The SPARK Agent Framework

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

Abstract

There is a need for agent systems that can scale to real-world applications, yet retain the clean semantic underpinning of more formal agent frameworks. We describe the SRI Procedural Agent Realization Kit (SPARK), a new BDI agent framework that combines these two qualities.

In contrast to most practical agent frameworks, SPARK has a clear, well-defined formal semantics that is intended to support reasoning techniques such as procedure validation, automated synthesis, and procedure repair. SPARK also provides a variety of capabilities such as introspection and meta-level reasoning to enable more sophisticated methods for agent control, and advisability techniques that support user directability. On the practical side, SPARK has several design constructs that support the development of large-scale agent applications.

SPARK is currently being used as the agent infrastructure for a personal assistant system for a manager in an office environment.

1. Introduction

Numerous agent frameworks have been published in the literature in recent years. These frameworks can be categorized into two groups with very distinctive characteristics.

The first group consists of the formal agent frameworks that provide elegant, semantically grounded models of agent behavior (e.g., AgentSpeak(L) [18], Golog/ConGolog [8], and 3APL [9]). The semantic underpinnings 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. 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 (e.g., PRS [11], PRS-Lite [13], JAM [10], RAPS [5], dMARS [4], and JACK [2]). These frameworks are more practical in nature, generally providing much more expressive mechanisms for 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.

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 [7], mobile robots [12, 13], crisis action planning [20], intelligence gathering [15], air operations [14]) 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.

These requirements have motivated us to develop a new agent framework called SPARK (the SRI Procedural Agent Realization Kit) that attempts to combine the best of both worlds. SPARK builds on a Belief-Desire-Intention (BDI) model of rationality [19] that has been heavily influenced by the PRS family of agent systems. SPARK provides a flexible plan execution mechanism that interleaves goal-directed activity and reactivity to changes in its execution environment. In contrast to the representations typically found in most practical agent systems, SPARK's procedural language has a clear, well-defined formal semantics that can support reasoning techniques for procedure validation, synthesis, and repair.

SPARK also contains a number of features and capabilities to support its use in constructing large-scale agent applications in rich and dynamic domains. On the conceptual side, these include an expressive procedure language with a range of control structures, a powerful introspective capability, rich metalevel control, and an ability to support

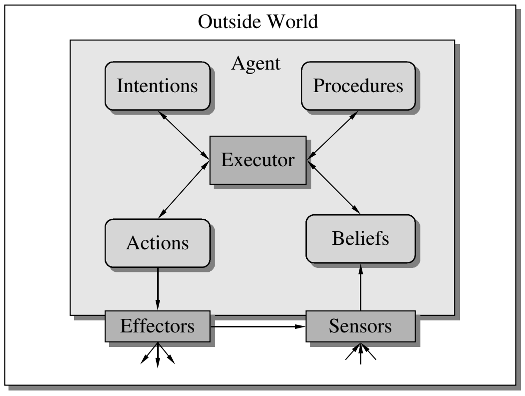

Figure 1. SPARK Agent Architecture

high-level user directability of agent activity. On the engineering side, the execution model and treatment of variables facilitate integration with external components. They also allow for a range of implementations from short-timescale reactive control systems to long-timescale workflow systems. Other features motivated by practicality include a module system, a well-defined failure mechanism, the ability to adapt behavior at runtime, and an integrated development environment (IDE) based on Eclipse [17].

This paper provides a high-level overview of SPARK's design. Sections 2, 3, and 4 outline the SPARK architecture, language, and semantic model respectively. Section 5 explores some of the advanced features of SPARK (including the meta-level and agent guidance). Section 6 describes the balance between formal and practical concerns in the system. Section 7 describes a current use of SPARK as the basis for an intelligent personal assistant designed to improve the productivity of a manager-level office worker. Finally, Section 8 compares SPARK with other agent systems.

