DSL Design for Reinforcement Learning Agents

Christopher Simpkins
College of Computing
Georgia Institute of Technology
Atlanta, GA, U.S.A.
chris.simpkins@gatech.edu

Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, GA, U.S.A.
spencer@cc.gatech.edu

Charles Isbell, Jr.
College of Computing
Georgia Institute of Technology
Atlanta, GA, U.S.A.
isbell@cc.gatech.edu

Abstract

Writing software that employs artificial intelligence (AI) is complex because the algorithms that must be implemented in general purpose programming languages are complex. One solution to this problem is to embed AI algorithms in domain specific languages (DSLs). DSLs are the "ultimate abstraction" for creating programs for a particular domain [1], but the question of how or even why to do this is not easily answered. We have developed a language with integrated reinforcement learning designed for writing intelligent agents. AFABL (A Friendly Adaptive Behavior Language), is implemented as an internal DSL shallowly embedded in the Scala programming language [3]. We discuss the development of AFABL, the basic elements of AFABL with an example, the way AFABL captures domain knowledge, the benefits of integrating reinforcement learning into a programming language and report the results of a programmer study which confirms and quantifies the usefulness of integrating reinforcement learning into a programming language.

CCS Concepts • **Computing methodologies → Intelligent agents**; *Q-learning*; • **Software and its engineering → Domain specific languages**;

ACM Reference Format:
Christopher Simpkins, Spencer Rugaber, and Charles Isbell, Jr.. 2017. DSL Design for Reinforcement Learning Agents. In *Proceedings of Workshop on Domain-Specific Language Design and Implementation at SPLASH, Vancouver, Canada, October, 2017 (DSLDI-2017),* 3 pages.
https://doi.org/

1 Languages for Intelligent Agents

An agent is an autonomous entity that senses its environment and takes actions that change the environment's state. In its simplest form an agent is a finite state machine. An intelligent agent pursues goals – the function that maps states to actions is created by the agent based on its goals. Writing intelligent agents is complex because the algorithms for creating those behavioral functions are complex, and writing intelligent agents that adapt to either partially specified tasks or environments with changing dynamics is even harder.

An early DSL for writing intelligent agents, ABL (A Behavior Language) [2], allows programmers to express an agent's "physical" and mental behaviors that the language's internal planning algorithms select in pursuit of goals. ABL was used to create ground-breaking interactive games and dramas. However, writing ABL programs is cumbersome because, among other things, programmers must specify preconditions for selecting actions and duplicate action specifications for each goal that may use them.

To improve ABL we set out to add adaptivity by integrating reinforcement learning into ABL [5]. Adaptivity would relieve programmers from specifying preconditions for behaviors, duplicating actions in the specification of different behaviors, and writing low-level action selection logic. ABL is an external DSL with a JavaCC-based parser and code generator emitting JVM bytecode. Our initial plan was to modify the ABL compiler, but upon reflection of the effort involved and the core questions we wanted to answer during the early stages of our research into language-integrated reinforcement learning we decided to write our DSL, which we then called AFABL, as an internal DSL embedded in the Scala programming language. Doing so allowed us to focus on issues of capturing domain knowledge through state and reward authoring, and study the usefulness of integrating reinforcement learning on a small scale to justify putting effort into expanding the language. This approach succeeded.

2 The AFABL DSL for Intelligent Agents

To ground the discussion in a concrete example, we write an agent for a toy problem in which the agent must simultaneously pursue two goals, described in Figure 1. Figure 2 shows AFABL code for a bunny agent. This code would typically fit in a single editor window and represents a tremendous amount of functionality. This agent pursues two goals simultaneously and prioritizes them based on the relative locations of the bunny, the food, and the wolf.

Three components – world, state abstraction and module reward – define a module specific learning problem on a subset of the world in which the agent may act. The world in which an agent acts is represented as a set of states, here the locations of bunny, food, and wolf. A state abstraction is the subset of the world state relevant to a particular behavior module. Reward is familiar to most people – a positive or negative signal indicating the goodness or badness of a particular state. Internally, AFABL uses these components to instantiate a Sarsa reinforcement learning algorithm [4], but the programmer need not be aware of any details of

DSLDI-2017, October, 2017, Vancouver, Canada
2017. ACM ISBN …$15.00
https://doi.org/

