Aplib: Tactical Agents for Testing Computer Games*

I. S. W. B. Prasetya[1] [0000−0002−3421−4635], Mehdi Dastani[1] [0000−0002−3421−4635], Rui Prada[2] [0000−0002−5370−1893], Tanja E. J. Vos[3] [0000−0002−6003−9113], Frank Dignum[4] [0000−0002−5103−8127], and Fitsum Kifetew[5] [0000−0003−1860−8666]

[1] Utrecht University, the Netherlands, `s.w.b.prasetya@uu.nl`
[2] Inst. de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento, Portugal
[3] Universidad Politecnica de Valencia (Spain) and Open University (The Netherlands)
[4] Umeå University, Sweden
[5] Fondazione Bruno Kessler, Italy

Abstract. Modern interactive software, such as computer games, employ complex user interfaces. Although these user interfaces make the games attractive and powerful, unfortunately they also make them extremely difficult to test. Not only do we have to deal with their functional complexity, but also the fine grained interactivity of their user interface blows up their interaction space, so that traditional automated testing techniques have trouble handling it. An agent-based testing approach offers an alternative solution: agents' goal driven planning, adaptivity, and reasoning ability can provide an extra edge towards effective navigation in complex interaction space. This paper presents aplib, a Java library for programming intelligent test agents, featuring novel tactical programming as an abstract way to exert control over agents' underlying reasoning-based behavior. This type of control is suitable for programming testing tasks. Aplib is implemented in such a way to provide the fluency of a Domain Specific Language (DSL). Its embedded DSL approach also means that aplib programmers will get all the advantages that Java programmers get: rich language features and a whole array of development tools.

Keywords: automated game testing · AI for automated testing · intelligent agents for testing · agents tactical programming · intelligent agent programming

1 Introduction

With the advances of technologies, computer games have become increasingly more interactive and complex. Modern computer games improve realism and user experience by allowing users to have fine grained control/interactions. A downside of this development is that it becomes increasingly difficult to test computer games. For example, to test that a computer game would maintain the correctness invariant of a certain family of states, the tester will first need to operate the game to bring it to at least one of such states. This often requires a long series of fine grained interactions with the game. Only

* This work is supported by European Union's Horizon 2020 research and innovation programme under grant agreement No 856716 Project iv4XR (Intelligent Verification/Validation for Extended Reality Based Systems).

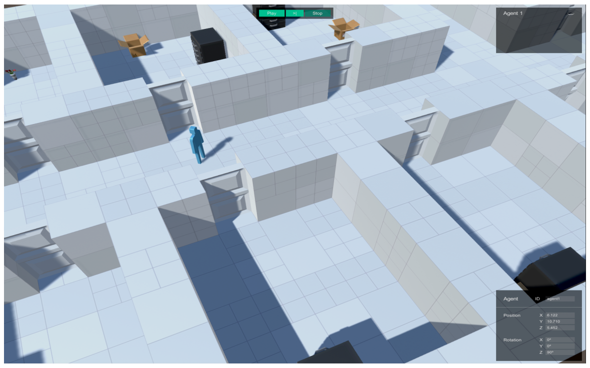

Fig. 1. *A 3D game called Lab Recruits where* aplib *were deployed aid testing.*

then the tester can check if the said invariant does hold in that state. Such a test is hard, error-prone, and fragile to automate. Consequently, many game developers still resort to expensive manual play testing. Considering that the game industry is worth over 100 billions USD, speeding up testing by effectively automating manual testing tasks is a need that cannot be ignored.

As indicated above, a common manual expensive test related task is to bring the game under test to a certain state of interest (goal state), either because we want to check if the state is correct, or because we need to do a specific action on this state that is required for the given test scenario. In principle this task is a search problem, for which solutions exist. However, in the context of computer games the problem is challenging. A game often employs randomness and it often consists of many entities that interact with each other and with the user. Some interactions might be cooperative while others can be adversarial. These and other factors lead to a vast and fine grained interaction space which is hard to deal with for the existing automated testing techniques such as search based [29, 20], model based [15, 43], or symbolic [4, 42]. The key to handle such a space, we believe, is to have an approach that enables the programming of *domain reasoning* to express which parts of the interaction space of a particular game are relevant to consider, and likewise what kinds of plans (for reaching given goal states) are needed. This allows the underlying test engine to focus its search on the parts of the interaction and plan spaces that semantically matter. We propose to base such a solution on a *multi-agent approach* since autonomous distributed planning and reasoning based interactions with environments are already first class features.

Contribution. This paper presents aplib[6], a Java library for programming intelligent agents suitable for carrying out complex testing tasks. They can be used in conjunction with Java testing frameworks such as JUnit, e.g. to collect and manage test verdicts. Figure 1 shows a 3D game we use as a pilot where aplib was used to automate testing (we will also use it later as a running example). Aplib features BDI (Belief-Desire-Intention [23]) agents and adds a novel layer of *tactical programming* that provides an abstract way to exert control on agents behavior. Declarative reasoning rules express when ac-

[6] "Agent Programming Library", `https://iv4xr-project.github.io/aplib/`.

tions are allowed to execute. Although in theory just using reasoning is enough to find a solution (a plan that would solve the given goal state) if given infinite time, such an approach is not likely to be performant enough. For testing, this matters as no developers would want to wait for hours for their test to complete. The tactical layer allows developers to program an imperative control structure over the underlying reasoning-based behavior, allowing them to have greater control over the search process. So-called *tactics* can be defined to enable agents to strategically choose and prioritize their short term actions and plans, whereas longer term strategies are expressed as so-called *goal structures*, specifying how a goal can be realized by chosing, prioritizing, sequencing, or repeating a set of subgoals.

While the concept of a hierarchical goal is not new, e.g. it can be solved by Hierarchical Task Networks (HTN) and Behavior Trees (BT), or can be encoded directly as BDI reasoning rules [9], aplib allows it to be expressed in terms of imperative programming idioms such as **SEQ** and **REPEAT**, which are more intuitive for programming control. The underlying reasoning based behavior remains declarative. Our tactical programming approach is more similar to tactical programming in interactive theorem proving, used by proof engineers to script proof search [40, 12, 22]. The use of this style in BDI agents and for solving testing problems is as far as we know new.

As opposed to dedicated agent programming languages [37, 41] aplib offers a Domain Specific Language (DSL) *embedded* in Java. This means that aplib programmers will program in Java, but they will get a set of APIs that give the fluent appearance of a DSL. In principle, having a native programming language for writing tests is a huge benefit, but only if the language is rich enough and has enough tool and community support. Otherwise it is a risk that most companies will be unwilling to take. On the other hand, using an embedded DSL means that the programmers have direct access to all the benefit the host language, in this case Java: its expressiveness (OO, λ-expression etc.), static typing, rich libraries, and wealth of development tools.

Paper structure. Section 2 first introduces the concept of testing tasks; these are the tasks that we want to automate. Section 3 explains the basic concepts of aplib agents and shows examples of how to create an agent with aplib and how to write some simple actions. Section 4 introduces the concept of goal structures, to express complex test scenarios, and our basic constructs for tactical programming. The section also explains aplib's 'deliberation cycle', which necessarily deviates from BDI's standard due to its tactical programming. Large scale case studies are still future work. However, Section 5 will briefly discuss our experience so far. Section 6 discusses related work, and finally Section 7 concludes and mentions some future work.

