Balancing Formal and Practical Concerns in Agent Design

David Morley and **Karen Myers**
Artificial Intelligence Center, SRI International
333 Ravenswood Ave., Menlo Park, CA 94025
{morley,myers}@ai.sri.com

Abstract

Most agent frameworks can be readily characterized into one of two groups. On the one hand, there are frameworks designed to have well-defined formal semantics, enabling formal reasoning about agents and their behavior. On the other hand, there are frameworks targeted at developing agent systems that can be deployed as practical applications interacting in highly dynamic real-world environments. These frameworks have tended to sacrifice formal grounding for the ability to tackle real-world problems.

An ideal agent system should combine the sophisticated representations and control of the more practical systems with principled semantics that enable reasoning about system behavior. In this paper, we discuss a number of both formal and practical requirements that must be met to satisfy the demands of large-scale applications. We also briefly describe work on a new BDI agent framework that aims to bridge the gap between formal and practical agent frameworks. This system, called SPARK (the SRI Procedural Agent Realization Kit), is currently being used as the agent infrastructure for a personal assistant system for a manager in an office environment.

Introduction

Numerous agent frameworks have been published in the literature in recent years. These frameworks fall into two groups with distinctive characteristics.

The first group consists of formal agent frameworks that provide elegant, semantically grounded models of agent behavior (e.g., AgentSpeak(L) (Rao 1996), Golog/ConGolog (Giacomo, Lesperance, & Levesque 2000), and 3APL (Hindriks *et al*. 1998)). The semantic underpinnings and elegance of these models enable formal reasoning about current and potential agent activity to ensure that the agent meets appropriate criteria for correctness and well-behavedness. These frameworks were designed primarily to have well-defined formal properties and do not scale well for large applications that need to integrate with and control other systems. Applications of these frameworks have generally been to test problems that illustrate features of the design, rather than to real-world problems.

The second group consists of agent frameworks that have been designed to support demanding applications, such as PRS (Ingrand, Georgeff, & Rao 1992), PRS-Lite (Myers 1996), JAM (Huber 1999), RAPS (Firby 1989), dMARS (d'Inverno *et al*. 1997), and JACK (Busetta *et al*. 1999). These frameworks are necessarily more practical in nature, generally providing much more expressiveness in encoding and controlling agent behavior to meet the requirements of their motivating applications. This increased sophistication, however, has generally come at the cost of formal grounding, with the systems taking on the character of general-purpose programming environments that are not amenable to formal analysis. For example, the JACK Agent Language is an extension of Java that incorporates agent-oriented concepts, and it comes with a suite of tools for developing and deploying "commercial-grade" multiagent systems. However, since JACK is an extension of Java, if you want to reason about and synthesize JACK agents' plans, you would need to be able to reason about and construct Java programs.

Our background lies primarily with agent frameworks in the second category. However, our efforts to deploy such frameworks in challenging applications (e.g., real-time tracking (Garvey & Myers 1993), mobile robots (Konolige *et al*. 1997; Myers 1996), crisis action planning (Wilkins *et al*. 1995), intelligence gathering (Myers & Morley 2001), air operations (Myers 1999)) has made clear to us the need to be able to support formal reasoning about both the agent's knowledge and its execution state. These capabilities are essential for system validation, effective awareness of the current situation, and the ability to project into the future. Thus, an ideal agent system should combine the sophisticated representations and control of the more practical systems with principled semantics that enable reasoning about system behavior.

In this paper, we describe key formal and practical capabilities that we feel are essential for building successful, large-scale agent systems. While balancing formal and practical requirements can be a challenge, we believe that it is possible to build improved agent frameworks that address both concerns. To that end, we describe a new agent framework called SPARK (Morley & Myers 2004) that tries to address pragmatic issues related to scalability while building on a strong semantic foundation that can facilitate a variety of advanced reasoning capabilities.

