which is not what we want. By means of the parametrised strategy *try* the initial state is preserved in the case that *sum* or *fail-act* fails. Third, we order *fail-act* after *sum* because if we were to use the strategy (*try*(*sum* | *fail-act*) ; *i-act*) and the current plan is a sum of two failing plans, then the whole strategy fails though there might have been possible to replace the failing plans with a "good" plan by applying *fail-act*. Fourth, we require that repair rules are of a particular format, that is $\phi \leftarrow p$ with p not containing the sum operator. This is in order to avoid situations where the application of *fail-act* entails the application of *sum* which entails the application of *fail-act* and so forth (that is, non-terminating strategies (*sum* ; *fail-act*)!). Such format does not result in the loss of expressivity since having one repair rule $\phi \leftarrow p_1 + p_2$ is equivalent to having two repair rules $\phi \leftarrow p_i$, with $i \in \{1, 2\}$. Fifth, the use of strategies can be tricky. Though one might be tempted to use the strategy *try*(*sum ; fail-act*) instead of *try*(*sum*) ; *try*(*fail-act*), the first one is "wrong", meaning that if *fail-act* is not applicable after *sum* then the original state is returned instead of the one reached by applying *sum*. The last observation is with respect to the normalisation strategy. Since "!" returns the state previous to the one that failed, we need to apply again *try*(*sum*); *try*(*fail-act*) to make sure that from the resulting state no τ steps can be taken.

By means of $tauClosure$, the definition of $do(a)$ is straight-forward:

$$do(a) = tauClosure; o\text{-}act[o\text{-}a \leftarrow a]; tauClosure$$

which corresponds to the definition of $\overset{a}{\Rightarrow}$. We note that $tauClosure$ is no longer applicable when *i-act* fails after *sum* and *fail-act* have been applied. This means that the only possible scenario is that the head of the current plan is an observable action. If this action is in fact a, then *o-act*[*o-a* $\leftarrow$ a] is successful, otherwise it fails.

The definition of the strategy $allow(R)$ makes use of the *match* construction:

$$allow(R) = \textit{match ms s.t. ready}(ms) \subseteq R$$

which means that $allow(R)$ succeeds if the current mental state satisfies the condition $ready(ms) \subseteq R$, where *ready* is a function defined on BUpL mental states. This function is implemented such that it returns the set of actions ready to be executed. For simplicity, we do not detail its implementation but briefly describe it. Recall that BUpL mental states are pairs of belief bases and plans. The function *ready* reasons on possible cases. If the current plan is a sum of plans then *ready* is called recursively. Otherwise, depending on the action a in the head of the plan, either a is enabled and so the function *ready* returns a, or a fails and the function *ready* recursively considers all the plans that can substitute the current one, that is, it recursively analyses the active repair rules.

So far, we have focused on tests on actions $\mathcal{T}_a$. We focus now on the general test cases$(\mathcal{T}_a, \mathcal{T}_f)$. We begin by first considering the test driver implementing the test case for checking whether $fact$ is in the last states reachable by executing $\mathcal{T}_a$, i.e., $s((\mathcal{T}_a, \Box(\neg\bigcirc true \rightarrow fact)))$. For this, we consider an auxiliary strategy $check(fact)$:

$$check(fact) = \mathit{match}(\mathcal{B}, p)\ \mathit{s.t.}\ fact \in \mathcal{B}$$

which is successful if $fact$ is in the belief base from the current state. With this strategy we can define $s((\mathcal{T}_a, \Box(\neg\bigcirc true \rightarrow fact)))$ simply as $s(\mathcal{T}_a); check(fact)$. We can further use $check(fact)$ for defining test drivers working with $\neg fact$ as $not(check(fact))$ and with $fact_1 \wedge fact_2$ as $check(fact_1); check(fact_2)$. The cases with respect to the temporal formulae are defined by case analysis. We present only the implementation of the non-trivial ones:

$$\begin{aligned} s(((a,R);\mathcal{T}_a, \Diamond fact)) &= \mathit{check}(fact)\ ?\ s((a,R);\mathcal{T}_a) : s((a,R))\ ;\ s((\mathcal{T}_a, \Diamond fact)) \\ s(((a,R);\mathcal{T}_a, \Box fact)) &= \mathit{check}(fact)\ ;\ s((a,R))\ ;\ s((\mathcal{T}_a, \Box fact)) \end{aligned}$$

which illustrates that the main difference between them is that for $\Diamond fact$ we stop checking $fact$ as soon as we reached a state where $fact$ is in the belief base; from this state we continue with only executing the test on actions. However, for $\Box fact$ we check until the end.