2 Testing Task

This section will introduce what we mean by a 'testing task', and what 'automating' it means. The typical testing task that we will consider has the form:

$$\underbrace{\phi}_{\text{situation}} \Rightarrow \underbrace{\psi}_{\text{invariant}} \qquad (1)$$

where ϕ is a state predicate characterizing a situation and ψ is a state predicate that is expected to hold on all instances of the situation ϕ (that is, on all states satisfying

ϕ). We call ψ an *invariant*, which is the term used by Ernst et al. [16] to refer to a predicate that is expected to hold at a certain control location in a program, e.g. when a program enters its loop, or when it exits; ϕ would then be a predicate that characterizes the control location of interest. This concept generalizes the well known pre- and post-conditions. E.g. if ϕ captures the exit of a method m, the invariant ψ then describes m's post-condition.

Since game testing typically has to be done in the so-called blackbox setup [3] where we abstract away from the source code (because it would otherwise be too complex), and hence also away from concepts such as programs' control location, we further generalize Ernst et al. by allowing ϕ to describe a family of game states that are semantically meaningful for human users; we call this a *situation*. For example ϕ could characterize the situation where a certain interactable game element, e.g. a switch, is visible, and ψ could then express the expectation that the switch should be in its 'off' state.

Since ϕ can potentially describe a very large, even infinite, set, the specification $\phi \Rightarrow \psi$ is tested by sampling a finite number of states, and then checking whether the invariant ψ holds in these states. Obviously such tests are only *relevant* when applied on sample states that satisfy the situation ϕ. Getting the game into a relevant state for testing $\phi \Rightarrow \psi$ is a non-trivial task for a computer. Since a game typically starts in specific initial states, it first *needs to be played* to move it to any specific other state. Consequently, when we want to automate the testing of $\phi \Rightarrow \psi$, the hard part is typically not in checking its invariant part, but in finding relevant states to test the implication.

Playing a game can be seen as the execution of a sequence of actions, e.g. moving up or down, interacting with some in-game entity, etc. The set of available actions might be different on different states. We will call a sequence of actions a *plan*. A *solution* is a plan that, when executed, would drive the game under test to a state relevant for $\phi \Rightarrow \psi$. In manual testing, a human is employed to search for such a solution. There are tools that can be used to record a script that can execute the plan and replay it whenever we need to re-test the corresponding situation. A major challenge, however, with script-based test automation is the manual effort required for maintaining the scripts when they break [2]. If the game designers introduce even a small change in a the game layout (e.g. an in-game door is moved to a different position), which happens very often during the development, a recorded script would typically break. Moreover, games are non-deterministic due to all sorts of random behavior (e.g. random moves by computer controlled enemies, or randomness due to timing effect). This makes such automation scripts for games even more fragile.

By 'automated testing' of $\phi \Rightarrow \psi$ we mean to replace the human effort by letting an *agent* search for solutions. This is a *search problem*: the space of possible plans is searched to find at least one that would solve ϕ. We can define the *robustness* of an automated test as how well it can cope with the non-determinism of the system under test. Since agents are typically reactive to the environment, agent-based test automation can thus be expected to be robust; this will be discussed later in Section 4.3.

Testing tasks can be generalized to test 'scenarios':

$$\underbrace{\phi_0 \,;\, \ldots \,;\, \phi_{k-1}}_{\text{scenario}} \Rightarrow \underbrace{\psi}_{\text{invariant}} \qquad (2)$$

Each ϕ_i is a state predicate describing a situation. The sequence $\phi_0; \ldots; \phi_{k-1}$ describes a scenario where executions of the game under test passes through the states satisfying each ϕ_i in the same chronological order as the sequence. In the state where ϕ_{k-1} is satisfied, the invariant ψ is expected to hold. For example, if developers employ UML Use Cases, these can be converted to the above form: each flow in a use case can be translated to a scenario, and its post condition to ψ. Testing a scenario is not fundamentally harder than testing a situation, since the next situation ϕ_{i+1} in the scenario defines the same kind of search problem as we had in situation testing where ϕ_i describes the starting states for the search.

3 Aplib Agency

This section will introduce our agent programming framework aplib and show how to use it to automate testing tasks.

Preliminary: Java functions. Since Java 8, functions can be conveniently formulated using so-called λ-expressions. E.g. the Java expression:

$$x \rightarrow x{+}1$$

constructs a nameless function that takes one parameter, x, and returns the value of $x{+}1$. Unlike in a pure functional language like Haskell, Java functions can be either *pure* (has no side effect) or impure/effectful. An *effectful* function of type $C{\rightarrow}D$ takes an object $u{:}C$ and returns some object $v{:}D$, and may also alter the state of u.

Importantly, a λ-expression can be passed as a parameter. Since as a function a λ-expression defines behavior, passing it as a parameter to a method or object essentially allows us to inject new behavior to the method/object. This allows us to extend the behavior of an agent without having to introduce a subclass. While the latter is the traditional OO way to extend behavior, it would clutter the code base if we plan to create e.g. many variations of the same agent. Our use of λ-expressions to inject behavior is essentially a generalization of the well-known Strategy Design Pattern [21].

3.1 Agent, Belief, and Goal

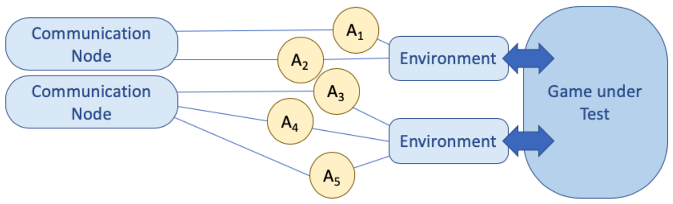

Fig. 2. *Typical deployment of* aplib *agents. A_i are agents, controlling the game under test through an interface called* Environment. *A* communication node *allows connected agents to send messages to each other.*

Figure 2 illustrates the typical way aplib agents are deployed. As common with software agents, aplib agents are intended to be used in conjunction with an external environment (in our case, this is the game under test) which is assumed to run independently. In aplib, the term 'Environment' refers, however, to a Java interface between the agents and the game. Aplib agents do *not* directly access nor control the game. Having the Environment in between keeps aplib neutral with respect to the technology used by the game under test. Developers do have to provide an implementation of this interface for each game what they want to test with aplib. This indeed requires effort, but it is a one-off investment, after which the developers would benefit from aplib's automation for the rest of the development process, as well as that of future versions of the game. The minimum functionality that an Environment should provide is a function to let an agent obtain relevant information about the current game state visible to it, and to send a command to some in-game entity that it is allowed to control.

Multiple agents can be deployed if the game is multi-player. In such a setup, agents may want to work together. A group of agents that wish to collaborate can register to a 'communication node' (see Fig. 2). This enables them to send messages to each other (singlecast, broadcast, or role-based multicast).

