Introduction to SPARK

version 0.3

David Morley (morley@ai.sri.com)

July 13, 2004

Abstract

SPARK is a new agent framework, being developed at the Artificial Intelligence Center of SRI International. Its design has been strongly influenced by its predecessor, PRS, and is based on the same Belief Desire Intention (BDI) model of rationality. The motivations for the development of SPARK include: support for development of large-scale agent applications, principled representation of procedures that will enable validation and automated synthesis, flexibility in the delivery platform (including the potential to run on PDAs and mobile platforms), and built-in support for user advisability of agents.

1 Introduction

This document provides a brief introduction to SPARK. SPARK is a new agent framework, being developed at the Artificial Intelligence Center of SRI International. Its design has been strongly influenced by its predecessor, the Procedural Reasoning System (PRS)[5].

PRS was one of the earliest agent-based frameworks, providing a flexible plan execution mechanism capable of both goal-directed activity and reacting to changes in its execution environment. The influence of PRS is evidenced by the large family of successors that extend or modify the original Lisp-based PRS [9, 7, 2, 3, 6, 8, 4].

The PRS family of agent-based languages allow the development of active systems that interact with a constantly changing an unpredictable world. The problem is divided among a set of *agents*, each of which maintains its own *knowledge base* of beliefs about the world. The agents continually update their knowledge bases in response to sensory information and reasoning about the state of the world and can perform actions the change things in the world. At any time, each agent has a set of *tasks* that it is trying to achieve. These tasks are either initially given to the agent or are introduced in response to perceived events or the internal working of the agent. Some tasks can be achieved by performing primitive actions. Others are achieved by breaking down the task into simpler tasks using hierarchical decomposition. To do this, the agent has a library of *procedures*

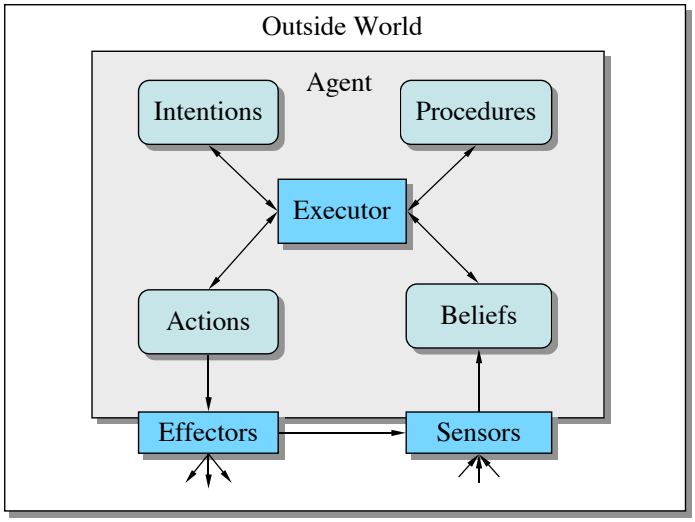

Figure 1: A SPARK Agent

that describe possible ways of breaking down the task. To break down the task, the agent chooses between the possible procedures and creates an *intention* to execute that procedure[1].

SPARK is new member of the PRS family that addresses some key limitations of earlier PRS systems. SPARK is more careful about handling delays that can cause race conditions, and has built in support for user advisability of agent. SPARK is designed so that in future it can be compiled down to efficient code with a small footprint that can fit on a PDA, yet so that it can also scale to large projects. By being based on Python and/or Java, SPARK avoids the current licensing and portability constraints of Allegro Common Lisp.

Section 2 describes the basic concepts of SPARK. These concepts are further expanded in the sections that follow. Section 8 describes planned enhancements to SPARK and section 9 compares SPARK with PRS.

2 Basic Concepts

Each SPARK interpreter process contains one or more agents. Each agent is embedded in the world and interacts with the world though sensors and effectors. Each agent has its own knowledge bases for beliefs and procedures. The knowledge bases are initially loaded from files written in the SPARK source language, SPARK-L. The knowledge base is then updated by the agent's sensors and through the agent executing procedures. The set of procedures that the agent is currently executing are called the intentions of the agent. At any time an agent may be executing multiple

[1]Intentions are both (i) a means of reducing the computational complexity faced by an agent by at least temporarily committing to one procedure and not constantly reconsidering possible ways of performing each task and (ii) a commitment to a certain behavior that helps coordination when dealing with other agents or human users.

intentions.

