Aplib: An Agent Programming Library for Testing Games

I. S. W. B. Prasetya
Utrecht University
s.w.b.prasetya@uu.nl

Mehdi Dastani
Utrecht University
m.m.dastani@uu.nl

ABSTRACT
Testing modern computer games is notoriously hard. Highly dynamic behavior, inherent non-determinism, and fine grained interactivity blow up their state space; too large for traditional automated testing techniques. An agent-based testing approach offers an alternative as agents’ goal driven planning, adaptivity, and reasoning ability can provide an extra edge. This paper provides a summary of aplib, a Java library for programming intelligent test agents, featuring tactical programming as an abstract way to exert control on agents’ underlying reasoning based behavior. Aplib is implemented in such a way to provide the fluency of a Domain Specific Language (DSL) while still staying in Java, and hence aplib programmers will keep 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

ACM Reference Format:

1 INTRODUCTION
Computer games have become increasingly more interactive and complex. Modern games improve realism by allowing users to have fine grained control/interactions. They also employ a large number of interacting in-game entities, whose behavior is often highly unpredictable. These make the games hard to test. The induced interaction space is vast, which is hard to deal with the existing automated testing techniques such as search based [8, 11], random [4, 13], model based [1, 18, 19], or symbolic [17].

Contribution. We need new integrated techniques that support intelligent navigation through a large interaction space and towards specific to-be tested game states. Such techniques should allow reasoning about the game (current and goal) states, actions, and plans in order to decide and develop strategies towards goal states. Such integrated reasoning techniques should also capture the strategic and tactical nature of game playing. Integrated reasoning techniques have been extensively studied in the field of multi-agent system engineering in order to design and build software agents that autonomously decide how to interact with their environment to achieve their objectives. This paper applies this approach to game testing and presents aplib\(^1\), a Java library for programming intelligent agents suitable for automating testing tasks.

Notable features of aplib are: multi-agent, goal-driven agents as in the BDI model [9], integration with Java testing frameworks JUnit, and a novel layer of tactical programming that provides an abstract way to exert imperative control on the underlying reasoning based agents behavior. Such a programming approach is useful for implementing testing scenarios that often require subtasks to be carried out in a certain order, while maintaining the adaptivity of the underlying behavior. As opposed to dedicated agent programming languages [2, 5, 6, 10, 14, 16, 20] aplib offers a Domain Specific Language (DSL) embedded in Java. Aplib programmers will thus program in Java, but they will get a set of APIs that give the fluent appearance of a DSL. A dedicated programming language for writing tests should be rich enough and have enough tool and community support as otherwise most companies will consider it as unacceptable risk. On the other hand, when using an embedded DSL the programmers have direct access to all the benefit of its host language (in this case Java) such as its expressiveness, static typing, rich libraries, and wealth of development tools.

2 APLIB TEST AGENTS
Given a game, one or more aplib test agents can be deployed to test it. To control the game, the agents interact with it through a proxy: an abstract interface called Environment. This indirection is essential to make aplib independent from the technology used by the game. It does mean that each game will have to implement its own instance of Environment. The effort should be nominal and moreover it is a one off investment. Typically the agents will control in-game player-characters. The Environment should provide at least a method for an agent to send a command to the character it controls and another to obtain information on what this character currently observes. Aplib makes no assumption on whether the provided observation is structural or visual (image), though in our current use cases we only work with structural information.

An agent can be given a set of goals. It runs in sense-reason-act cycles a la [7, 12, 15], until it has no goal left to achieve, or it runs out of computing budget. At each cycle, it obtains an observation on the game state, reasons about it to decide which action to perform to bring it closer to its current goal, and to perform the chosen action. The game itself runs autonomously. It may have in-game entities that independently influence the game state during a cycle.

An agent \( A \) can be given a testing task, which is a form of a goal, formulated in aplib/Java as:

\[
\text{testgoal}(\text{goalname}, A) \cdot \text{toSolve}(\phi) \cdot \text{invariant}(\psi)
\]

\(^{1}\)https://ivex-project.github.io/aplib/

where $\phi$ is a goal for $A$, i.e., a predicate describing a family of game states whose correctness need to be verified, and $\psi$ is a predicate expressing the correctness property to hold on those states. To verify this, the agent will have to play the game to get it to at least one state satisfying $\phi$ before it can check if $\psi$ holds. The hard part in automating such a task is not in checking $\psi$, but rather in ‘reaching’ $\phi$, i.e., finding a right plan to move the game to the $\phi$ state.

To reach the goal, the agent has actions, e.g. to move the agent’s in-game character some small distance in a certain direction, or to make the character interacts with another entity. At each cycle one action is selected for execution. Actions are guarded, i.e., move a La Action Systems [3]. The syntax of an action is defined as:

$$\text{action(name) . do(f) . on(q)}$$

where $f$ describes what the action does and $q$ is its guard, expressed in plain Java or as a Prolog-style query. The action is only enabled for execution if $q$ evaluates to true on the agent’s current state (which includes its view on the current game state). Importantly, the guards express the reasoning of when to do what action. If more than one actions are enabled, one is chosen randomly.