BDI with goal structure. As typical in BDI (Belief-Desire-Intent) agency, an aplib agent has a concept of belief, desire, and intent. An agent's state reflects its belief. It contains information on the current state of the game under test. Such information is a 'belief' because it may not be entirely factual. E.g. the game may only be willing to pass current information of in-game entities *in the close vicinity* of the agent. So, the agent's information on far away entities might over time become obsolete. The agent can be given a *goal structure*, defining its desire. Unlike flat goal-based structures used e.g. in 2APL [9] and GOAL [24], in this paper we employ a richly structured goal structure, with different nodes expressing different ways a goal could be achieved through its subgoals; more on this will be discussed Section 4. Abstractly, an aplib agent is a tuple:

$$A \ = \ (s, E, \Pi, \beta)$$

where s is an object representing A's state and E is its environment.

Π is a goal structure, e.g. it can be a set of goals that have to be achieved sequentially. Each goal is a pair, let's denote it with $g \dashleftarrow T^*$, where g is the goal itself and T is a 'tactic' intended to achieve g. In BDI terms, T reflects intention. When the agent decides to work on a goal $g \dashleftarrow T^*$, it will commit to it: it will apply T *repeatedly* over multiple execution cycles until g is achieved, or the agent has used up its 'budget' for g.

The β in the tuple represents the agent's computing **budget**. Budget is used to control how long the agent should persist on pursuing its current goal. Executing a tactic consumes some budget. So, this is only possible if $\beta{>}0$. Consequently, a goal will automatically fail when β reaches 0. Budget plays an important role when dealing with a goal structure with multiple goals as the agent will have to decide how to divide the budget over different goals. This will be discussed later in Section 4.

Example. Figure 3 shows a scene in a game called *Lab Recruits*[7]. Imagine that we want to test that the door (white circled) works (it can be opened). Two buttons (red circled) are present in the room. In a correct implementation, the door can be opened by activating the button closest to the door. A player (yellow circled) can activate a button

[7] https://github.com/iv4xr-project/labrecruits

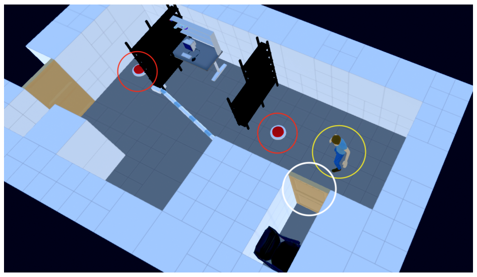

Fig. 3. *A setup where we have to test that a closet door (circled white) can be opened.*

by moving close to it and interacting with it. Suppose the door is identified by door_1 and its corresponding button button_1. The testing task above can be specified as follows:

$$\underbrace{\mathsf{button}_1 \text{ is active}}_{\text{situation}} \Rightarrow \underbrace{\mathsf{door}_1 \text{ is open}}_{\text{invariant}} \tag{3}$$

Figure 4 shows how we create a test agent named Smith to perform the aforementioned testing task. First, lines 1-3 show the relevant part of the environment the agent will use to interface with the Lab Recruits game; it shows the primitive commands available to the agent. The method $\mathsf{interact}(i, j)$ will cause an in-game character with id i (this would be the character controlled by the agent) to interact with another in-game entity with id j (e.g. a button). The method also returns a new 'Observation', containing information on the new state of game-entities in the visible range of i. The method $\mathsf{moveToward}(i, p, q)$ will cause the character i to move *towards* a position q, given that p is i's current position. Simply teleporting to q is not allowed in most games. Instead, the method will only move i some small distance towards q (so, it may take multiple update cycles for a to actually reach q). The method also returns a new observation.

Line 5 creates an empty agent. Lines 11-13 configure it: line 11 attaches a fresh state to the agent; then, assuming labrecruitsEnv is an instance of LabRecruitsEnv (defined in lines 1-3), line 12 hooks this environment to the agent. Line 13 assigns a goal named Π to the agent. The goal is defined in lines 6-10, stating that the desired situation the agent should establish is one where the in-game button_1 is active (line 7). Line 9 associates a tactic named $\mathsf{activateButton_1Tac}$ to this goal, which the agent will use to achieve the latter. Line 10 lifts the defined goal to become a goal structure. More precisely, line 6 creates a 'test-goal'. An ordinary goal, created using a constructor named **goal** rather than **testgoal**, simply formulates desired states to be in. A test-goal *additionally* specifies an invariant (line 8). It formulates a testing task as discussed in Section 2. E.g. lines 7 and 8 formulate the testing task in (3). When the goal part is achieved, the invariant will be tested on the current agent state. If this returns true, the test passes, otherwise it fails. Its automation is provided by the tactic $\mathsf{activateButton_1Tac}$ that should specify some strategy to go towards the button and activate it.

3.2 Action (Elementary Tactic)

A tactic is made of 'actions', composed hierarchically to define a goal-achieving strategy. Such composition will be discussed in Section 4.1. In the simple case though, a

```
class LabRecruitsEnv extends Environment {                                    1
   Observation interact(String id1, String id2) ...                           2
   Observation moveToward(String id, Vec3 p, Vec3 q) ...                      3
}                                                                             4
var Smith = new TestAgent() ;                                                 5
var Π = testgoal("g",Smith)                                                   6
       . toSolve   (s → isActive(s.getEntity("button1")))                     7
       . invariant(s → isOpen(s.getEntity("door1")))                          8
       . tactic(activateButton1Tac)                                           9
       . lift() ;                                                            10
Smith  . withState(new AgentState())                                         11
       . withEnvironment(labrecruitsEnv)                                     12
       . setGoal(Π)                                                          13
       . budget(200)                                                         14
```

Fig. 4. *Creating an agent named* Smith *to test the Lab Recruits game. The code is in Java, since* Aplib *is a DSL embedded in Java. The notation $x{\to}e$ in line 7 is Java lambda expression defining a function, in this case a predicate defining the goal.*

tactic is made of just a single action. An *action* is an effectful and guarded function over the agent's state. The example below shows the syntax for defining an action.

$$\textbf{var}\,\alpha \;=\; \textbf{action}(\text{"}id\text{"})\;.\;\underbrace{\textbf{do}_2(f)}_{\text{behavior}}\;.\;\underbrace{\textbf{on}_{\_}(q)}_{\text{guard}} \tag{4}$$

This statement[8] defines an action with "id" as its id, and binds the action to the Java variable α. The f is a function defining the behavior that will be invoked when the action α is executed. This function is effectful and may change the agent state. The q, is a pure function, called the 'guard' of the action, specifying when the action is eligible for execution. Notice that the pair f, q can be seen as expressing a *reasoning rule* $q \to f$.

The guard q can be a predicate or a *query*. More precisely, let Σ be the type of the agent state and R the type of query results. We allow q to be a function of type $\Sigma{\to}R$. Whereas a predicate would inspect a state $s{:}\Sigma$ and simply return a true or a false, a query inspects s if it contains some object r satisfying a certain property. E.g. q might be checking if s contains a closed door. If such a door can be found, q returns it, else it returns **null**. This gives more information than just a simple true or false.