At SPARK's core is the executor whose role is to manage the execution of intentions. The executor of an agent repeatedly selects one of the current intentions to process and performs a single step of that intention. This generally involves performing tests on the agent's beliefs and, based on the result, immediately starting to execute some action. Primitive actions cause effects through the effectors. Non-primitive actions are expanded by the executor according to the procedures the agent has available. On completion of actions, the beliefs may be updated, which can in turn trigger the creation of new intentions based on the available procedures.

In general, procedures have local variables to record information. In executing a step of a procedure, the executor commonly tests the agent's beliefs and then starts to execute some action based on those tests. The testing of beliefs is expressed as finding values for local variables that satisfy some predicate expression. There may be many possible solutions, however the executor commits to one solution by binding the local variable appropriately, and then proceeds to execute the relevant action.

2.1 Modules

A SPARK agent's knowledge is expressed in terms of data values, functions over these data values, predicates expressing relationships between data values, and tasks that specify activities to be performed. Tasks include activities such as performing some action, trying to make some condition true, and combinations of simpler tasks.

The agent's initial knowledge comes from source files written in the SPARK-L language, where these files are grouped into hierarchically arranged *modules*

In SPARK-L there are many cases where there is a need to reference hierarchically arranged objects. These include files, modules, and the functions, predicates, actions declared within modules. To refer to such objects within SPARK-L we use the *path* syntactic construct. A path is written using "." to separate the hierarchy levels, for example, `foo.bar.aux` is a three-level path that we could use to name a module.

Every spark source file is *in* a particular module – this is the module that all declarations in the file are added into. For convenience, the *relative path* syntactic construct allows a path to be written relative to the source file's module. A relative path starts with one or more "."s. Within a file in module `a.b.c` , the relative path name `.` maps to `a.b.c` , `.d` maps to `a.b.c.d` , `.d.e` maps to `a.b.c.d.e`, and so on. Each additional "." prefix goes up a level in the hierarchy, thus `..` maps to `a.b` , `...` maps to `a` , and so on.

The SPARK-L source files for a module specify:

- *declarations* of named entities such as functions, predicates, and actions. These declarations specify information about the entity such as how many arguments it takes, and so on.
- *definitions* of how those entities are implemented – for example that `+` is defined by a Python

```
importfrom: message IsSpam subjectOf sendTo
importfrom: subject
importfrom: person
export: forwardMessage InterestedIn

{defpredicate (InterestedIn $person $subject)}

{defaction (forwardMessage $message)}

{defprocedure forwardMessageUnlessSpam
 cue: [do: (forwardMessage $message)]
 precondition: (not (IsSpam $message))
  body:
 [forall: [$person] (InterestedIn $person (subjectOf $message))
                    [do: (sendTo $person $message)]]
}

{defprocedure reportSpam
 cue: [do: (forwardMessage $message)]
 precondition: (IsSpam $message)
 body: [do: (sendTo SpamCollector $message)]
}

(InterestedIn Bill implementation)
(InterestedIn Bill documentation)
(InterestedIn Bob implementation)
```

Figure 2: A module written in SPARK-L

function or that `located` is defined by a set of knowledge base facts, and so on

- a collection of *facts* that form the agent's initial beliefs
- a collection of *procedures* that form the agents initial procedure library
- a set of *advice* on how to select between procedures

2.2 Example File

Figure 2 shows a simple file written in SPARK-L, the language provided by SPARK for writing modules. This file imports entities with identifiers `IsSpam`, `subjectOf`, and `sendTo` from the `message` module, and imports all entities declared in the `subject` and `person` modules (which presumably include `implementation`, `documentation`, `Bill`, `Bob`, and `SpamCollector`).

The file *declares* four identifiers: `InterestedIn` names a predicate taking two arguments, `forwardMessage` names an action taking one argument, and `forwardMessageUnlessSpam` and `reportSpam` name pro-

cedures. The `export:` statement allows other files to import declarations of `InterestedIn` and `forwardMessage`.