Observe that the semantics of the testing language was defined such that we have a separation between implementing test drivers and *reporting* the results. This is important since running a test driver should be orthogonal to the interpretation and the analysis of the possible output. One plausible and intuitive interpretation is the following one. When the test driver is successful the tester has the confirmation that the test case corresponds to a "good" trace in the agent program. When the test driver fails, the tester can further define new strategies to obtain more information. Consider, as an example, a strategy returning the states previous to the failure. More sophisticated implementations like gathering information about traces instead of states are left to the imagination of the reader. These traces correspond to the shortest counter-examples. This follows from the semantics of the testing language. At each action execution a check is performed whether forbidden actions are possible. If this is the case, then the test fails.

Assuming that we fix an interpretation of the results as above, we proceed by showing that test drivers are partially correct and complete with respect to the definition of test cases.

Definition 1. *Given a test case $\mathcal{T}$ and the corresponding test driver $s(\mathcal{T})$, we say that the application of $s(\mathcal{T})$ is correct, if, on the one hand, successful executions of the test driver are successful applications of the test case, and if, on the other hand, the test driver fails then test case also fails. Similarly, s is complete if (un)successful applications of the test case $\mathcal{T}$ are (un)successful executions of the test driver $s(\mathcal{T})$.*

Before stating the main result, we show two helpful lemmas. Recall that, at each repetition step, the strategy $tauClosure$ tries to apply *sum* and *fail-act* only once. Intuitively, this is sufficient for the following reason. Let us first consider *fail-act*: if, on the one hand, after the application of *fail-act* no action can take place then applying *fail-act* again can do no good, since nothing changed; if, on the other hand, after applying once *fail-act* the first action of the new plan can be executed then we are done, the faulty plan has been repaired. From this, we have the following lemma:

Lemma 1. *The strategy try(fail-act) is idempotent, i.e., for any ms try(fail-act)^2 $@ms$ = try(fail-act) $@ms$.*

Proof. Let Res = *try*(*fail-act*) $@ms$. Any $ms' \in Res$ different from ms is the result of applying the rewrite rule *fail-act* so it has the form $(\mathcal{B}, p\theta)$, where $\phi \leftarrow p \in \mathcal{R}$ (the set of repair rules) and $\theta \in Sols(\mathcal{B} \models \phi)$. If *fail-act* were again applicable for such ms', the resulting term ms'' is also of the same form since $\mathcal{R}$ is fixed and $\mathcal{B}$ does not change. Thus, any ms'' is already an element of Res and so *try*(*fail-act*) $@Res$ = Res. □

An analogous reasoning works also for *sum*. Taking into account that the "+" operator is commutative and associative and that the ";" operator is associative, a *normal form* (i.e., sum of plans with only sequence operators) always exists. Since *sum* is applied to states where the plans are reduced to their normal form we have that states with *basic* plans will always be in the result of trying to apply *sum* more than once.

Lemma 2. *Given a mental state ms we have that sum! $@ms \subseteq$ try(sum) $@ms$.*

Proof. We only consider the interesting case where *sum* is applicable, that is, when *try*(*sum*) $@ms$ = *sum* $@ms$. Let $ms = (\mathcal{B}, p)$ where p has been reduced to the form $\sum_{i=1}^{n} p_i$ and p_i are basic plans (composed by only the ";" operator). Since *sum* is commutative, we have that *sum* $@ms = \{(B, \sum_{j=1}^{k} p_{i_j}) \mid \forall k, i_j \in \{1, \ldots, n\}\}$, i.e., any possible combination of p_i. On the other hand, *sum*! $@ms = \{(B, p_i) \mid i \in \{1, \ldots, n\}\}$ which is clearly included in *sum* $@ms$. □

Theorem 1 (Partial Correctness & Completeness). *Given ms a mental state, $\mathcal{T}$ a test case we have that $s(\mathcal{T})@ms$ = $\mathcal{T}@ms$.*

Proof. We consider only the strategy do. The proof for the compositions follows from the definitions of the strategies. We proceed, by showing, as usually, a double inclusion. "$\subseteq$": By the definition of $do(a)$ we have that the result of applying it on ms is:

$$Res = tauClosure \ @ \ \underbrace{(\textit{o-act}[\textit{o-a} \leftarrow a] \ @ \ \underbrace{tauClosure \ @ \ ms}_{Res'})}_{Res''}$$