More precisely, the action α is executable on a state s if it is both control and guard-enabled on s. For now we can ignore control-enabledness. The action is *guard-enabled* on s when $q(s)$ returns some non-null r. The behavior function f has the type $\Sigma{\to}R{\to}\,V$

[8] Note that **action**, **do**$_2$, and **on**$_\_$ are not Java keywords. They are just methods. However, they also implement the Fluent Interface design pattern [19] commonly used in embedded Domain Specific Languages (DSLs) to 'trick' the syntax restriction of the host language to allow methods to be called in a sequence as if they form a sentence to improve the DSL's fluency.

```
var approachButton1 = action("approachButton1").                          1
  . do2((AgentState s) → (Entity button1) → {                             2
        var o = s.env.moveTowards(s.id, s.position, button1.position) ;   3
        s.markObservation(o) ;                                            4
        return s }                                                        5
    )                                                                     6
  . on_((AgentState s) → {                                                7
        var e = s.getEntity("button1") ;                                  8
        if (e==null) return null ;                                        9
        if (distance(s.position,e.position) < 0.01) return null ;         10
        return e ;                                                        11
    ) ;                                                                   12
```

Fig. 5. *An action that would move an agent closer to* button_1. *Notice that we again use* λ-*expressions (lines 3 and 7) to conveniently introduce functions without having to create a class.*

for some type V. When the action α is executed on s, it invokes $f(s)(r)$[9]. The result $v = f(s)(r)$, if it is not null, will then be checked if it achieves the agent's current goal.

For example, Figure 5 shows an action that can help agent Smith from Fig. 4. In the game Lab Recruits, to interact with a button a player character needs to stand close to the button. Although in Fig. 3 the character seems to stand close to button_1, it is not close enough. The tactic in Fig. 5, when invoked, will move the character closer to the button (but will not interact with it, yet). It may take several invocations to move the character close enough to the button. The action's guard specifies that the action is only enabled if button_1 exists (line 9) in the agent's belief. and furthermore its distance to the agent is $\geq$0.01 unit (line 10). The behavior part of the action, line 3, will then move the agent some small distance towards the button. Line 4 will incorporate the returned new observation (of the game state) into the agent's state.

Reasoning. Most of agent reasoning is carried out by actions' guards, since they are the ones that inspect the agent's state to decide which actions are executable. The reader may notice that the guard in the example in Fig. 5 is imperatively formulated, which is to be expected since aplib's host language, Java, is an imperative programming language. However, aplib also has a Prolog backend (using tuprolog [13]) to facilitate a declarative style of state query.

Figure 6 shows an example. To use Prolog-style queries, the agent's state needs to extend a class called StateWithProlog. It will then inherit an instance of a tuprolog engine to which we can add facts and inference rules, and then pose queries over these. Imagine a level in Lab Recruits where we have multiple doors and buttons. Some buttons may crank multiple doors when toggled. Suppose a test agent wants to get to a state where two doors, door_1 and door_2, are open. The example shows the definition of

[9] This scheme of using r essentially simulates *unification* a la pgrules in 2APL. Unification plays an important role in 2APL. The action in (4) corresponds to pgrule $q(r)? \mid f(r)$ The parameter s (the agent's state/belief) is kept implicit in pgrules. In 2APL this action is executed through Prolog, where q is a Prolog query and r is obtained through unification with the fact base representing the agent's state.

```
class AgentState extends StateWithProlog {                                      1
  ... addRules(                                                                 2
        clause(openDoors("B","D1","D2"))                                        3
        . IMPby(                                                                4
          or(and(close("D1"),connected("B","D1")),                              5
             and(open("D1")  ,unConnected("B","D1"),                            6
                  close("D2"),connected("B","D2"))))                            7
}                                                                               8
action("open_doors_1and2")                                                      9
  . do2((AgentState s) →                                                        10
        (Result r) → {                                                          11
          var b = s.getEntity(stringval(r.get("B")))  ;                         12
          if  (distance(s.position ,b.position) > 0.01)                         13
               o = s.env.moveTowards(s.id ,s.position ,b.position)  ;           14
          else o = s.env.interact(s.id ,b.id)  ;                                15
          s.markObservation(o)  ;                                               16
          return b })                                                           17
  . on_((AgentState s) → s.query(openDoors("B",door1,door2)))                   18
```

Fig. 6. *An example action whose guard, line 18, is formulated declaratively in the Prolog style.*

an action named “open_doors_1and2″ that will do this. Note that after opening one of these doors, the agent should be careful when trying to open the second. It needs to find a button that indeed opens the second door, but without closing the first one again. The reasoning needed to handle this is formulated as a Prolog rule called openDoors defined in lines 3-7. With the help of this rule, the guard for the action ”open_doors_1and2” can now be formulated as a Prolog query openDoors(B, door$_1$, door$_2$), which in aplib is expressed as in line 18. The predicate is true if door$_1$ is closed and B is a button connected to it (so, toggling the button would crank the door). Else, if door$_1$ is open, B should be connected to door$_2$, but not to door$_1$ (so, toggling it will not close door$_1$ again). So, assuming a solution exists, invoking the action above multiple times will first open door$_1$, unless it is already open, and then door$_2$. Notice that the guard is *declarative*, as it only characterizes the properties that a right button should have; it does not spell out how we should iterate over all the buttons in the agent’s belief to check it.

4 Structured Goals and Tactics

A goal can be very hard for an agent to achieve/solve directly. For example imagine a level in the game Lab Recruits, similar to Fig. 1, where we have to test some feature F located in some specific room. Let isInteracted$_F$ be the goal representing the agent is at F and manages to interact with it (and hence test it). To achieve this the agent will first need to reach the room where F is. To access this room a door D needs to be opened first. The door can be closed, in which case the agent first needs to find a specific button B that opens it. If the agent does not know all these steps, then directly solving isInteracted$_F$ will be very difficult.

We can help the agent by providing intermediate goals that it needs to solve first. We can formulate this as a 'goal structure' as the one below:

$$\begin{array}{l}\mathbf{SEQ}(\,\mathbf{FIRSTof}(\,\mathsf{isOpen}_\mathsf{D}, \mathbf{SEQ}(\mathsf{isActivated}_\mathsf{B}, \mathsf{isOpen}_\mathsf{D})),\\ \qquad \mathsf{isInteracted}_\mathsf{F})\end{array}$$

where $\mathsf{isOpen}_\mathsf{D}$ and $\mathsf{isActivated}_\mathsf{B}$ are intermediate goals. **SEQ** and **FIRSTof** are examples of so-called goal combinators explained below.

In aplib a composite goal is called a *goal structure*. It is a tree with goals as the leaves, and goal-combinators as nodes. The goals at the leaves are ordinary goals or test-goals, and hence they all have tactics associated to each. The combinators do not have their own tactics. Instead, they are used to provide a high level control on the order or importance of the underlying goals. Available combinators are as follows; let G and $G_1,...,G_n$ be goal structures:

- If g←--T∗ is a goal with the tactic T associated to it, $g.\mathsf{lift}()$ turns it to a goal structure consisting of the goal as its only leaf. T is implicitly attached to this leaf.
- $\mathbf{SEQ}(G_1,...,G_n)$ is a goal structure that is achieved by achieving all the subgoals $G_1,...,G_n$, and in that order. This is useful when G_n is hard to achieve; so $G_1,...,G_{n-1}$ act as helpful intermediate goals to guide the agent. Goal structures of this form also naturally express test scenarios as in (2).
- $H = \mathbf{FIRSTof}(G_1,...,G_n)$ is a goal structure where, given H to achieve, the agent will first try to achieve G_1. If this fails, it tries G_2, and so on until there is one goal G_i that is achieved. If none is achieved, H is considered as failed.
- $H = \mathbf{REPEAT}\ G$ is a goal structure where, given H to achieve, the agent will pursue G. If after sometime G fails, e.g. because it runs out of budget, it will be tried again. Fresh budget will be allocated for G, taken from what remains of the agent's total budget. This is iterated until G is achieved, or until H's budget runs out.