Formal Capabilities

Our work on large-scale applications has made clear the need for a well-defined semantics for an agent's knowledge and execution model to enable various forms of reasoning about the agent and its capabilities. Two key challenges that motivate the need for such reasoning are *knowledge modeling* and *self-awareness*.

Knowledge Modeling

The success of an agent is linked closely to the adequacy of its knowledge for responding to stimuli (events, tasks) within its operating environment. Although numerous paradigms for representing such activity knowledge have been proposed, most of the practical agent frameworks have adopted a *procedural* representation of knowledge (Ingrand, Georgeff, & Rao 1992; Myers 1996; Huber 1999; Firby 1989; d'Inverno *et al*. 1997).

Formulating procedural representations of activity knowledge presents a significant modeling challenge. Current practice relies almost exclusively on handcrafting procedural knowledge – a practice that is both time-consuming and error-prone. The use of a formally grounded representation framework can help to address the knowledge modeling problem by enabling three types of reasoning: *verification and validation*, *automated procedure synthesis*, and *learning and adaptation*.

Verification and Validation Verification and validation methods can be applied to help identify shortcomings in handcrafted knowledge, including problems of *correctness* (e.g., Does the agent have the ability to attain a designated goal?) and *well-behavedness* (e.g., Will the agent avoid nonrecoverable failure states? Will it avoid undesirable behaviors such as thrashing and deadlock?).

There is a rich body of work on verification and validation methods that can be applied to these problems. However, much of that work has assumed simpler control constructs than are found in current agent representations of knowledge. Additionally, more research is needed to address issues such as world-state dynamics and complex temporal interactions.

Automated Procedure Synthesis The rich body of work on generative planning provides a starting point for automating aspects of the procedure modeling problem. In particular, recent work has started to address more practical models that incorporate concepts such as metric resources and time, which occur commonly in many applications (Smith, Frank, & Jónsson 2000). While there is hope that procedural knowledge could be synthesized automatically from declarative models of primitive activities using plan synthesis technologies, we note that the complexity of typical agent procedures is well beyond the state of the art in planning. Still, automated plan synthesis methods could play a role in helping to formulate limited forms of procedural knowledge.

Learning and Adaptation Automated learning methods cannot in isolation solve the knowledge modeling problem for large-scale applications for two reasons. First, the temporal extent and complexity of the procedures lie well beyond what is possible or even envisioned with current methods. Second, the most successful learning methods to date are grounded in statistical techniques that require large numbers of training cases to support appropriate generalization. In contrast, many of the nuances in agent knowledge relate to special cases that occur infrequently.

Learning can, however, play an important role in addressing the knowledge modeling problem, especially when focused on adapting existing procedural knowledge to handle unexpected situations that arise at runtime.

Self-awareness

Self-awareness refers to an agent's ability to introspect on its own knowledge and activities to understand its operation. As agent systems grow in complexity, self-awareness is essential to ensuring appropriate agent execution. Here, we consider three capabilities related to self-awareness: *temporal projection*, *explanation*, and *knowledge limits*.

Temporal Projection Much of the focus in agent design today is on reactive methods that respond to situation changes by deciding what to do next based on local criteria. Such approaches are not suitable for temporally extended problems, however, where good short-term decisions may be poor long-term decisions. A self-aware agent should be cognizant of its longer-term objectives and ensure that local decisions are consistent with achieving those objectives.

Temporal projection will be an important technique in providing such a long-term outlook. Given a formal representation of an agent's objectives, commitments, and knowledge, temporal projection supports the ability to reason into the future to ensure that objectives can be met and undesirable consequences avoided.

Explanation Agents will operate on behalf of a human and so must be able to justify their actions to him or her. Explanation of activity requires a clean formal model of agent activity and knowledge, to enable clear communication to a human.