If the normalisation strategy "!" from the definition of $tauClosure$ terminates, then by definition, there exists an $i \geq 0$ s.t.:

$$Res_i = \textit{i-act} \ @ \ (\textit{try}(\textit{fail-act}) \ @ \ (\textit{try}(\textit{sum})@Res_{i-1}))$$

and for any $ms_i \in Res_i$ we have that *i-act* $@$ (*try*(*fail-act*) $@$ (*try*(*sum*) $@ms_i$)) is empty (1). Thus, we can construct the computation:

$$ms_0 \xrightarrow{\tau}^{*} ms_1 \xrightarrow{\tau}^{*} \ldots \xrightarrow{\tau}^{*} ms_{i-1} \xrightarrow{\tau}^{*} ms_i$$

where we take $ms_j \in Res_j$ with $j \leq i$, ms_0 as ms and * denotes at most 3 τ steps, corresponding to the 3 possible rule labels for τ steps. By the definition of $tauClosure$, Res' is the union of *try*(*fail-act*) @ (*try*(*sum*)@Res_i). This implies that any $ms' \in Res'$ is obtained from a ms_i after eventually applying *sum* and *fail-act*. From (1) we have that from ms' it is not possible to apply *i-act*. Furthermore, by the lemmas, whatever state can be reached from ms' by *sum* and *fail-act* is already in Res'. Thus, $ms \Rightarrow ms'$. By definition, Res'' is empty iff *o-act*[*o-a* $\leftarrow a$] @ms' fails for any element $ms' \in Res'$. That is, if Res'' is empty then $ms \overset{a}{\not\Rightarrow} ms'$ and thus $a@ms$ returns the empty set.
If Res'' were not empty, then for any element ms'' contained in it we have that $ms' \xrightarrow{a} ms''$, thus $ms \Rightarrow\xrightarrow{a} ms''$. Similarly, for any element $ms_f \in Res$ we have $ms'' \Rightarrow ms_f$ and from this we can conclude that $ms \overset{a}{\Rightarrow} ms_f$, thus ms_f is also an element of $a@ms$.
"$\supseteq$": By the definition of $\Rightarrow$ we have that, if no τ divergence, then there exists a $k \geq 0$ s.t. $ms \xrightarrow{\tau^k} ms_1$ and $ms_1 \not\rightarrow$. The trace τ^k can be divided in m packages of the form:

$$\sigma_m = (\textit{sum}^{i_m}; \textit{fail-act}^{j_m}; \textit{i-act}^{l_m})^m,$$

with $\sum_m (i_m + j_m + l_m) * m = k$. By the lemmas we have that *sum*^{i_m};*fail-act*^{j_m}; *i-act* is obtained by applying the strategy *try*(*sum*); *try*(*fail-act*); *i-act* (2). As for *i-act*$^{l_m - 1}$, it is obtained by (*try*(*sum*); *try*(*fail-act*); *i-act*)$^{l_m - 1}$ (3). If successive applications of *i-act* are possible then neither *fail-act* nor *sum* is applicable (at most one of *i-act*, *fail-act*, *sum* is enabled at a time) thus trying to applying them is harmless, i.e., does not change the state. Repeating m times the same argument from (2) + (3) and taking into account that we have that sequences σ_m where l_m is 0 are mapped to *try*(*sum*); *try*(*fail-act*) we can derive that $ms_1 \in tauClosure@ms$ (4).
If $ms_1 \xrightarrow{a} ms'$, then $ms' \in$ *o-act*[*o-a* $\leftarrow a$] @ms_1. Applying a similar reasoning for ms' we obtain (4'): $ms_2 \in tauClosure@ms'$. In consequence, we have that if $ms \overset{a}{\Rightarrow} ms_2$ then also $ms_2 \in do(a)@ms$.
If $ms_1 \overset{a}{\not\rightarrow} ms'$, then *o-act*[*o-a* $\leftarrow a$] @ms_1 fails, thus this is also the case for $do(a)$. $\square$

Observe that in our proof we consider only finite computations. Thus, infinite computations do not violate the result. Since τ divergence is undecidable for BUpL agents, we cannot provide conditions such that test drivers terminate for all test cases. The most we can do, with respect to divergent computations, is to state the following proposition as a consequence of the above result:

Corollary 1 (Divergence). *If the application of $s(\mathcal{T})$ diverges then so does $\mathcal{T}$.*

5 A Running Example

The BUpL builder described in Figure 1 has a small number of states. Thus, verification by model-checking is feasible. We provide now an illustration of the utility of testing. Consider the agent from Figure 3[6]. It is meant to implement the specification "the agent should always construct towers, the order of the blocks is not relevant, however each tower should use more blocks than the previous, and additionally, the length of the towers must be an even number"[7] (for example, 21, 4321 are "well-formed" towers).