Dynamic Subgoals. Rather than providing a whole goal structure to an agent, sometimes it might be better to let the agent dynamically introduce or cancel subgoals. For example imagine an agent A which initially is given a goal structure $\Pi = \mathsf{SEQ}(\mathsf{isOpen}_\mathsf{D}, \mathsf{inRoom}_\mathsf{R})$. As the agent works on the first subgoal, $\mathsf{isOpen}_\mathsf{D}$ imagine that it discovers that the door D is closed, and hence the subgoal cannot be reached before another subgoal is solved (i.e. activate the button that opens the door).

Rather than pre-programming how to handle this in Π we can let the tactic of $\mathsf{isOpen}_\mathsf{D}$ to make this decision instead. Since a tactic has access to the agent's state, it can inspect this state. Based on what it discovers it may then decide to insert a new subgoal, let's call it $\mathsf{isActivated}_\mathsf{B}$, that will cause the agent to first find the button B and activate it in order to open D. The agent can do this by invoking $\mathsf{addBefore}(\mathsf{isActivated}_\mathsf{B})$, that will then change Π to:

$$\mathbf{SEQ}(\mathbf{REPEAT}(\mathbf{SEQ}(\mathsf{isActivated}_\mathsf{B}, \mathsf{isOpen}_\mathsf{D})), \mathsf{inRoom}_\mathsf{R})$$

The **REPEAT** construct will cause the agent to move back to $\mathsf{isActivated}_\mathsf{B}$ upon failing $\mathsf{isOpen}_\mathsf{D}$. The sequence $\mathbf{SEQ}(\mathsf{isActivated}_\mathsf{B}, \mathsf{isOpen}_\mathsf{D})$ will then be repeatedly attempted until it succeeds. The number of attempts can be controlled by assigning budget to the **REPEAT** construct (budgeting will be discussed below).

Budgeting. Since a goal structure can introduce multiple goals, they will be competing for the agent's attention. By default, aplib agents use the blind commitment policy [31] where an agent will commit to its current goal until it is achieved. However, it is possible to exert finer control on the agent's commitment through a simple but powerful budgeting mechanism.

When the agent was created, we can give it a starting computing budget β_0 (else it is assumed to be ∞). Let Π be the agent's root goal structure. For each sub-structure G in Π we can specify $G.\mathsf{bmax}$: the *maximum* budget G will get. Else, the agent conservatively assumes $G.\mathsf{bmax} = \infty$. By specifying bmax we control how much the agent should commit to a particular goal structure. This limit can be specified at the goal level (the leaves of Π), if the programmer wants to micro-manage the agent's commitment, or higher in the hierarchy of Π to strategically control it.

Once it runs, the agent will only work on a single goal at a time. The goal g it works on is called the *current goal*. Every ancestor of a current g is also current. For every goal structure G, let β_G denote the remaining budget for G. At the beginning, $\beta_\Pi = \beta_0$. When a goal or goal structure G in Π becomes current, budget is allocated to it as follows. When G becomes current, its parent either becomes current as well, or it is already current (e.g. the root Π is always current). Ancestors H that are already current keeps their β_H unchanged. Then, the budget for G is allocated by setting β_G to $\mathbf{min}(G.\mathsf{bmax}, \beta_{\mathsf{parent}(G)})$, after we recursively determine $\beta_{\mathsf{parent}(G)}$. This budgeting scheme is *safe*: the budget of a goal structure never exceeds that of its parent.

When working on a goal g, any work the agent does will consume some budget, say δ. This will be deducted from β_g and from the budget of its ancestors. If β_g becomes ≤ 0, the agent aborts g. It must then find another goal from Π.

4.1 Tactic

Rather than using a single action, Aplib provides a more powerful means to achieve a goal, namely tactic. A tactic is a hierarchical composition of actions. Methods used to compose them are also called *combinators*. Figure 7 shows an example of a tactic, composed with a combinator called FIRSTof. Structurally, a tactic is a tree with actions as leaves and tactic-combinators as nodes. The actions are the ones that do the actual work. Furthermore, recall that the actions also have their own guards, controlling their enabledness. The combinators are used to exert a higher level control over the actions, e.g. sequencing or choosing between them. This higher level control supersedes guard-level control[10]. The following tactic combinators[11] are provided; let $T_1, ..., T_n$ be tactics:

[10] While it is true that we can encode all control in action guards, this would not be an abstract way of programming tactical control and would ultimately result in error prone code.

[11] Earlier, in Section 1, we mentioned a relation with theorem provers. LCF-family theorem provers like HOL and Isabelle also have a concept of 'tactic', which basically is a function that constructs a proof of a given conjecture [40, 12, 22]. Since the solving proof is usually not known upfront, similar tactic combinators are used to control a search over the possible proof space. E.g. in HOL we have THEN, and ORLSE. These correspond to our SEQ and FIRSTof. HOL's REPEAT has no direct tactical counterpart in aplib, though aplib's deliberation cycles implicitly introduce a top-level repetition —this will be elaborated in Section 4.2.

```
var activateButton₁Tac = FIRSTof(
    action("activateButton1").do₂(..).on_(..).lift()
  . approachButton₁.lift()
  . action("explore").do₂(..).lift()
  )
```

Fig. 7. *The tactic for agent* Smith *in Fig. 4, composed from three other tactics. The first (its full code is not shown) is an action to activate* button_1 *if it is close enough to the agent. Otherwise, the action approachButton*$_1$ *(defined in Fig. 5) will move the agent towards the button, if it is visible to the agent (see the action's guard). Else,* FIRSTof *falls back to the last tactic that will explore the area around the agent to search the button. Note that without using a combinator like* FIRSTof *the control flow will have to be explicitly programmed into the actions' guards, resulting in a less abstract agent program, not to mention that the control flow would then be implicit, which makes the code harder to understand and more error prone.*

1. If α is an action, $T = \alpha.\textbf{lift}()$ is a tactic. Executing this tactic on an agent state s means executing α on s, which is only possible if α is enabled on s (if its guard results a non-null value when queried on s).
2. $T = \textbf{SEQ}(T_1,...,T_n)$ is a tactic. When invoked, T will execute the whole sequence T_1, ..., T_n.
3. $T = \textbf{ANYof}(T_1,...,T_n)$ is a tactic that randomly chooses one of enabled T_i's and executes it. A **SEQ** tactic is *enabled* if its first sub-tactic is enabled. For other combinators, it is enabled if one of its sub-tactic is enabled.
4. $T = \textbf{FIRSTof}(T_1,..,T_n)$ is a tactic. It is used to express priority over a set of tactics if more than one of them could be enabled. When invoked, T will invoke the first enabled T_i from the sequence $T_1,..,T_n$.