Knowledge Limits Robust operation requires that agents be able to recognize when they are in situations that lie outside of their capabilities. Otherwise, agents may undertake activities that are ill-advised and possibly even dangerous. Such recognition requires that agents be able to reflect on the limitations of their problem-solving knowledge and execution abilities relative to the current operational context.

Practical Capabilities

The development of large agent systems that operate in real-world environments requires certain practical capabilities

not found in many current agent frameworks. We consider several such capabilities here.

Failure Handling Failure is an inevitable part of real-world interactions, especially when agents operate in dynamic and unpredictable environments. Virtually any action that an agent performs may be subject to unexpected failure. Thus, failure handling is something that the agent framework should support at a fundamental level in a principled way, as opposed to current ad hoc practices, that rely on hand-compiled responses.

Modularity Modularity is extremely important when constructing large knowledge bases, just as is the case when developing large programs. In particular, there may be multiple individuals defining knowledge, or knowledge could be imported from multiple preexisting sources. Ideally, knowledge bases should employ some mechanism for defining multiple namespaces to reduce the potential for naming conflict.

Scalability When dealing with a knowledge base that contains large numbers of facts, the efficiency of access to the knowledge base can become a limiting factor. It may not be feasible for the agent to keep its entire knowledge base in an internal knowledge store at runtime, in which case the framework must be able to integrate knowledge kept in external sources.

Integration Real-world agent systems will be embedded in rich environments and will need to interact with multiple external components. In many large agent systems the role of the agents is to connect together and manage the external systems. For this reason, agents should be able to directly execute external code with a minimum of interfacing overhead and effort.

Multiple Timescales Different agents need to operate on different timescales. At one extreme lie task-level control systems for managing the low-level actions of robots, where fast real-time responses are essential. At the other extreme lie agents that manage workflow processes, some of which can span months. Ideally, an agent framework should be capable of covering these extremes, possibly using different implementations for the different timescales but relying on a common representation and semantics.

Persistence When an agent is responsible for managing processes that endure for extended periods of time, it must handle not only the failure of the actions that it initiates, but also the unexpected termination of agent processes. Thus, the agent should be capable of persisting beyond the extent of an individual process and even over time periods during which the host machine may need to reboot. This requires the ability for the agent to save and restore its state.

Programmer Support As agent systems grow in complexity, the effort involved in managing their development increases. To construct large agent systems, an agent framework should provide tools to support the development process, including an interactive development environment, static analysis tools, refactoring tools, and runtime debugging tools.

SPARK

The need to jointly address the above theoretical and practical requirements has led us to develop a new agent framework called SPARK (the SRI Procedural Agent Realization Kit). SPARK is an open-source agent framework, built on Python, that aims to bridge the gap between the formal agent frameworks and the practical agent frameworks.

SPARK builds on a Belief-Desire-Intention (BDI) model of rationality that has been heavily influenced by the PRS family of agent systems (Ingrand, Georgeff, & Rao 1992). It provides a flexible plan execution mechanism that interleaves goal-directed activity and reactivity to changes in its execution environment. However, in contrast to the representations typically found in most practical agent systems, SPARK's procedural language has a clear, well-defined formal semantics that is intended to support reasoning techniques for procedure validation, synthesis, learning, and repair.

Here we provide a brief overview of SPARK. Additional details can be found in (Morley & Myers 2004).

SPARK Overview

Figure 1 depicts the overall architecture for a SPARK agent. Each *agent* maintains a *knowledge base* (KB) of beliefs about the world and itself that is updated both by sensory input from the external world and by internal events. The agent has a library of *procedures* that describe ways of responding to changes and ways of hierarchically decomposing goals. The agent has a set of *advice*, provided by a user, that helps guide the agent when making decisions. At any given time the agent has a set of *intentions*, which are procedure instances that it is currently executing. The hierarchical decomposition of goals bottoms out in *primitive actions* that instruct effectors to interact with the outside world in which the SPARK agent is embedded.