The *definition* of `InterestedIn` and `forwardMessage`, that is, how they are implemented, is not specified explicitly in this file. Because of this, they are taken to use the appropriate default implementation. The default implementations are:

- Functions are functors – simple record-like constructors/deconstructors - as in Prolog
- Predicates are extensional – defined as a set of facts stored in the knowledge base
- Actions are hierarchical – a set of procedures must be written to specify alternative ways of decomposing the action.
- Constants are symbols – values that represent paths.

The example file includes five pieces of information for the knowledge bases: two procedures that represent possible ways of performing a `forwardMessage` action and three facts about people's interests.

In this file, there is no ambiguity in the use of identifiers. However, it is not hard to imagine cases where entities with the same indentifier are imported from different modules, or the the file declares an entity with the same identifier as one that is imported. Whenever this occurs, the simple identifier cannot be used to reference the entity. Instead, it is necessary to use a path that specifies the module that the identifier is from. For example, `message.IsSpam` (the `IsSpam` from the `message` module) or `.forwardMessage` (the `forwardMessage` from the current module).

In the example file, we can see examples of the four main kinds of syntactic expressions:

- Term expressions (*TERM*) – expressions that (given some variable bindings) denote a specific data value. These include `Bill`, `(subjectOf $message)`, and `implementation`.
- Predicate expressions (*PRED*) – expressions that denote a relationship between data values. These include `(InterestedIn $person $subject)` and `(not (IsSpam $message))`.
- Task network expressions (*TASK*) – expressions that specify actions to be performed and conditions to be achieved. For example, `[do: (sendTo SpamCollector $message)]`.
- Statements (*STATEMENT*) – top level forms such as `{defaction (forwardMessage $message)}`.

2.3 Syntactic Structure

At this point it may be helpful to describe the general structure of SPARK-L *Term*s. The atomic syntactic elements of SPARK-L are:

INTEGER an integer literal such as `1`

FLOAT a floating point literals such as `1.0`

STRING a string literals such as `"Hi!"`. A *PATH*

PATH a path such as `Bill` or `person.Bill`. A *PATH* consists of an "id", either an alphanumeric id (a sequence of letters, digits, and underscores, not starting with a digit), or a special id (a sequence of characters in `-+*/%^&|!?<>=`), with an optional prefix (a sequence of alphanumeric ids separated by periods). Often a *PATH* is interpreted as referring to some declared entity (a function, predicate, action, etc.) through consideration of the local and imported declarations. In this case we will refer to the *PATH* as an *ENTITY*.

TAG a path that ends with ":" such as `importfrom:` or `seq:`

VAR a variable such as `$x` consists of an alphanumeric id prefixed by one or more dollar signs.

Compound syntactic structures are constructed using parentheses `()`, braces `{}`, and brackets `[]`. Brackets are used for collections. Parentheses and braces enclose an *ENTITY* followed by parameters. Braces are used for special syntactic constructs that allow optional keyword parameters. Braces always introduce a new scope for variables.

In describing compound syntactic structures, we use *(`X`)** to represent zero or more occurrences of `X` and *(`X`)?* to represent zero or one occurrence of `X`.

3 Values and Term Expressions

SPARK data types include the primitive types integer, float, and string, and compound types such as records, lists, and closures of various kinds.

The syntactic structures of the SPARK language that describe data values are *TERM*s. These include:

- An *INTEGER* evaluates to the specified integer, e.g. `1`.
- A *FLOAT* evaluates to the specified floating point number, e.g., `1.34`.
- A *STRING* evaluates to the specified string, e.g., `"string"`.
- A *VAR* evaluates to the value of the variable, e.g., `$x`.
- An *ENTITY* that names a constant evaluates to the value of that constant
- `(`*ENTITY(TERM)**`)` – where the initial *ENTITY* names a function – evaluates to the result of applying the function to the values of the parameters, e.g., `(add 1 $x)`. By convention, function names begin with a lowercase letter.
- `[`*(TERM)**`]` evaluates to a list of the given values, e.g., `[1 2 3]`

Other *TERM*s include closures, which are described in Section **??**.