2. SPARK

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 provide declarative representations of activities for responding to events and for decomposing complex tasks into simpler tasks. At any given time the agent has a set of *intentions*, which are procedure instances that it is currently executing. The hierarchical decomposition of tasks bottoms out in *primitive actions* that instruct *effectors* to bring about some change in the outside world or the internal state of the agent.

At SPARK's core is the *executor* whose role is to manage the execution of intentions. It does this by repeatedly selecting one of the current intentions to process and performing a single step of that intention. Steps generally involve activities such as performing tests on and changing the KB, adding tasks, decomposing tasks by applying procedures, or executing primitive actions.

3. Language Constructs

The specification of a SPARK agent consists of declarations of constant, function, predicate, and action symbols, together with facts that give the initial state of extensional predicates and procedures that define how to respond to events. The key syntactic structures are term expressions, logical expressions, actions, task expressions, and procedures.

A *term expression*, ν, represents a value. Atomic term expressions are constants such as `42`, `"Hi"`, and `pi`, and variables of the form `$x`. These are combined to form compound term expressions, including lists (e.g., `[1 2 3]`), function applications (e.g., `(- $x 1)`), and closures (e.g., `{fun [$x] (- $x 1)}`). Closures represent parameterized term, logical, and task expressions that can be passed around as first-order values.

A *logical expression*, ϕ, is constructed from predicate symbols applied to term expressions (e.g., `(p 1 $x)`), logical operators (e.g., `(not (q))`, `(and (q) (r))`, `(or (q) (r))`), and existential quantifiers (e.g., `(exists [$x $y] (p $x $y))`). In SPARK, the testing of a logical expression binds previously unbound variables to values that make the expression true.

An *action*, α, is constructed from an *action* symbol and term expressions as parameters (e.g., `(send $authorization bill)`). The action may be a *primitive action*, in which case it is performed by executing some procedural attachment, or a *nonprimitive action*, in which case procedures describe how to hierarchically decompose the action into subtasks.

A *task expression*, τ, is an activity that when attempted may either succeed or fail. Tables 1 and 2 summarize SPARK's task expressions (simplified somewhat for the sake of presentation). The basic task expressions include trying to perform some action and trying to bring about or "achieve" the truth of some atomic logical expression if it is not already true. The compound task expressions are constructed from basic task expressions using *task operators*. In each of the compound task expressions (with the sole exception of τ in `[try:` τ τ_1 τ_2`]`) if one of the component task expressions fails then the compound task expression then fails. Also, each test of a condition ϕ is intended to remain valid up to the start of execution of the appropriate task expression. For example, at the time `[while:` ϕ τ_1 τ_2`]` starts

`[noop:]`	Do nothing.
`[fail:]`	Fail.
`[conclude:` ϕ`]`	Add fact ϕ to the KB.
`[retract:` ϕ`]`	Remove facts matching ϕ from the KB.
`[do:` α`]`	Perform the action α.
`[achieve:` ϕ`]`	Attempt to make ϕ true.

Table 1. Basic Task Expressions

`[seq:` $\tau_1\ \tau_2$`]`	Execute τ_1 and then τ_2.
`[parallel:` $\tau_1\ \tau_2$`]`	Execute τ_1 and τ_2 in parallel.
`[if:` $\phi\ \tau_1\ \tau_2$`]`	If ϕ is true, execute τ_1 otherwise execute τ_2.
`[try:` $\tau\ \tau_1\ \tau_2$`]`	If τ succeeds, execute τ_1, otherwise execute τ_2.
`[wait:` $\phi\ \tau$`]`	Wait until ϕ is true, then execute τ.
`[while:` $\phi\ \tau_1\ \tau_2$`]`	Repeat τ_1 until ϕ has no solution then execute τ_2.

Table 2. Compound Task Expressions

executing τ_2, ϕ is guaranteed to be false. The exact meaning of this will be made clear in Section 4.2.