Each procedure has a *cue* that specifies the goal or event that the procedure responds to, a *precondition* that specifies conditions under which that procedure can be used, and a *body* that specifies the response. The body is in the form of a *task expression* constructed from *basic tasks* such as performing primitive actions, posting goals, and making explicit changes to the KB. The basic tasks are combined using task operators, such as parallel composition, iteration, and so on. Logical variables (of the form `$varname`) link the components. For example, Figure 2 shows a procedure that can be triggered by a fact of the form `(Order $id $user $item)` becoming true.

At SPARK's core is the *executor*, whose role is to manage the execution of intentions. It does this by repeatedly selecting a procedure instance that is ready to progress from

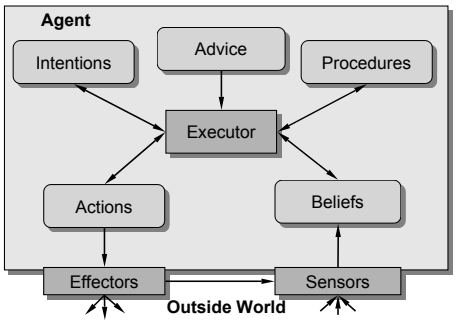

Figure 1: SPARK agent architecture

```
{defprocedure "Monitor Orders"
  cue:
    [newfact: (Order $id $user $item)]
  precondition:
    (and (Expensive $item)
         (Supervisor $user $boss))
  body:
    [parallel:
      [do: (inform $boss $id)]
      [conclude: (UnderReview $id)]]}
```

Figure 2: Example SPARK procedure

the current intentions, performing a single step of that procedure instance, and updating the set of intentions to reflect the changes caused by step execution and sensory input.

In (Morley & Myers 2004), we define the semantics of a SPARK agent's behavior in terms of finite state machines (FSMs). An agent's KB is interpreted as a set of ground atomic literals where KB changes are simple additions and deletions of ground atomic literals. The intention structure is interpreted as a set of FSMs, one per intended procedure instance together with the current state of each FSM. The FSM corresponding to each procedure instance is derived from the body task expression of the procedure. The execution of an agent can be modeled quite directly as the interleaved execution of a set of FSMs where that set changes as a consequence of the execution of the FSMs and sensory input KB changes.

Motivating Application: CALO

The need to address the formal and practical issues raised in the previous sections is well highlighted by the current driving domain for SPARK, which involves the development of an intelligent personal assistant for a high-level knowledge worker. This assistant, called CALO (Cognitive Assistant that Learns and Observes), will be able to perform routine tasks on behalf of its user (e.g., arrange meetings, complete online forms, file email), as well as undertake open-ended processes (e.g., purchasing a computer online), and anticipate future needs of its user.

At the heart of CALO is a *task manager* that initiates, tracks, and executes activities and commitments on behalf of its user, while remaining responsive to external events. The task manager is capable of fully autonomous operation (i.e., for tasks that are delegated completely by the user), but can also operate in a mixed-initiative fashion when the user prefers to be more involved in task execution. The desired level of autonomy can be specified by the user through *guidance* mechanisms (Myers & Morley 2003).

To date, task manager development has focused on managing the user's calendar, performing certain routine tasks (e.g., email management), and supporting the computer purchase capability.

Being part of a large-scale system deployed in the real world, the task manager benefits from the practical aspects of SPARK; however, CALO also requires that it be possible to reason about the procedures being used. For example, currently in one scenario the task manager is executing a procedure for purchasing a computer, but it becomes apparent that an authorization step in the procedure is not succeeding quickly enough for the purchase to go through in time. This triggers a request to another component to reason about the current procedure to see if there are any changes to the procedure that can be made to get around the problem. Future scenarios will involve reasoning about how well existing procedures are working to try to learn new procedures. Without a well-defined formal model of the procedures, such reasoning is extremely difficult.