Functions whose result does not depend upon the state of the knowledge base are called *static* functions. The result of *dynamic* functions (i.e., fluents) depends deterministically upon the state of the knowledge base. That is, for any given state of the knowledge base (or any given point in time) two calls of the same function with the same arguments return the same value, but for different states of the knowledge base, different results may be returned. For convenience we also allow *non-deterministic* functions whose results may vary each time they are called[2].

When all the free variables in a term expression are bound, the term expression can be evaluated to produce a value. In certain circumstances, term expressions containing unbound variables can be matched to values to bind those variables. For example, the term expression `[$x [$y $z]]` can be matched to the value `[1 [2 3]]`, bindings `$x` to 1, `$y` to 2, and `$z` to 3.

Data values in SPARK are governed by a strict principle: *the equality of two values is never allowed to change* – if any two values are "equal" at one point in time, then they must always be equal. This restriction affects the handling of "compound" values, such as lists and records:

- In SPARK, two lists are considered equal if they contain the same sequence of elements. Since the equality of two lists is not allowed to change, the elements of those two lists are not allowed to change. Thus lists are immutable data structures in SPARK and destructive modification of lists is not allowed.
- Similarly, records constructed in SPARK are considered equal if they have the same functor and the same sequence of argument values. Thus records are immutable data structures in SPARK.
- It is possible to create "mutable" objects that can be used as values, such as queues. However, since the equality of two values is not changeable, two "mutable object" values are equal if and only if they refer to the same object. Thus two different queues may contain the same elements, but they will never be equal. Not only that – any update to a queue is considered a change in the agent's knowledge base and is only allowed at certain restricted times.

4 Predicate Expressions

The syntactic structures of the SPARK language that describe relationships between data values are *predicate expressions*, *PRED*s. These include:

- `(`*ENTITY (TERM)**`)` – where the initial *ENTITY* names a predicate – represents the application of that predicate to the given values, e.g., `(P 1)`. By convention, predicate names begin with an uppercase letter.
- `(and` *(PRED)**`)` is the conjunctive logical connective, e.g., `(and (P $x) (Q) (R))`.

[2]Although the semantic interpretation of non-deterministic functions is questionable.

- `(or` *(PRED)**`)` is the disjunctive logical connective, e.g., `(not (P $y))`.
- `(not` *PRED*`)` is logical negation, e.g., `(not (P 1))`.
- `(exists` *VARLIST PRED*`)` – where *VARLIST* is `[`*(VAR)**`]` – is the existential quantifier, e.g., `(exists [$x $y] (P $x $y))`

As with functions, we have *static*, *dynamic*, and *non-deterministic* predicates. Predicates whose solutions do not depend upon the state of the knowledge base are called *static* functions. A predicate fluent whose solutions depends deterministically upon the state of the knowledge base is called *dynamic*. A predicate whose solutions may be different each time it is tested, even in the same knowledge base state, is called *non-deterministic*[3].

As in logic programming languages, all occurrence of a variable have the same value. When testing a predicate expression there may be multiple alternative sets of bindings for the variables that make the predicate expression true. However, once the executor starts to execute some action, it must select and commit to one set of bindings for the variables.

In SPARK, each term expression that is used as an argument to a predicate either (i) represents a unique data value if all the variables in it have previously been bound, or (ii) represents a pattern to match against data values that the predicate generates, thereby binding any variables in the term expression that have not previously been bound. In either case, after successfully testing a predicate, all the (free) variables that appeared in the term expression arguments passed to the predicate must be bound to data values. For example, if `$x` is already bound, and we wish to test the predicate expression, `(P $x $y (f $x $z))`, the first argument can be evaluated to give a data value, but the second and third arguments will be treated as patterns to match against possible values. Once the predicate expression is satisfied, all three variables that appear in the expression will be bound to data values.

A consequence of the requirement that all variables be bound after successfully testing a predicate expression means that `(= $x $y)` is only allowed if at least one of `$x` or `$y` is already bound – otherwise it would be forced to enumerate all possible data values as bindings for `$x` and `$y`! This is unlike Prolog, where `X=Y` would succeed with variables `X` and `Y` *unified*, but not bound to any specific value[4].