At its simplest, a SPARK *procedure* has the form $\{$`defprocedure` *name* `cue:` *event* `precondition:` ϕ `body:` $\tau\}$. This indicates that if ϕ is true when *event* occurs, then executing τ is a valid way of responding. The cue event may be of the form `[newfact:` ϕ`]` to respond to the fact ϕ being added to the KB, or `[achieve:` ϕ`]` or `[do:` α`]` to respond to a new task. The procedure in Figure 2 specifies that whenever a fact of the form `(order $id $user $item)` becomes true, where `$item` is expensive and `$user`'s boss is `$boss`, then the action `(inform $boss $id)` should be performed.

4. Semantics

The *cognitive state* of an agent at any time is given by its KB and a set of active procedure instances called the *intention structure*. We model the task expressions that form the bodies of procedures by finite state machines (FSMs). This allows us to model the intention structure by a set of *FSM instances*, one per intended procedure instance. Each FSM instance is an FSM plus an *FSM state* (abbreviated to *state* where unambiguous) indicating the current execution state of that FSM.

```
{defprocedure "PlaceOrder"
  cue: [newfact: (order $id $user $item)]
  precondition: (and (Expensive $item)
                     (Supervisor $user $boss))
  body: [do: (inform $boss $id)]}
```

Figure 2. An Example Procedure

Both sensory input and procedure execution can change the state of the KB, which in turn can cause new procedure instances to be added to the intention structure. The dynamics of an agent are modeled by transitions on the KB and intention structure models. Hierarchical task expansion is modeled as assertions of "desires" into the KB that trigger the addition of new FSM instances. On completion of these FSM instances, the KB is updated to record their success or failure. For primitive actions, we do not explicitly model the action of the effectors. Instead we assume that the effectors are triggered by the existence of the "desires" and that the success or failure of the actions is recorded by sensory input.

4.1. KB Semantics

An agent's KB is a set of ground literals. A ground literal is considered true with respect to the KB if and only if it is an element of that set. We evaluate the truth of a logical expression, given an assignment of values to free variables, in the standard way. KB changes are simple additions and deletions of ground literals.

4.2. Task Expressions as State Machines

Each task expression, τ, is interpreted as an FSM, or more precisely a family of FSMs equivalent under state renaming. Each transition of the FSM is labeled with *guard* conditions that test the KB and *effects* that modify the KB. The set of FSM state names, Σ, includes the *distinguished states*:

σ°	the initial state
σ^+	the success state
σ^-	the failure state.

An FSM, M, is defined to be a set of transitions, each of the form $\sigma_s \xrightarrow{\vec{c}|\vec{e}} \sigma_t$ where $\sigma_s \in \Sigma \setminus \{\sigma^+, \sigma^-\}$ is the source state; $\sigma_t \in \Sigma \setminus \{\sigma^\circ\}$ is the target state; $\vec{c} = c_1 ... c_n$ is a sequence of conditions of the form $c_i = \phi$ or $\bar{\phi}$ where ϕ is a logical expression; and $\vec{e}$ is a sequence of effects of the form ψ or $\bar{\psi}$ where ψ is an *atomic* logical expression. ε denotes an empty condition sequence or effect sequence. The FSM $\{\sigma^\circ \xrightarrow{\varepsilon|\vec{e}} \sigma^+\}$ is so common that we abbreviate it as $M^{\vec{e}}$. A transition is *allowed* if all conditions ϕ are true with respect to the KB at the time of the transition and all conditions $\bar{\phi}$ are not true.

Note that uniformly renaming the non-distinguished states in an FSM yields a new FSM that is functionally identical to the original. We will use an individual FSM, M, to represent the equivalence set $|\tau|$ of identical FSMs corresponding to a task expression, τ, writing $M = |\tau|$ rather than $M \in |\tau|$.

To define the FSMs corresponding to compound task expressions, we introduce the following operators.

- $\phi{::}M$ prepends the condition ϕ to the conditions of each transition in M leading from the initial state, σ°.
- $[\sigma]\, M$ replaces all occurrences of σ° with σ.
- $M\,\lceil\sigma\rceil$ replaces all occurrences of σ^+ with σ.
- $M\,\lfloor\sigma\rfloor$ replaces all occurrences of σ^- with σ.
- $M_1 \cup M_2$ contains all transitions from both M_1 and M_2 where the states of M_1 and M_2, other than the distinguished states and those explicitly introduced, have been uniformly renamed to ensure that they are disjoint.