Balancing Practical and Formal Concerns in SPARK

With SPARK, we are attempting to provide a formally grounded framework that can support the development of practical agent systems.

On the formal side, our efforts to date have focused on foundations for knowledge modeling and self-awareness, including a formal model of execution, introspective capabilities, and the ability to construct new procedures on-the-fly at runtime. We have not yet started work on techniques related to verification and validation, procedure synthesis, procedure learning, and explanation, but plan to do so in the future.

On the practical side, where our main emphasis has been, we have been taking care to implement the desired capabilities in a way that is consistent with the requirements for the formal grounding.

For example, SPARK has predicates and logical variables that allow a logic-based interpretation of many of the representation constructs. However, SPARK does not use full unification in the style of logical programming languages such as Prolog. Instead it uses a restricted form of pattern matching. As a result, SPARK data values are always

fully instantiated and there is no need for variable dereferencing or explicit unbinding of variables. The benefits of this approach include a simpler implementation and easier integration of SPARK with other procedural, functional, and object-oriented programming languages. The fact that SPARK is unable to pass around partially instantiated data structures means that some logic programming idioms are not possible in SPARK, but it does not affect the logical semantics of the language.

SPARK provides much finer control over failure handling than its predecessor, PRS, and does this in a way that is well-grounded formally. The concept of failure is fundamental within the finite state machine model of SPARK execution. SPARK provides specific task operators to deal with task failure and meta-level events that can trigger procedures that implement customized failure responses. More research is required into different ways of specifying failure recovery, but the existing SPARK language constructs and formal model provide a good starting point.

SPARK also provides better control over race conditions The semantic model places strict limits on changes in the knowledge base that can occur between conditions being tested (e.g., in conditional task operators or procedure preconditions) and subsequent actions.

For modularity, SPARK uses a hierarchical namespace for functions, predicates, actions, and constants. Similar to the Java and Python programming languages, SPARK uses a correspondence between the module hierarchy and the hierarchical file system to locate definitions of imported symbols. For example, when a module imports a predicate `a.b.SomePred` from module `a.b`, SPARK knows to look for the definition of symbols in that module in a file `a/b.spark` relative to a given search path.

SPARK's treatment of variables and the finite state machine model of execution facilitate integration of SPARK with external components and allow for simple and efficient implementations. At the same time, this execution model is more amenable to formal analysis than, for example, the extended Java programs of JACK. The language and execution model were designed to allow a wide range of diverse implementations:

- The current implementation of SPARK is an interpreter written in Python. Python was chosen because it is an excellent rapid prototyping language that is widely used, with an open-source implementation available on a large number of platforms. The Jython implementation of Python compiles to the Java Virtual Machine and makes the integration of Python code and Java code seamless. This ability has facilitated the integration of SPARK with other systems such as OAA (Cheyer & Martin 2001) and SHAKEN (Barker, Porter, & Clark 2001).
- By taking out the meta-level capabilities and restricting the use of recursive predicates, it is feasible for SPARK code to be compiled into very tight, efficient code in target languages such as C and C++ that could be used for applications in which fast response time is critical (e.g., robot control).
- When dealing with very large knowledge bases or the need for persistent agents, it is feasible for SPARK to store its entire knowledge base in an external database and still be compatible with the execution model.

For programmer support, SPARK includes an Integrated Development Environment (IDE) built on top of IBM's open-source Eclipse Platform (Object Technology International, Inc. 2003). The IDE provides editing and debugging capabilities, with access to the internal state of execution of the SPARK agents. We plan to extend SPARK's current static analysis of source code to include support for an optional type system.

Conclusion

Current agent frameworks will need to evolve to meet the requirements inherent to large-scale applications. While many of those requirements relate to practical concerns, it is clear that advanced reasoning techniques grounded in strong formal models will also play an essential role. Our work on SPARK constitutes an attempt to develop an agent framework that combines the best of both worlds. While still a work in progress, our vision for SPARK is to combine the strengths of a principled formal model with associated reasoning capabilities and a practical design that will scale to the demands of real applications.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010 and by internal funding from SRI International. The authors thank others who have contributed to the development of SPARK, including Mabry Tyson, Alyssa Glass, Ken Conley, Regis Vincent, Andrew Agno, and Michael Ginn.