When dealing with compound predicate expressions, this modality of the variables (i.e., whether they are bound yet or not) is very important:

- In a conjunction, the variables are bound from left to right, so `(and (= $x $y) (= $x 1))` would cause an error (unless either `$x` or `$y` were already bound), whereas the predicate expression `(and (= $x 1) (= $x $y))` would not result in an error (although it may fail if either `$x` or `$y` is already bound to something other than 1).
- In a disjunction, if any variable, say `$y`, appears in one disjunct but not another, and is not already bound prior to the disjunction, then after the disjunction succeeds the variable may

[3] With dubious semantics as in the case of non-deterministic functions.

[4] Avoiding true unification and instead using pattern matching makes the implementation of SPARK much simpler.

or may not be bound. However, SPARK requires the modality of all the variables used in any expression to be fixed before testing that expression. If `$y` were to appear in a later term expression, it may not be possible to know whether that term expression should be treated as a data value or as a pattern to match a data value. For this reason, SPARK does not allow a variable that is bound in one disjunct but not another to appear in any later expression.

- SPARK currently applies the Closed World Assumption to predicates and uses negation-as-failure [1] to test negated predicate expressions, that is, `(not (P $x))` is equated with the failure to prove `(P $x)`[5]. If a binding is found for `$x` that satisfies `(P $x)`, then the negated expression fails. If no binding is found, the negated expression succeeds, but does not bind any variables.

  SPARK enforces the restriction that in a negated predicate expression, all free variables must be bound prior to testing the negation. If it did not then negation-as-failure would return incorrect results.

  For example, consider the predicate expression `(and (= $x 2) (not (= $x 1)))`. Testing this leads to the expected result of `$x` being bound to 2 and the negation and conjunction succeeding. Consider what would happen if we were to attempt to test the predicate expression `(and (not (= $x 1)) (= $x 2))` where we have swapped to conjuncts. If `$x` were previously unbound, the test of `(= $x 1)` within the negation would succeed with `$x` being bound to 1, causing both the negated expression and the conjunction to fail. SPARK therefore disallows this second form of the conjunction from being used where `$x` is not previously bound. If you indeed want this second behavior, you can get it, but you must explicitly include the existential quantifier that negation-as-failure is implicitly inserting: `(and (not (exists $x (= $x 1))) (= $x 2))`

5 Task Network Expressions

In SPARK, a procedure includes a network of tasks to execute. This network of tasks is expressed as a *task network expression*, *TASK*.

A task network expression may represent a *basic task*, commonly to perform some action or achieve some state, or a it may represent a *compound task network* that is a combination of simpler task network expressions. Each task network expression can be annotated with *modifiers* that annotate expression.

Syntactically, each task network expression is of the form `[`*(TASKTAG)? (MODTAG)**`]`, where *TASKTAG* and *MODTAG* consist of a *TAG* followed by parameters. If the *TASKTAG* is left out, it defaults to the `succeed:` basic task which immediately succeeds without doing anything. The *MODTAG*s are modifiers.

For example, the task

[5] This will be generalized later to include predicates whose value is unknown. This will enable alternative implementations of negation and make it possible to have procedures that actively test the value of an unknown predicate.

```
[do: (forwardMessage $m)
 label: response]
```

is a basic task network expression that calls for the action `(forwardMessage $m)` to be performed. A label `response` is associated with this task.

5.1 Basic Tasks

The bottom-level tasks in any task network expression are basic tasks. The two main basic tasks are to perform some action, and to achieve the truth of some predicate. Basic tasks *TASKTAG*s include:

- `do:` *ACT* – where *ACT* is `(`*ENTITY (TERM)**`)` and *ENTITY* names an action
  e.g., `[do: (paint $house red)]`. This basic task attempts to perform the specified action. Actions can either be primitive or non-primitive.
  - Primitive actions are performed by executing some arbitrary Python or Java code.
  - Non-primitive actions are performed by expanding the action using procedures in the agent's procedure library. SPARK selects a procedure matching the action for which the procedure precondition is satisfied. SPARK then executes the body of that procedure. This may involve executing primitive actions and other non-primitive actions, which also need to be expanded.
- `achieve:` *PRED*
  e.g., `[achieve: (Color $house red)]` This basic task attempts to make the specified predicate true. If the predicate is already true, then the achieve task succeeds immediately. Otherwise SPARK selects a procedure matching the achieve task and executes the body of that procedure.
- `succeed:`
  Always succeeds.
- `fail:` *TERM*
  e.g., `[fail: (resource_failure "insufficient paint")]` Always fails. The single argument expresses the reason for failure[6].
- `context:` *PRED*
  e.g., `context: (HasBoss $employee $boss)`

  The context predicate expression is tested. If there is a solution to that predicate expression, the first generated set of variable bindings is kept. If there is no solution, the task network expression fails.
- `conclude:` *PRED*
  e.g., `conclude: (P 1)`

  The given fact is added to the knowledge base (if it is not present). All variables here must be bound.

[6] Once the type system is in place, the argument will need to be an instance of the Failure type.

- `retractall:` *VARLIST PRED*
  e.g., `retractall: [$x] (P $x)`

  This removes all facts for which some binding of the given variables cause it to match the given fact pattern.

- `retract:` *PRED*
  e.g., `retract: (P 1)`

  This removes the given fact from the knowledge base (if it was present). All variables here must be bound. It is equivalent to `retractall: []` *PRED*

Basic tasks should not use dynamic predicates in their parameters. The value of dynamic functions depends upon the time that the functions are evaluated. Given that the state of the knowledge bases may change over the execution of a basic action, the value of any dynamic functions used in parameters to a basic action may change over the course of execution of that action. Therefore it is recommended that to avoid unpredictable results, only static functions should be used in parameters to basic actions that may take an extended time. This recommendation is neither tested nor enforced by SPARK.

5.2 Compound Task Network Expressions

Compound task network expressions combine simpler task network expressions in different ways:

- Tasks can be executed in parallel or sequentially.
- Conditional execution of tasks is allowed, based on either the truth of predicate expressions or the successful execution of other tasks.
- Tasks can be iterated. The iteration can be based on the set of solutions for some predicate expression, based on the truth of some dynamic predicate expression, or explicitly terminated within the loop body. Variables that are bound only once across multiple iterations are not very useful, instead the iteration task network expressions allow variables that are local to each individual iteration. Each time through the loop, there is a fresh set of loop variables[7].

Compound task network *TASKTAG*s include:

- `parallel:` *(TASK)**
  e.g., `[parallel: [do: (walk)] [do: (chewGum)]]`

  Executes the subtask networks in parallel. The subtask networks are not (necessarily) executed in left to right order. SPARK enforces the restriction that if a variable in the `parallel:` task network expression was not bound prior to execution of the task network, then that variable must not appear in more than one subtask network [8]. If the execution of a subtask

[7] Any information required to be kept between iterations can be stored in the knowledge bases.

[8] This is because we cannot know which of these subtask networks should bind the variable and which should use the value.

network fails, then all the other subtask networks that are still running are interrupted and execution of the parallel task network fails.

- `seq:` *(TASK)**
  e.g., `[seq: [do: (paint roof red)] [do: (paint walls brown)]]`

  Executes the given subtask networks in the specified sequence. If any subtask network fails, the sequence task network fails (with the same reason) without executing subsequent subtask networks. The knowledge base does not change between starting to execute the `seq:` and starting the first subtask network, but changes may occur between the subtask networks.

- `select:` *(PRED TASK)**
  e.g., `[select: (Tired) [do: (sleep)] (Hungry) [do: (eat)]]`

  The arguments to the `select:` compound task network expression are alternating *PRED*s and *TASK*s. Each pair represents a possible alternative execution path. Execution of the `select:` task network expressions requires testing the *PRED*s *in order*, finding the first *PRED* that has a solution, choosing a variable binding corresponding to one solution, and then executing the corresponding *TASK*. If no *PRED* evaluates to true, then the `select:` task network expression fails immediately, otherwise its success or failure is the same as the *TASK* that it selected to be executed.