We can now specify the subset of possible FSMs that correspond to SPARK task expressions. We require the following three well-behavedness conditions on such FSMs: (i) σ° has only outgoing transitions, (ii) for any KB state at least one outgoing transition from σ° must be allowed, (iii) σ^+ and σ^- are the only states that have no outgoing transitions.

- $|\texttt{[noop:]}| = \{\sigma^\circ \xrightarrow{\varepsilon|\varepsilon} \sigma^+\} = M^\varepsilon$ There is a single transition from the initial state to the success state with an empty condition list and no effect on the KB.

- $|\texttt{[fail:]}| = \{\sigma^\circ \xrightarrow{\varepsilon|\varepsilon} \sigma^-\}$ There is a single transition from the initial state to the failure state with no effect on the KB.

- $|\texttt{[conclude:}\ \psi\texttt{]}| = \{\sigma^\circ \xrightarrow{\varepsilon|\psi} \sigma^+\} = M^\psi$ There is a single transition from the initial state to the success state with an empty condition list and the effect of adding ψ to the KB.

- $|\texttt{[retract:}\ \psi\texttt{]}| = \{\sigma^\circ \xrightarrow{\varepsilon|\bar{\psi}} \sigma^+\} = M^{\bar{\psi}}$ There is a single transition from the initial state to the success state with an empty condition list and the effect of removing ψ from the KB if it is present.

- $|\texttt{[do:}\ \alpha\texttt{]}| = \{\sigma^\circ \xrightarrow{\varepsilon|(d\ \alpha)} \sigma, \sigma \xrightarrow{(s\ \alpha)|\varepsilon} \sigma^+, \sigma \xrightarrow{(f\ \alpha)|\varepsilon} \sigma^-\}$ The desire to perform action α is posted by asserting $(d\ \alpha)$ into the KB. The FSM then waits in state σ until either the success $(s\ \alpha)$ or failure $(f\ \alpha)$ of the action has been reported.

- $|\texttt{[achieve:}\ \phi\texttt{]}| = \{\sigma^\circ \xrightarrow{\phi|\varepsilon} \sigma^+, \sigma^\circ \xrightarrow{\bar{\phi}|(d\ \phi)} \sigma, \sigma \xrightarrow{(s\ \phi)|\varepsilon} \sigma^+,$ $\sigma \xrightarrow{(f\ \phi)|\varepsilon} \sigma^-\}$ Here the desire to achieve ϕ is posted only if the condition ϕ is not already satisfied.

- $|\texttt{[seq:}\ \tau_1\ \tau_2\texttt{]}| = |\tau_1|\,\lceil\sigma\rceil \cup [\sigma]\,|\tau_2|$ where the success state of $|\tau_1|$ and the initial state of $|\tau_2|$ are replaced by a new state σ joining the two FSMs. Note that if there are no transitions to the success state of $|\tau_1|$, then it will be impossible to reach any of the states in $|\tau_2|$.

- $|\texttt{[if:}\ \phi\ \tau_1\ \tau_2\texttt{]}| = \phi{::}|\tau_1| \cup \bar{\phi}{::}|\tau_2|$ where the condition ϕ is added to all initial transitions of $|\tau_1|$ and $\bar{\phi}$ is added to those of $|\tau_2|$. Note that rather than sequentially performing the test and then executing the appropriate task expression, we instead require the appropriate condition, ϕ or $\bar{\phi}$, to hold at the time τ_1 or τ_2 performs its first transition, thus eliminating the possibility of something changing the truth of ϕ between the test and the task execution. This provides an important transactional capability that many agent systems lack.