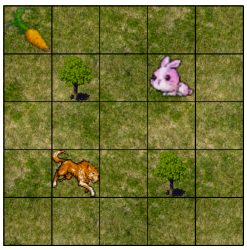

Figure 1. The bunny must pursue two goals simultaneously: find food and avoid the wolf. The bunny may move north, south, east, or west. When it finds food it consumes the food and new food appears elsewhere in the grid world, when it meets the wolf it is eaten and "dies." Sometimes the wolf is near the food, putting the bunny's goals in conflict.

```
case class FindFoodState(bunny: Location, food: Location)
val findFood = AfablModule(
  world = new BunnyWorld,
  stateAbstraction = (worldState: BunnyState) => {
    FindFoodState(worldState.bunny, worldState.food)
  },
  moduleReward = (moduleState: FindFoodState) => {
    if (moduleState.bunny == moduleState.food) 1.0 else -0.1
  }
)
val bunny = AfablAgent(
  world = new BunnyWorld,
  modules = Seq(findFood, avoidWolf), // AvoidWolf similar to FindFood
  agentLevelReward = (state: BunnyState) => {
    if (state.bunny == state.wolf) 0.0
    else if (state.bunny == state.food) 1.0
    else 0.5
  }
)
```

Figure 2. A bunny agent in the AFABL DSL.

reinforcement learning *algorithms*. The AFABL programmer need only be familiar with the reinforcement learning *problem* – world states, agent actions, and rewards. An AFABL agent is composed of independent behavior modules and an arbitrator that uses an agent level reward function to learn when it should listen to each module. AFABL allows programmers to express these components concisely, with very little cognitive distance between the concepts that make up the agent and the code that represents them.

One can reason that the benefit of using AFABL over a general purpose programming language like Scala is that a programmer using a general purpose programming language must encode the action selection logic manually, resulting in more code and more cognitive burden. If, for example, we incrementally adapt our AFABL bunny agent to worlds in which the bunny must not only find food and avoid a wolf, but also find a mate, account for spoiling food, and picky mates that will not accept the bunny unless it has recently eaten, all we need to do is add a module for finding a mate. The AFABL agent will adapt to the additional factors – spoiling food, picky mates – using reinforcement learning. In contrast, agent programs written in general purpose programming languages must add considerable code to deal with the changing world dynamics. The complexity of AFABL programs grows in a sub-linear, often logarithmic fashion when task environment changes can be handled automatically by AFABL's reinforcement learning algorithms.

3 Quantitative Benefits of AFABL

We conducted a study in which 16 programmers completed two programming tasks using Scala and AFABL. In Task 1 programmers wrote a bunny agent for a bunny-food-wolf world. In Task 2 programmers wrote a bunny agent for a world that is identical to the world in Task 1 except that the bunny must also find mates. As in Task 1, the bunny's percepts are complete state descriptions: the locations of the bunny, the wolf and the mate. We analyzed the submissions of study participants to compare Scala agents to AFABL agents in terms of code size, time spent writing Scala versus AFABL agents, the McCabe cyclomatic complexity of Scala versus AFABL agent code, and the performance of the agents on the assigned tasks. Our results showed statistically significant improvements in code complexity and agent performance for AFABL agents over Scala agents.

4 Conclusion

Shallowly embedding AFABL in Scala allowed us to implement the language relatively easily so that we could first answer the question of whether it would be worthwhile to develop AFABL into a more complete language with additional features and a more robust implementation (perhaps a deeply embedded or external DSL). Our results suggest that it is worthwhile to integrate reinforcement learning into a programming language. Refinement of AFABL's syntax (reward authoring is not always necessary), additional features, and the question of the particular language implementation strategy (internal versus external DSL, deep versus shallow embedding) remain.

References

[1] Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *ACM COMPUTING SURVEYS* 28 (1996).

[2] Michael Mateas and Andrew Stern. 2004. *Life-like Characters. Tools, Affective Functions and Applications.* Springer, Chapter A Behavior Language: Joint Action and Behavioral Idioms.

[3] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala* (1 ed.). Artima.

[4] Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems.* University of Cambridge, Department of Engineering.

[5] Christopher Simpkins, Sooraj Bhat, Charles Isbell, and Michael Mateas. 2008. Towards Adaptive Programming: Integrating Reinforcement Learning into a Programming Language. In *OOPSLA '08: ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Onward! Track.* Nashville, TN USA.