4.2 Aplib deliberation cycle

Consider a goal g↞T^*. When this goal becomes current, recall that the agent will then *repeatedly* execute T until g is achieved (or until its budget is exhausted). Aplib agents execute their tactics in cycles. In BDI agency these are called *deliberation cycles* [32, 10, 38]: in each cycle, an agent senses its environment, reasons which action to do, and then performs this action. To make itself responsive to changes in the environment, an agent only executes *one action per cycle*. So, if the environment's state changes at the next cycle, a different action can be chosen to respond to the change. However, if T contains a sub-tactic T' of the form $\textbf{SEQ}(T_1,..,T_n)$ things become more complicated. If T' is selected, the agent has to execute the whole sequence[12] which will take least n cycles, before it can repeat the whole T again. This makes the execution flow of a tactic non-trivial. We therefore have to deviate from the standard BDI deliberation [38].

Imagine an agent $A = (s, E, \Pi, \beta)$. At the start, A inspects its goal structure Π to determine which goal g↞T^* in Π it should pursue, and calculates how much of the budget β should be allocated for achieving g (β_g). A will then repeatedly apply T over multiple cycles until g is achieved, or β_g is exhausted. At every cycle, A does the following:

[12] Breaking off in the middle can be expressed using a combination of **FIRSTof** and **SEQ**.

1. *Sensing.* The agent asks the Environment to provide a fresh state information.
2. *Reasoning.* The agent determines which actions α in T are executable on the current state s. This is the case if α is guard-enabled on s and furthermore also control-enabled. The definition of latter is somewhat complicated. Let us explain it with an example instead. Suppose $T = \textbf{ANYof}(\alpha_0, \textbf{SEQ}(\alpha_1, \alpha_2), \alpha_3)$. The first time T is considered for execution, α_0, α_1 and α_3 becomes control-enabled, but not α_2. If α_0 turns out to be *not* guard-enabled, and α_1, α_3 are, only the latter two are executable. Suppose α_1 is chosen for execution. At the next cycle only α_2 is control-enabled. If it is also guard-enabled it can be executed, else it remains control-enabled for the next cycle. After α_2 is executed, the execution of the whole T is completed, and it can be repeated again.

   If no action is executable, the agent will sleep until the next cycle. Note that since the game under test runs autonomously, it may in the mean time move to a new state, and hence in the next cycle some actions may become enabled.
3. *Execution and resolution.* Let α be the selected action. It is then executed. If its result v is non-null, it is considered as a candidate solution to be checked against the current goal g. If v achieves g (so, g is solved), the agent inspects the remaining goals in Π to decide the next one to handle. The whole cycle is repeated, but with the new goal. If there is no goal left, then the agent is done. If g is *not* achieved, it is maintained and the whole cycle is repeated.

4.3 Test Robustness

Let us now explain more concretely why aplib test automation is more robust. Recall the tactic $\mathsf{activateButton_1Tac}$ (Fig. 7) to activate $\mathsf{button_1}$. Notice that it uses the tactic $\mathsf{approachButton_1}.\textbf{lift}()$ (defined in Fig. 5) to approach the button first in case the agent is not standing next to it. Notice that the location is not hard-wired in this tactic, but instead queried from the button itself. Let us also replace the call to moveTowards in line 3 in Fig. 5 with navigateTo. This will cause the agent to use aplib's 3D-space path finding to guide itself towards the given location. If the game designer now moves the button elsewhere, e.g. to swap its position with the far button in Fig. 3, the tactic will still work, as long as there is a path that reaches the button. The tactic $\mathsf{approachButton_1}$ requires however that the button is already in the agent's belief, which would not be the case if the developer moves it to a new position that is initially not visible to the agent. Fortunately the enclosing tactic $\mathsf{activateButton_1Tac}$ can deal with that, by falling back to the 'explore' tactic to search the button first.

If the level contains some random fire hazard, we can replace $\mathsf{approachButton_1}$ in $\mathsf{activateButton_1Tac}$ with a more adaptive variant e.g.:

$$\textbf{FIRSTof}(\mathsf{avoidHazardTac},\ \mathsf{approachButton_1}.\textbf{lift}())$$

If the agent now detects fire when it on its way to $\mathsf{button_1}$, it will first try to evade the fire before resuming its navigation to $\mathsf{button_1}$. Importantly, since the tactic executability is re-checked at every deliberation cycle, the agent will be able to timely invoke the above re-planning.

5 Proof of Concept

Lab Recruits	
C# scripts	64 files, 3524 sloc
animation control	12 files, 2763 lines
Implementation of Environment	
game-side	393 sloc C#
Java-side	1056 sloc Java
Support	
Domain specific tactics	505 sloc Java
General support (world representation, pathfinding)	1250 sloc Java
Utilities	240 sloc Java
Tests with aplib	
(game logic) button & door	74 sloc Java (28 sloc actual test-code)
(level test) state transitions (3)	117 sloc Java (61 sloc actual test-code)
(level test) simple reachability	69 sloc Java (19 sloc actual test-code)
(level test) complex reachability	98 sloc Java (46 sloc actual test-code)

Fig. 8. Some statistics of the experiment with the Lab Recruits game.

We conducted a pilot on the previously mentioned Lab Recruits game[13], as a proof of concept, and to get a preliminary idea on the effort to integrate aplib into the development cycle and to write tests. Lab Recruits is developed by a group of students using an established game development framework called Unity 3D. It consists of about 3500 lines of C# scripts. In Unity, not all dynamics are programmed in such scripts. E.g. animation is designed with a separate tool, from which meta files ($\approx$ 2700 lines) are generated and compiled to behavior.

To extend their entertainment, most games are replayable on different instances of the playing world, so-called *levels*. Levels have unique layout, monsters and items drop, etc. The logic (game rules) is however the same over all levels. Levels are often meticulously hand crafted (it is an art that computers have not mastered yet), hence requiring significant human effort. A level in Lab Recruits represents a laboratory building, consisting of rooms, in one or multiple floors, populated by in-game objects, such as tables, and chairs. Some of them are interactable, such as buttons. Some of them represent hazard, such as, fire. In addition to testing the correctness of the general game logic, note that every newly crafted level also requires testing, e.g. to make sure that in-game entities which are necessary for completing the level are indeed reachable by the player.

Integration Effort As remarked in Section 3.1 to use aplib the developers need to first provide an implementation of the interface Environment for their game. For Lab Recruits this amount to about 1400 lines of Java and C# —see Fig. 8. While this gives test agents basic control over the game, an important lesson we learned is that this is not enough. More abstract ways to control and navigate through the game are necessary. These are provided as a library of tactics ($\approx$ 500 lines) and support classes e.g. to do path-planning on a 3D surface ($\approx$ 1200 lines). Such tactics are quite game-specific, but much of the path-planning functionality is generic and will in the future be migrated to aplib's standard library.

[13] https://github.com/iv4xr-project/labrecruits

While the amount of integration code is relatively substantial compared to the size of Lab Recruits itself, it does not mean that if we extend Lab Recruits with new game objects and new logic the integration code will grow as much. Moreover, the same integration can be used to test as many new levels as we have, no matter how large or complex they are.