- $|\texttt{[try:}\ \tau\ \tau_1\ \tau_2\texttt{]}| = |\tau|\,\lceil\sigma_1\rceil\,\lfloor\sigma_2\rfloor \cup [\sigma_1]\,|\tau_1| \cup [\sigma_2]\,|\tau_2|$ where the success state of $|\tau|$ and initial state of $|\tau_1|$ are replaced by a new state σ_1, and the failure state of $|\tau|$ and the initial state of $|\tau_2|$ are replaced by a new state σ_2.

- $|\texttt{[wait:}\ \phi\ \tau\texttt{]}| = M^\varepsilon\,\lceil\sigma\rceil \cup [\sigma]\,\phi{::}|\tau|$ where there is an initial transition to an internal state σ after which it is possible to transition into $|\tau|$ only if ϕ is true. The initial M^ε transition exists to satisfy well-behavedness condition (ii).

- $|\texttt{[while:}\ \phi\ \tau_1\ \tau_2\texttt{]}| = M^\varepsilon\,\lceil\sigma\rceil \cup [\sigma]\,\phi{::}|\tau_1|\,\lceil\sigma\rceil \cup [\sigma]\,\bar{\phi}{::}|\tau_2|$ where there is an initial transition to an internal state σ at which point, if ϕ is true, τ_1 will be executed returning to σ; otherwise τ_2 will be executed. The initial M^ε transition exists to satisfy well-behavedness condition (i).

For simplicity in this paper, we have restricted the above semantic model of SPARK task expressions to atomic term expressions (i.e., no closures, list, or functions) and we have ignored the housekeeping involved in managing different "do" and "achieve" task instances. We have skipped the semantics for tasks of the form $\texttt{[parallel:}\ \tau_1\ \tau_2\texttt{]}$ due to the complexity of treating failed subtasks.

We have also not described the treatment of free variables. Allowing free variables requires extending the execution state of the FSM to include a set of variable bindings, mapping variables to values. Roughly speaking, a transition is allowed if there is a set of variable bindings consistent with the existing variable bindings that makes the positive transition conditions true. When a transition is performed, one of the possible sets of bindings is selected and those variable bindings are added to the execution state of the FSM. In this way, the set of variable bindings grows progressively as the FSM is executed.

4.3. Execution Semantics

The execution of an agent consists of repeatedly processing sensory input, stepping procedure instances, and dropping completed procedure instances. Each of these activities can change the KB, which in turn can cause new procedure instances to be added to the intention structure.

We model the application of changes $\vec{e}$ to a KB by the function $K' = apply(\vec{e}, K)$ that adds or removes atomic facts atomic facts from K as required. We model the intention structure by a set of FSM instances, being pairs of an FSM and an FSM state, $I = \{\langle M^1, \sigma^1\rangle \ldots \langle M^k, \sigma^k\rangle\}$. The consequential changes to the intention structure caused by a change in the KB are then modeled by a set of FSM instances to be added to the intention structure, calculated as follows.

Given a set of procedures, a function $appl(K', \phi)$ determines a set of applicable FSMs $\{M_1, \ldots, M_n\}$ that could be intended when a fact ϕ has been added to produce the new KB K'. We collect the FSMs $M_i = |\tau_i|$ derived from the body τ_i of each procedure whose cue matches ϕ and whose precondition is satisfied by K'. The cue `[do:` α`]` matches $(d\ \alpha)$, `[achieve:` ϕ'`]` matches $(d\ \phi')$, and `[newfact:` ϕ`]` matches anything else.

A selection function, $select(appl(K', \phi), K', \phi)$ determines which of these FSMs get added to the intention structure. The default selection function acts as follows, depending upon whether ϕ is (i) a desire to perform a non-primitive action $(d\ \alpha)$ or to achieve a condition $(d\ \phi')$, or (ii) some other new fact.

(i) If $appl(K', (d\ x))$ is empty, a "failure" FSM $M^{(f\ x)}$ is added. Otherwise, a single FSM is selected randomly.

(ii) All FSMs in $appl(K', \phi)$ are used.

The change-inducing steps in the execution of an agent can be modeled by a fair interleaving of the following operations on the KB and intention structure.