[6] The code presents only the constructions which are additional to the ones from Figure 1.
[7] Since it is just meant to be an illustration, the notion of specification is merely informal.

$$
\begin{aligned}
\mathcal{A} = \{\ & incLength(x) = (length(x), \{\ \neg length(x), length(x+1)\ \}),\\
& addBlock(x) = (\neg on(x,0), \{\ on(x,0), clear(x)\ \}),\\
& setMax(x,y) = (max(y), \{\ \neg max(y), max(x)\ \}),\\
& finish(x,y) = (\neg done(x) \wedge done(y), \{\ \neg(done(y)), done(x)\ \})\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{P} = \{\ & build(n,c) = move(c-n, 0, c-n-1); incLength(c-n-1); build(n-1,c)\\
& generate(x,y) = addBlock(x);\ generate(x-1,y),\\
& p_0(x,y) = setMax(x,y); generate(x,y))\ \}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{R} = \{\ & length(x) \wedge max(y) \wedge (x \leq y) \leftarrow build(y, y+x-1),\\
& length(x) \wedge max(x) \wedge done(y) \wedge (x \geq y) \leftarrow finish(x,y); \bot,\\
& max(x) \wedge done(x) \leftarrow setMax(x+2,x), generate(x+2,x)\ \}
\end{aligned}
$$

Fig. 3. A BUpL Builder with Infinite State Space

The agent is designed such that it always builds a higher tower. The example can be understood as a typical agent with *maintenance* goals. Since the number of its mental states continuously increases, instead of model-checking, we test it. For illustration purposes, the implementation of the agent is on purpose faulty: assuming a correct initialisation, the agent program does not perform a sanity check with respect to the parity of X before adding the fact $done(X)$ to signal that it constructed a tower X.

Thanks to the fact that the strategy language S has been incorporated into the Maude system, it was relatively easy to extend the implementation from [1]. In this way, we provide a testing framework as alternative to the model-checking facility. We have experimented with different test cases which we applied to the Maude prototype of the BUpL builder. For example, we have considered the test whether $done(2)$ appears in the belief base after executing $move(2,0,1)$. To implement it, we only needed to apply the strategy $do(move(2,0,1))$; $check(done(2))$. The application of the strategy failed, meaning that the agent is not conformant with the test case. On the contrary, the application succeeded when the correct agent program is tested. We have run our tests on a Fedora 10 system (Kernel linux 2.6.27.12-170.2.5.fc10.x86_64) with an AMD Athlon(tm) 64 Processor 3500+ and 1 GB memory. The process of executing the BUpL builder with respect to the test case $do(move(2,0,1))$; $check(done(2))$ took 1876ms and generated 35745 rewrites. The number of rewrites is high mainly because the strategy language is implemented at the meta-level and because computations at the meta-level involve many rewrite steps. For the correct agent, the output generated by Maude illustrates that the strategy has succeeded and that the resulting state reflects that $done(2)$ has been updated to the belief base and that the current tower is 21. By means of the Maude command `next` we can further see that there are no more solutions (corresponding to faulty executions). More examples and the actual Maude code (also including more test case implementations) can be downloaded from our website `http://homepages.cwi.nl/~astefano/agents/bupl-strategies.php`.

6 Conclusions and Future Work

In this paper, we focused on two aspects. First, we have provided a formalisation for testing BUpL agents. Second, we have introduced rewrite strategies to define test drivers that implement test cases. For simplicity, we have considered testing individual agents. Generalising our current results to multi-agent systems should be easy in a *particular framework* as the one proposed in [2]. There, the interaction between agents is achieved not by means of communication but by *action-based coordination* mechanisms. The advantage of this approach is that the framework is compositional and thanks to this, the verification problem (by model-checking) of the whole system can be reduced to the verification of individual agents. Using the compositionality result we can obtain the same reduction when we consider *testing* instead of *model-checking*.

References

1. L. Astefanoaei and F. S. de Boer. Model-checking agent refinement. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 705–712. IFAAMAS, 2008.
2. L. Astefanoaei, F. S. de Boer, and M. Dastani. The refinement of choreographed multi-agent systems. In *Proceedings of the 9th International Workshop on Declarative Agent Languages and Technologies (DALT)*. LNAI, 2009. to appear.
3. R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*. Springer, 2005.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.
5. S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 174(11):3–25, 2007.
6. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proceedings of the 4th Workshop on Rewriting Logic and its Applications (WRLA)*, volume 71 of *ENTCS*. Elsevier, 2002.
7. 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 (AAMAS)*, 2(4):357–401, 1999.
8. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier, 2000.
9. B. Meyer. Seven Principles of Software Testing. *IEEE Computer*, 41(8):99–101, 2008.
10. D. C. Nguyen, A. Perini, and P. Tonella. A Goal-Oriented Software Testing Methodology. In *Agent Oriented Software Engineering (AOSE)*, pages 58–72, 2007.
11. T.-F. Serbanuta, G. Rosu, and J. Meseguer. A rewriting logic approach to operational semantics (extended abstract). *Electronic Notes in Theoretical Computer Science (ENTCS)*, 192(1):125–141, 2007.
12. M. Wooldridge. Agent-based software engineering. *IEEE Proceedings Software Engineering*, 144(1):26–37, 1997.
13. Z. Zhang, J. Thangarajah, and L. Padgham. Automated unit testing intelligent agents in PDT. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1673–1674. IFAAMAS, 2008.