  The knowledge base does not change between starting to execute the `select:` and starting the selected subtask network. Thus both the chosen *PRED* and any conditions that held at the start of the `select:` hold at the start of execution of the chosen subtask network.

  In the example above: if the agent is tired and not hungry, it will sleep; if it is hungry and not tired it will eat; if it is both hungry and tired it will sleep and not eat; if it is neither hungry nor tired, the `select:` task network will fail.

  The treatment of variables in `select:` task network expressions is analogous to the treatment of variables in `or` predicate expressions – any previously unbound variable that does not appear in every alternative (*PRED-TASK* pair) cannot be used after the `select:` task network expression.

- `wait:` *(PRED TASK)**
  e.g., `[wait: (Tired) [do: (sleep)] (Hungry) [do: (eat)]]`

  This task network expression is very similar to `select:`, except that instead of failing if none of the *PRED*s is true, it waits until one is true. In the example, if the agent is neither tired nor hungry, it would wait until it was either tired or hungry.

  If the predicate expression of one of the alternatives is true immediately after execution of the `wait:` task network starts, then the appropriate subtask network starts execution immediately. If not, a subtask network will start at some later time at which the predicate expression is true. Because it may be necessary to wait for a predicate expression to become true, any condition holding at the start of the `wait:` may no longer be true. However, is is guaranteed that the *PRED* for the selected *TASK* will be true at the start of executing the *TASK*. This means that if one of the predicate expressions becomes true, then becomes false again before the corresponding subtask network has a chance to start, the subtask network will *not* start. Instead the `wait:` task network waits again until one of the predicate expressions becomes true.

- `try:` *(TASK TASK)**
  e.g., `[try: [do: (lift $block)] [conclude: (Succeeded)] [] [do: (panic)]]`

  The `try:` task network expression is in some ways similar to the `select:` task network expression, but instead of selecting an alternative based on the truth of a predicate expression, the alternative is selected based on the success of executing a task network expression: In the example, if the task network expression `[do: (lift $block)]` succeeds then the task network expression `[conclude: (Succeeded)]` is executed. If not, the task network expression `[]` is executed immediately. If this succeeds (which it always does) then the task network expression `[do: (panic)]` is executed.

  SPARK treats the task network expressions as coming in pairs. The first task network expression of the first pair is executed. If it succeeds, then the second task network expression in that pair is immediately executed and the success or failure of the `try:` task network expression is based on the success or failure of that second task network expression. If it fails, then the first task network expression of the next pair is immediately executed. If that succeeds, the second task network expression of that pair is immediately executed and so on.

  If none of the executed task network expressions succeeds, then the `try:` task network expression fails with the same reason as the last of the executed task network expressions.

  The treatment of variables in `try:` task network expressions is similar to the treatment of variables in `or` predicate expressions: any previously unbound variable that does not appear in every alternative (*TASK-TASK* pair) cannot be used after the `try:` task network expression.

There are also iterative compound task network expressions:

- `while:` *VARLIST PRED TASK*
  e.g., `[while: [] (Hungry) [do: (eat pancake)]]`

  A while task network tests a predicate expression and if it is true attempts to execute the given subtask network, it then repeats the test and subtask network execution until the predicate expression is false. If the predicate expression tests false the repeat task network succeeds. If the subtask network fails for some reason, the while task network fails with the same reason.

- `forall:` *VARLIST PRED TASK*
  e.g., `[forall: [$x] (Wall $x) [do: (paint $x blue)]]`

  A forall task network finds all solutions for the given predicate expression and then executes the subtask network for each solution. The list of variables given are local to the predicate expression and the subtask network and are not visible outside the forall task network expression. All other variables in the predicate expression and subtask network should be bound before executing the forall task network.

  If all the *taskexpr* executions succeed then the forall task network expression succeeds. In the degenerate case of no solutions, the forall task network expression will always succeed. If one of the subtask network expressions fails, the forall task network expression fails without executing any further task network expressions.