Testing with aplib We used aplib agents to test Lab Recruits' general logic and a number of sample levels —an overview is given in Fig. 8. To test the general logic it is sufficient to make a minimalistic level exposing the aspects of the logic that we want to test. E.g. a button in Lab Recruits should open/close doors bound to it (and only those doors). This proves that if they are bound correctly, they will also interact correctly. This can be tested with a mini level with one button and several doors. The corresponding testing task takes 74 lines of code, though only a third of them describes the task itself.

A typical testing tasks when testing a level is to verify that every entity (or at least, the key entities) has the right behavior, e.g. that a button would open the right door (in other words, whether the level binds the correct doors to the button). In our experiment, testing three such buttons takes about 120 lines of code, but only about half of them actually describe the task.

Another typical testing task is to check if key entities in a level are actually reachable. In the simple case, an entity is reachable through an unobstructed path in the level. However, note that the entity might *not* be visible from the agent's initial position. So, solving such a task also involves searching the level. On the other hand, this contributes to the robustness of the test: if the developers change the level's layout or move the entity elsewhere, the test code will not break as long as the entity remains reachable. In our example, such a test takes about 70 lines. The code is reusable, irrespective the size and complexity of the level, as long as the target entity is reachable in the above sense.

In a more complex situation, reaching an entity requires opening a series of doors that block the path to it. To verify the entity's reachability, we simply translate the needed sequence of essential buttons (that should be toggled to open the guarding doors) into subgoals. For a setup that involves three buttons and three doors it takes about 100 lines of test code; only about half of them actually describe the task. The approach can be smarter (e.g. if we can eliminate the need to add subgoals), but this is not the goal of the current pilot, and left as future work.

6 Related Work

Software agents have been employed in various domains, e.g. computer games, health care, and control systems [26, 30, 27]. With aplib we have another usecase, namely automated testing. Using agents for software testing has actually been attempted before [36, 33, 5, 35]. However, these works use agents to test services or web applications, which are software types that can already be handled by non-agent techniques such as model based [43] or search based [20, 1] testing, whereas we argued that high interactivity of computer games poses a different level of challenge for automated testing.

To program agents, without having to do everything from scratch, we can either use an agent 'framework', which essentially provides a library, or we use a dedicated agent programming language. Examples of agent frameworks are JADE [6] and aplib for Java, HLogo [7] for Haskell, and PROFETA [18] for Python. Examples of dedicated agent languages are JASON [8], 2APL [9], GOAL [24], JADEL [25], and SARL [39]. HLogo is an agent framework that is specialized for developing an agent-based simulation. On the other hand, JADE and aplib are generic agent frameworks that can be connected to any environment. Aplib is light weight compared to JADE. E.g. the latter supports distributed agents and FIPA compliance which aplib does not have. JADE does not natively offers BDI agency, though BDI agency, e.g. as offered by JADEL, can be implemented on top of JADE. In contrast, aplib and PROFETA are natively BDI agent frameworks.

Among the dedicated agent programming languages, some are dedicated for programming BDI agents. The good thing is that they offer Prolog-style declarative programming. On the down side e.g. available data types are restricted (e.g. no support for collection and polymorphism), which is a serious hinderance if we are to use them for large projects. One with a very rich set of language features (collection, polymorphism, OO, lambda expression) is SARL, though it is non-BDI. PROFETA and aplib are somewhere in between. Both are BDI DSLs, but they are embedded DSLs rather than a native language as SARL. To improve its fluency as a DSL, aplib makes heavy use of design patterns such as Fluent Interface [19] and Strategy Pattern [21]. PROFETA and aplib's host languages are full of features (Python and Java, respectively), that would give the strength of SARL that agent languages like JASON and GOAL cannot offer.

Aplib's distinguishing feature compared to other implementations of BDI agency (e.g. JACK, JASON, 2APL, GOAL, JADEL, PROFETA) is its tactical programming of plans (through tactics) and goals (through goal structures). An agent is essentially set of actions. The BDI architecture does not traditionally impose a rigid control structure on these actions, hence allowing agents to react adaptively to changing environment. However, there are also goals that require certain actions to be carried out in a certain order over multiple deliberation cycles. Or, when given a hard goal to achieve, the agent might need to try different strategies, each would need to be given enough commitment by the agent, and conversely it should be possible to abort it so that another strategy can be tried. All these imply that tactics and strategies require some form of control structures, although not as rigid as in e.g. procedures. All the aforementioned BDI implementations do not provide control structures beyond intra-action control structures. This shortcoming was already observed by [17], stating domains like autonomous vehicles need agents with tactical ability. They went even further, stating that Agent Oriented Software Engineering (AOSE) methodologies in general do not provide a sufficiently rich representation of goal control structures. While inter-actions and inter-goals control structures can be encoded through pushing and popping of beliefs or goals into the agent's state, such an approach would clutter the programs and error prone. An existing solution for tactical programming for agents is to use the Tactics Development extension [17] of the Prometheus agent development methodology [34]. This extension allows tactics to be graphically modelled, and template implementations in JACK can be generated from the models. In contrast, Aplib provides the features directly at the programming level. It provides the additional control structures suitable for tactical programming over the usual rule-based style programming of BDI agents. When programming test agents, having an option to exert control helps the tester to narrow the agents' search space which may benefit their performance, which is important when we start to accumulate a large number of tests.

Let us also mention the agent language IndiGolog [11] from the Golog-family [28]. The original Golog [28] allows a model of an environment to be expressed in a mix of imperative statements and 'situation calculus' axioms (comparable to Hoare triples). A Golog agent solves goals *off-line*, using the model. The obtained plan (sequence of actions) are then executed on the environment. Such an approach is less suitable for testing a game due to the latter's non-determinism. In contrast, IndiGolog offers a mix of reactive programming and model-based off-line planning. If a test-goal can be broken into subgoals where some are robust against the game's non-determinism, off-line planning can be employed to handle the latter. Although testing is not a main use-case of IndiGolog nor Golog, their idea actually resembles a well known testing approach called Model Based Testing (MBT) [15] where Labelled Transition Systems (LTS) or Extended Finite State Machines (EFSMs) are often used as models. In MBT, a model also defines correctness (e.g. when the model specifies b to happen after a, the implementation is expected to behave in the same way), in addition to providing guidance on how to reach a given goal state as in Golog. Aplib currently has no MBT capability; this is future work. Extending aplib with MBT would benefit from aplib's tactical layer, which as pointed out in Section 4.3 improves agents' robustness against non-determinism, which in terms of MBT would allow more goals to be solved off line. Since requiring developers to provide detailed models is unlikely to scale up, future research should be focused on model learning [43], e.g. to learn the parts of the model that only serve to provide goal solving guidance, so that developers only need to focus on the parts that capture the game's correctness.

7 Conclusion & Future Work

We have presented aplib, a BDI agent programming framework featuring multi agency and novel tactical programming and strategic goal-level programming. We choose to offer aplib as a Domain Specific Language (DSL) embedded in Java, hence making the framework very expressive. Despite the decreased fluency, we believe this embedded DSL approach to be better suited for large scale programming of agents, while avoiding the high expense and long term risk of maintaining a dedicated agent programming language.