Sensory Input applies a set of changes to the KB.

Task Execution selects an FSM instance $\langle M, \sigma\rangle \in I$ with a transition $\sigma \xrightarrow{\vec{c}|\vec{e}} \sigma'$ that is allowed in K. The state of the FSM instance is updated $I' = (I \setminus \{\langle M, \sigma\rangle\}) \cup \{\langle M, \sigma'\rangle\}$ before the changes $\vec{e}$ are made.

Task Removal selects a completed FSM instance $\langle M, \sigma\rangle \in I$ where $\sigma \in \{\sigma^+, \sigma^-\}$ and removes it: $I' = I \setminus \{\langle M, \sigma\rangle\}$. If the FSM was for a desire $(d\ x)$ the appropriate fact $(s\ x)$ or $(f\ x)$ is added to the KB.

Each of these steps is immediately followed by the addition of FSM instances $\langle M, \sigma^\circ\rangle$ for each $M \in select(appl(K', \phi), K', \phi)\}$ where ϕ has become true (i.e., $\phi \in K' \backslash K$).

5. Reflective Capabilities / Advanced Features

SPARK provides two introspective capabilities (influenced heavily by concepts in PRS) that endow each agent with a level of "self-awareness" of its own execution state. One capability is a set of predicates and actions that provide access to an agent's intention structure. The second capability is a set of *meta-predicates* that track significant execution events, including the existence of multiple candidate procedure instances for a task, and the successful or failed completion of a task. The value of these meta-predicates is that they can be used to trigger *meta-procedures* for modifying system behavior. Examples of current usage of such meta-procedures within SPARK are given below.

- Meta-predicates can trigger the logging of information about the agent's state and progress of execution.
- Customized failure response can be achieved through meta-procedures that respond to task failure meta-predicates.
- The default procedure selection mechanism (Section 4.3) can be replaced by alternatives. For example, meta-level procedures can be used to select between the applicable procedures on the basis of advice supplied by the user. More sophisticated meta-level predicates might be used to indicate that a new procedure is required and request one from the user, or from a planning or reasoning system.

5.1. Agent Guidance

One unique feature of SPARK is its built-in capability to support directability of an agent through user-specified *guidance*. This guidance, which is represented as a set of declarative policies, can be used to explicitly bound the activities that an agent is allowed to perform without user involvement. Guidance is expressed in a high-level language that shields the user from having to know the details of the system's underlying knowledge or its execution state at any point in time.

Guidance could be defined from the perspective of either explicitly enabling or restricting agent activities. We are interested in domains where agents will generally need to operate with high degrees of autonomy. For this reason, we assume a permissive environment: unless stated otherwise, agents are allowed to operate independently of human interaction. In this context, guidance is used to limit the scope of activities that can be performed autonomously. Activities outside that set may still be performable, but require some form of interaction with the user. SPARK supports two types of guidance: *strategy preference* and *adjustable autonomy* (see [16] for details).

Guidance for strategy preference comprises recommendations on how an agent should accomplish tasks. These preferences could indicate specific procedures to employ or restrictions on procedures that should not be employed, as well as constraints on how plan variables can be instantiated. For example, the directive *"Use Expedia to find options for flying to Washington next week"* expresses a preference over approaches to finding flight information for planning a particular trip. On the other hand, the directive *"Don't schedule project-related meetings for Monday mornings"* restricts the choice for instantiating parameters that denote project meeting times.

Guidance for adjustable autonomy enables a supervisor to vary the degree to which agents can act without human intervention. This form of guidance can be used to force the agent to seek permission for executing designated procedures (e.g., *"Obtain permission before making any revisions to my schedule"*) and to require that the user be consulted for certain decisions such as instantiating variables or selecting among alternative procedures (e.g., *"Consult me when deciding how to respond to requests to cancel staff meetings"*).

Guidance can be asserted and retracted throughout the scope of an agent's operation, thus providing the means to tailor agent behavior dynamically in accord with a user's changing perspective on what he or she feels comfortable delegating to the agent. This comfort level may change as a result of the user acquiring more confidence in the agent's ability to perform certain tasks, or the need for the user to focus his problem-solving skills on more important matters.