To make test runs performant, we will often need to superimpose some form of strategic control over the bare reasoning provided by actions’ guards, e.g. which actions should be given precedence for execution. To do this, aplib allows actions to be composed into tactics: $\alpha . \text{lift}()$ lifts an action $\alpha$ to become a tactic; $\text{SEQ}(T_1, \ldots, T_n)$ is a tactic that sequentially executes its sub-tactics; $\text{FIRSTTof}(T_1, \ldots, T_n)$ executes the first enabled/executable sub-tactic; $\text{ANY}(T_1, \ldots, T_n)$ randomly chooses one of its enabled sub-tactic. When using a tactic $T$ to achieve a goal, at each cycle the agent will look for an action in $T$ which is both control and guard enabled. Due to the SEQ, it may take multiple cycles to complete $T$. If by that time the goal is not achieved yet, the next cycle will repeat $T$. This goes on until the goal is achieved.

When the testing task $h = \text{testgoal()} . \text{toSolve}(\phi) . \text{invariant}(\psi)$ is non-trivial, the agent may need some help to automate it. Instead of simply giving $h$ to the agent, we give it a so-called goal structure of the form $\text{SEQ}(g_1, \ldots, g_k, h)$ where $g_i$’s are intermediate sub-goals provided as help. The agent should first pursue them in the given order. After solving $g_k$, solving the final $h$ should be easier for the agent. Other constructors to construct goal structures are $\text{FIRSTTof}(G_1, \ldots, G_n)$ and $\text{REPEAT}(G)$. Additionally, it is possible for the agent to dynamically insert a goal to, or remove one from, its goal structure. More details can be found in aplib’s documentation.

### Example

Figure 1 shows a 3D game called Lab Recruits\(^2\). Players can explore the inside of a lab e.g. to find a certain designated room. The lab consists of rooms, and possibly multiple floors. It is populated with various in-game entities, some can be interacted to, some may be moving, and some may represent hazard such as fire. Imagine that the game designer has designed a ‘level’ (in this case that would be a ‘lab’). A typical testing task would be to verify that critical rooms in the level are reachable from the player’s assumed starting position. In this game, access to a room can be guarded by doors, which initially can be closed. A correct in-game button should be activated to open them. Solving this can be complicated if the level is large, with buttons that can be hidden behind closed doors\(^3\). As our example here, let us however consider a simple instance of this problem.

Consider a level where the player starts in the room $R$ with a door $D_1$ and several buttons. We want to verify that button $B_1$ would open $D_1$. This is how to formulate this testing task for a test agent $A$:

\[
\begin{align*}
\text{var } h &= \text{testgoal(“check door”, } A) \\
&. \text{toSolve(st} \rightarrow \text{isActive(st.get(“B_1”)))} \\
&. \text{invariant(st} \rightarrow \text{isOpen(st.get(“D_1”)))}
\end{align*}
\]

The notation $st \rightarrow e$ is Java’s lambda-expression; above it is used to define state predicates, where the bound variable $st$ represents the agent’s state, initial position. To operate it, the agent needs to stand next to it, it will need to have at least an action to approach it:

\[
\begin{align*}
\text{var } &\text{approachButton}_1 = \text{action(“approach } B_1”)} \\
&. \text{do(st} \rightarrow \text{navigateTo(st.get(“B_1”).position) ; return st}) \\
&. \text{on(st} \rightarrow \text{distance(st.position, st.get(“B_1”).position) } \geq 0.1)
\end{align*}
\]

The method navigateTo will drive the agent to $B_1$’s position. Internally, it uses a 3D path planning algorithm. It may take multiple cycles to actually reach $B_1$. To add adaptiveness, e.g. to avoid hazard that dynamically occurs underway, we can instead use a tactic e.g.:

\[
\text{FIRSTTof( avoidHazardTactic, approachButton}_1, \text{lift}() )
\]

To automate the testing task $h$ above, simply approaching $B_1$ is enough. The agent needs to interact with it, then navigate (close enough) to the door $D_1$ to get its recent state, and then we can verify $h$’s invariant. This can be programmed using sub-goals:

\[
\text{SEQ}(G_1, G_2, G_3, h, \text{lift}() )
\]

where the goal $G_1$ would bring the agent to stand next to the button $B_1$, $G_2$ would make $B_1$’s state turned to activated, and $G_3$ brings the agent to a position where the door $D_1$ can be observed. Then finally the agent will check $h$.

### 3 CONCLUSION

We have briefly summarized the features of the tactical Agent Programming Library aplib, suitable for programming automated testing on computer games. More details can be found in aplib’s github site, along with the Lab Recruits demo.

**Acknowledgement.** This work is funded by EU ICT-2018-3 H2020 Programme, grant nr. 856716.

---

\(^2\)https://github.com/iv4xr-project/labrecruits

\(^3\)E.g. the game Dungeons & Dragons Online, https://www.ddo.com, has a number of such notorious levels.
REFERENCES