With the above features aplib would be a good choice to be used as a framework to program test agents for testing highly interactive software such as computer games. Our experience so far with the Lab Recruits case study (Fig.1) shows that even a simple test agent that can navigate within a closed terrain already introduces automation that is previously not possible. Larger and more thorough case studies are still future work. We would also like to explore the use of emotion modelling framework such as FAtiMA [14] alongside aplib agents to allow us to test user experience (e.g. whether the game becomes too boring too quickly), which is an aspect of a great concern in the game industry.

While in many cases relying on reasoning-based intelligence is enough, there are also cases where this is not. Recently we have seen rapid advances in learning-based

AI. As future work we seek to extend aplib to let programmers hook learning algorithms to their agents to teach the agents to make the right choices, at least in some situation, as an alternative when rule-based reasoning becomes too complicated (e.g. when it involves recognizing visual or audio patterns).

References

1. Alshahwan, N., Harman, M.: Automated web application testing using search based software engineering. In: 26th Int. Conference on Automated Software Engineering. IEEE (2011)
2. Alégroth, E., Feldt, R., Kolström, P.: Maintenance of automated test suites in industry: An empirical study on visual gui testing. Information and Software Technology **73**, 66–80 (2016)
3. Ammann, P., Offutt, J.: Introduction to software testing. Cambridge University Press (2016)
4. Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., Mcminn, P., Bertolino, A., et al.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software **86**(8) (2013)
5. Bai, X., Chen, B., Ma, B., Gong, Y.: Design of intelligent agents for collaborative testing of service-based systems. In: 6th Int. Workshop on Automation of Software Test. ACM (2011)
6. Bellifemine, F., Poggi, A., Rimassa, G.: JADE–a FIPA-compliant agent framework. In: Proc. PAAM (1999)
7. Bezirgiannis, N., Prasetya, I., Sakellariou, I.: Hlogo: A parallel Haskell variant of NetLogo. In: 6th Int. Conf. on Simulation and Modeling Methodologies, Tech. and Applications (SIMULTECH). IEEE (2016)
8. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason, vol. 8. John Wiley & Sons (2007)
9. Dastani, M.: 2APL: a practical agent programming language. Autonomous agents and multi-agent systems **16**(3) (2008)
10. Dastani, M., Testerink, B.: Design patterns for multi-agent programming. Int. Journal Agent-Oriented Software Engineering **5**(2/3) (2016)
11. De Giacomo, G., Lespérance, Y., Levesque, H.J., Sardina, S.: IndiGolog: A high-level programming language for embedded reasoning agents. In: Multi-Agent Programming, pp. 31–72. Springer (2009)
12. Delahaye, D.: A tactic language for the system coq. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. Springer (2000)
13. Denti, E., Omicini, A., Calegari, R.: tuProlog: Making Prolog ubiquitous. ALP Newsletter (Oct 2013)
14. Dias, J., Mascarenhas, S., Paiva, A.: Fatima modular: Towards an agent architecture with a generic appraisal framework. In: Emotion modeling. Springer (2014)
15. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies (2007)
16. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Science of computer programming **69**(1-3), 35–45 (2007)
17. Evertsz, R., Thangarajah, J., Yadav, N., Ly, T.: A framework for modelling tactical decision-making in autonomous systems. Journal of Systems and Software **110** (2015)
18. Fichera, L., Messina, F., Pappalardo, G., Santoro, C.: A Python framework for programming autonomous robots using a declarative approach. Sc. of Computer Programming **139** (2017)
19. Fowler, M., Evans, E.: Fluent interface. martinfowler. com (2005), `https://martinfowler.com/bliki/FluentInterface.html`

20. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering. ACM (2011)
21. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley (1994)
22. Gordon, M.J., Melham, T.F.: Introduction to HOL A theorem proving environment for higher order logic. Cambridge Univ. Press (1993)
23. Herzig, A., Lorini, E., Perrussel, L., Xiao, Z.: BDI logics for BDI architectures: old problems, new perspectives. KI-Künstliche Intelligenz **31**(1) (2017)
24. Hindriks, K.V.: Programming Cognitive Agents in GOAL (2018), `https://goalapl.atlassian.net/wiki/spaces/GOAL/overview`
25. Iotti, E.: An agent-oriented programming language for JADE multi-agent systems. Ph.D. thesis, Università di Parma. Dipartimento di Ingegneria e Architettura (2018)
26. Jennings, N., Jennings, N.R., Wooldridge, M.J.: Agent technology: foundations, applications, and markets. Springer Science & Business Media (1998)
27. Leitão, P.: Agent-based distributed manufacturing control: A state-of-the-art survey. Engineering Applications of Artificial Intelligence **22**(7) (2009)
28. Levesque, H.J., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.B.: GOLOG: A logic programming language for dynamic domains. The J. of Logic Programming **31**(1-3), 59–83 (1997)
29. McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability **14**(2), 105–156 (2004)
30. Merabet, G.H., Essaaidi, M., Talei, H., Abid, M.R., Khalil, N., Madkour, M., Benhaddou, D.: Applications of multi-agent systems in smart grids: A survey. In: Int. conf.on multimedia computing and systems (ICMCS). pp. 1088–1094. IEEE (2014)
31. Meyer, J.J., Broersen, J., Herzig, A.: Handbook of Logics for Knowledge and Belief, chap. BDI Logics, pp. 453–498. College Publications (2015)
32. Meyer, J.J.C.: Agent technology. In: Wah, B.W. (ed.) Encyclopedia of Computer Science and Engineering. John Wiley & Sons (2008)
33. Miao, H., Chen, S., Qian, Z.: A formal open framework based on agent for testing web applications. In: Int. Conf. on Computational Intelligence and Security (CIS). IEEE (2007)
34. Padgham, L., Winikoff, M.: Prometheus: A practical agent-oriented methodology. In: Agent-oriented methodologies. IGI Global (2005)
35. Paydar, S., Kahani, M.: An agent-based framework for automated testing of web-based systems. Journal of Software Engineering and Applications **4**(02) (2011)
36. Qi, Y., Kung, D., Wong, E.: An agent-based testing approach for web applications. In: 29th Int. Computer Software and Applications Conference (COMPSAC). vol. 2. IEEE (2005)
37. Rafael, H., Mehdi, D., Jürgen, D., Amal, E.: Multi-Agent Programming-Languages, Platforms and Applications. Springerg (2005)
38. Rao, A.S., Georgeff, M.P.: An abstract architecture for rational agents. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning (1992)
39. Rodriguez, S., Gaud, N., Galland, S.: SARL: a general-purpose agent-oriented prog. language. In: Int. Conf. on Intelligent Agent Technology. IEEE (2014)
40. Schmidt, D.A.: A programming notation for tactical reasoning. In: International Conference on Automated Deduction. pp. 445–459. Springer (1984)
41. Seghrouchni, A.E.F., Dix, J., Dastani, M., Bordini, R.H.: Multi-Agent Programming: Languages, Tools and Applications. Springer (2009)
42. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Int. Conference on Computer Aided Verification. Springer (2006)
43. Vos, T., Tonella, P., Prasetya, I.S.W.B., Kruse, P.M., Bagnato, A., Harman, M., Shehory, O.: FITTEST: A new continuous and automated testing process for future internet applications. In: CSMR-WCRE. IEEE (2014)