Guidance is implemented through meta-level procedures that test properties associated with tasks, procedures, and actions. Two key properties used in specifying advice and guidance are *features* and *roles*. Features identify characteristics of procedures and actions (e.g., "COMMUNICATIVE"), whereas roles identify how particular parameters are used within an action (e.g., "RECIPIENT" parameter of a "SEND_EMAIL" action).

6. Balancing Formal and Practical Concerns

SPARK has been designed as a framework to support the development of real-world agent systems. That requirement has led to the inclusion of several capabilities within SPARK geared toward practical concerns. In addition, our experience with applications has made clear the need for a well-defined semantics for both an agent's knowledge and its execution model to enable various forms of reasoning about the agent and its capabilities. To date, we have concentrated on the practical aspects of the framework, as described in Section 6.1. Our work on the formal side has been limited to implementing the foundations to support the future reasoning capabilities described in Section 6.2.

6.1. Practical Features

The main practical concerns that have driven the development of SPARK are *ease of integration, scalability, failure handling,* and *developer support*.

SPARK's treatment of predicates and logical variables allows a logic-based interpretation of its task representation as required to support reasoning about procedures. However, SPARK does not use full unification as in other logic-based languages, but rather a restricted form of pattern matching. As a consequence, data values in SPARK are always fully instantiated, eliminating the need for variable dereferencing or explicit unbinding of variables. The benefits include a simpler implementation and easier integration of SPARK with external components written in procedural, functional, and object-oriented programming languages, without sacrificing logical semantics.

The language and execution models of SPARK were designed to allow diverse implementations covering different timescales. The current implementation of SPARK is as an interpreter written in Python. Python was chosen for its rapid prototyping capabilities, open source implementation, and ability to compile to the Java Virtual Machine via Jython (facilitating SPARK's integration with systems such as OAA [3] and SHAKEN [1]). By suitably restricting SPARK higher-level constructs, it is feasible to compile SPARK into efficient code in target languages such as C and C++ that could be used for applications requiring fast response times (e.g., robot control).

The semantic model of SPARK's `if:`, `try:`, and `wait:` constructs provides much finer control over failure handling and race conditions 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. 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.

For modularity, SPARK uses a hierarchical namespace for functions, predicates, actions, and constants. This simplifies both the parallel development of agent knowledge and the reuse of knowledge, both of which are critical for large-scale applications.

For developer support, SPARK has an IDE built on top of IBM's Eclipse Platform [17]. The IDE provides editing and debugging capabilities, with access to the internal state of execution of SPARK agents.

In the future, we plan to extend SPARK with a number of additional practically-motivated capabilities: a type system for static and dynamic type checking; compilation into a variety of target languages; the ability to have procedures persisting over very long times, with the ability to recover after machine crashes.

6.2. Formal Reasoning

Two main challenges motivate the need for formal reasoning within an agent system: *knowledge modeling* and *self-awareness*.

Knowledge modeling refers to the formulation of the activity knowledge that an agent uses to respond to stimuli (events, tasks) within its operating environment. Formulating procedural representations of activity knowledge, as required by SPARK, presents a significant modeling challenge. Current practice mostly relies on the handcrafting of such 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. First, *verification and validation* techniques can be applied to identify shortcomings in handcrafted knowledge. Second, *procedure synthesis* techniques can be used (in some cases) to generate composite procedures from descriptions of possible activities. Third, *learning* methods can be used to adapt existing procedural knowledge to handle unexpected situations that arise at runtime. Our future work will explore how to provide such reasoning services within SPARK.