References

Barker, K.; Porter, B.; and Clark, P. 2001. A library of generic concepts for composing knowledge bases. In *Proc. of the First Int. Conf. on Knowledge Capture (K-Cap'01)*.

Busetta, P.; Rönnquist, R.; Hodgson, A.; and Lucas, A. 1999. JACK - components for intelligent agents in Java. Technical Report 1, Agent Oriented Software Pty. Ltd. http://www.agent-software.com.

Cheyer, A. J., and Martin, D. L. 2001. The Open Agent Architecture. *Journal of Autonomous Agents and Multi-Agent Systems* 4:143–148.

d'Inverno, M.; Kinny, D.; Luck, M.; and Wooldridge, M. 1997. A formal specification of dMARS. In *Agent Theories, Architectures, and Languages*, 155–176.

Firby, R. J. 1989. *Adaptive execution in complex dynamic worlds*. Ph.D. Dissertation, Yale University CS Dept. Technical Report RR-672.

Garvey, T., and Myers, K. 1993. The intelligent information manager. Final Report SRI Project 8005, Artificial Intelligence Center, SRI International, Menlo Park, CA.

Giacomo, G. D.; Lesperance, Y.; and Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1-2):109–169.

Hindriks, K.; de Boer, F.; van der Hoek, W.; and Meyer, J.-J. 1998. Formal semantics for an abstract agent programming language. In *Intelligent Agents IV: Proc. of the Fourth Int. Workshop on Agent Theories, Architectures and Languages, LNAI 1365*. Springer-Verlag. 215–229.

Huber, M. J. 1999. JAM: A BDI-theoretic mobile agent architecture. In *Proc. of the Third Int. Conf. on Autonomous Agents (Agents '99)*, 236–243.

Ingrand, F. F.; Georgeff, M. P.; and Rao, A. S. 1992. An architecture for real-time reasoning and system control. *IEEE Expert* 7(6).

Konolige, K.; Myers, K.; Ruspini, E.; and Saffiotti, A. 1997. The Saphira Architecture: A design for autonomy. *Journal of Experimental and Theoretical AI* 9.

Morley, D. N., and Myers, K. L. 2004. The SPARK agent framework. In *Proc. of the Third Int. Joint Conf. on Autonomous Agents and Multi-agent Systems (AAMAS-2004)*.

Myers, K. L., and Morley, D. N. 2001. Human directability of agents. In *Proc. of the First Int. Conf. on Knowledge Capture*.

Myers, K. L., and Morley, D. N. 2003. The TRAC framework for agent directability. In Hexmoor, H.; Falcone, R.; and Castelfranchi, C., eds., *Adjustable Autonomy*. Kluwer Academic Publishers.

Myers, K. L. 1996. A procedural knowledge approach to task-level control. In *Proc. of the Third Int. Conf. on AI Planning Systems*. AAAI Press.

Myers, K. L. 1999. CPEF: A continuous planning and execution framework. *AI Magazine* 20(4).

Object Technology International, Inc. 2003. Eclipse platform technical overview. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

Rao, A. S. 1996. AgentSpeak(L): BDI agents speak out in a logical computable language. In de Velde, W. V., and Perram, J. W., eds., *Agents Breaking Away, Lecture Notes in Artificial Intelligence, Volume 1038*. Springer-Verlag. 42–55.

Smith, D.; Frank, J.; and Jónsson, A. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review* 15(1).

Wilkins, D. E.; Myers, K. L.; Lowrance, J. D.; and Wesley, L. P. 1995. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI* 7(1):197–227.