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. The introspective and meta-level capabilities within SPARK provide a strong foundation upon which to build self-awareness functionality. Currently, we are developing three capabilities of this type. *Temporal projection* supports the ability to reason into the future based on a formal representation of an agent's objectives, commitments, and knowledge, to ensure that objectives can be met and undesirable consequences avoided; temporal projection is critical to ensuring that local decisions made by an agent are consistent with long-term objectives. *Explanation* of agent activity is necessary when agents operate on behalf of a human, and so must be able to justify their actions to him or her. *Recognition of knowledge limits* is essential to ensuring that an agent does not undertake activities that lie beyond the scope of its problem-solving abilities.

7. SPARK-based Personal Assistant

SPARK provides general-purpose agent technology for a range of domains that require reactive task execution. To date, the driving application for SPARK has been the development of an intelligent personal assistant for a high-level knowledge worker. This assistant, called CALO, 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 SPARK-based *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 SPARK's *guidance* mechanisms (see Section 5.1).

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

8. Comparison to Other Agent Frameworks

At the theoretical end of the spectrum of agent languages there are logic-based agent languages, such as AgentSpeak(L) [18], Golog/ConGolog [8], and 3APL [9], that were designed primarily to have well-defined formal properties. These do not scale well for use in building large applications that need to integrate with and control other systems.

At the other extreme lie commercial agent development systems such as JACK [2]. 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 sytems. However, since JACK is an extension of Java, if you want to reason about and synthesize JACK agents' plans, you need to be able to reason about and construct Java programs – not something that Java was designed for.

SPARK falls between these extremes. The finite state machine model of SPARK execution is much more amenable to formal analysis than the extended Java programs of JACK. An additional advantage is that SPARK allows new procedures to be constructed on-the-fly at runtime as a result of reasoning about new situations. Compared with the logic-based agent languages, SPARK's treatment of variables and the FSM model of execution make SPARK a closer fit for integration with external components and allow for simple and efficient implementations.

SPARK has been heavily influenced by PRS, particularly with respect to the design of the meta level. Compared with PRS, SPARK has a more precisely defined semantic model and a greater emphasis on engineering issues.

The robotics community has produced a number of agent systems that serve as task-level controllers for managing the activities of mobile robots (e.g., PRS-lite [13] and RAP [6]). As with SPARK, these sytems act as embedded controllers in a dynamic world, applying predefined libraries of procedural knowledge to perform tasks and respond to unexpected events. Like SPARK, these procedural languages provide great flexibility and expressiveness in order to handle the complexities inherent to activity in the real world. The nature of the robotics applications, however, has resulted in some significant technical differences from SPARK. Responsiveness is paramount for robotics applications, resulting in architectural designs that try keep the execution cycle time on the order of tens of milliseconds. For this reason, the type of reflective reasoning provided by SPARK is not supported. Furthermore, program variables of different types are often used to characterize world state rather than the more general declarative knowledge base used within SPARK. Finally, little attention has been paid by the robotics community to issues related to reasoning about procedures, with the result that the semantics of the richer representation languages are not well defined.

9. Conclusions

To tackle many real-world applications, agent systems need both a strong formal model and a practical implementation that scales well and integrates with other components. Motivated by these two needs, SPARK has taken the middle ground between theoretical agent formalisms and large-scale agent implementation systems, by providing an expressive and practical agent development system backed by a logical semantic model.

SPARK is currently being used as the process representation and execution mechanism in CALO, a large agent system that integrates a variety of AI technologies. To date, the development of SPARK has concentrated on the practical aspects of the formalism required to support CALO. In the future, we plan to apply the formal properties of SPARK to the problem of reasoning about agents and their capabilities.

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 wish to 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

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

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

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

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

[5] R. J. Firby. *Adaptive execution in complex dynamic worlds*. PhD thesis, Yale University CS Dept., 1989. Technical Report RR-672.

[6] R. J. Firby. Task networks for controlling continuous processes. In *Proc. of the Second Int. Conf. on AI Planning Systems*, Menlo Park, CA, 1994. AAAI Press.

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

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

[9] K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Meyer. 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*, pages 215–229. Springer-Verlag, 1998.

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

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

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

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

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

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

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

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

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

[19] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proc. of the First Int. Conf. on Multiagent Systems*, San Francisco, 1995.

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