5.3 Exceptions: Failures and Errors

It is possible for task (and the task network expressions that contain them) not to complete successfully. This occurs when a task raises an *exception*. There are two kinds of exceptions: *failures* and *errors*:

- Failures occur when context conditions are tested and found to be false, when there are no procedures applicable for a task being executed, when a procedure explicitly signals failure by executing a `fail:` task, and so on. Failures are expected to occur during the execution of well-written code.
- Errors are exceptions that should not occur in well-written code and correspond to programming errors, such as division by zero, attempting to execute a task with unbound variables in parameters that disallow them, and so on.

When execution of a task network expression fails (or more generally, raises an exception) rather than succeeding, the flow of control does not proceed as it would if the task network expression had succeeded. Instead the failure is propagated up the to a point where the failure is handled by a `try:` task network expression – possibly killing of the entire intention if it is not handled anywhere. Thus if execution of a task network expression fails, the enclosing task network expression will fail, and so on, stopped only by a task network expression that explicitly handles the failure. Any variable bindings that a task network expression may have made are undone when that task network expression fails.

5.4 Variables in Task Network Expressions

In general, the execution of a task network, consists of executing some sequence of basic tasks, where the order and timing of those basic tasks is directed by the compound task networks. As preconditions are tested and basic tasks executed, additional variable bindings are made. Once the executor has committed to a variable binding, that variable binding will remain until either:

- a task network expression in which the variable is bound raises an exception, or
- inside an iterative construct, the loop body that bound the variable finishes (variables that are bound in a loop body are local to the loop body and are considered fresh variables on each iteration).

5.5 Modifiers

The following modifiers can be used to annotate the task network expression:

- `label:` *PATH*

This modifier attaches a label to the task network expression for referencing it later.

- `comment:` *STRING*

  This modifier provides documentation for the task network expression.

6 Closures

Closures are data values that represent "executable" objects constructed from expressions – functions from term expressions, predicates from predicate expressions, and actions from task network expressions. Closures make it possible to create functions, predicates, and tasks that operate on other functions, predicates and tasks.

Function closures are data values that represent functions. The closure `{fun [$x] (- $x 1)}` is analogous to the lambda expression `#'(lambda (x) (- x 1))` in Lisp or `lambda x: x - 1` in Python. The general form of a function closure is {`fun` *VARLIST TERM*}. You apply a function closure to arguments using the `applyfun` function, thus `(applyfun {fun [$x] (- $x 1)} 9)` would evaluate to `8`.

Predicate closures are data values that represent predicates. Predicate closures are of the form {`pred` *VARLIST PRED*}. For example `{pred [$x $y $z] (and (= $x $y) (= $y $z))}`. You apply a predicate closure to arguments using the `applypred` predicate, thus `(applypred {pred [$x $y $z] (and (= $x $y) (= $y $z))} 1 1 $a)` would succeed with `$a= 1`.

Task network closures are data values that encapsulate a task network expression to produce an action. Task network closures are of the form {`task` *VARLIST TASK*}. For example `{task [$x] [wait: (P $x) [retract: (P $x)]]}`. You apply a task network closure to arguments using the `applyact:` action, thus
`[do: (applyact {task [$x] [wait: (P $x) [retract: (P $x)]]} 7)]` would wait until `(P 7)` is true and then retract `(P 7)`.

The braces introduce a new scope for variables. Any variable such as `$x` that appears within the braces is distinct from a variable `$x` outside the braces. The value of the outer `$x` can still be accessed from within the braces, however it must be accessed as the variable `$$x`. The extra dollar sign indicates that the variable comes from the immediately enclosing scope not the current scope. Thus, `{fun [$x] (- $x $$y)}` is the SPARK equivalent of `#'(lambda (x) (- x y))` in Lisp. Additional dollar signs can be added to specify scopes further out. For example, `$$$x` would refer to the variable `$x` two scopes out. For example, the following obtuse predicate expression evaluates has two solutions `$x`=1, `$y`=4 and `$x`=2, `$y`=8:
`(and (Member $x [1 2]) (= $y (call {fun [$z] (* $z $$x)} 4)))`

7 File Contents

SPARK-L files are located by means of a *logical file system* that maps file name paths to physical files. SPARK uses an environment variable that lists a number of directories. To find the file corresponding to path `a.b.c`, SPARK looks for file `a/b/c.spark` in each of these directories in order, and selects the first one.

The SPARK-L files contains

- statements used to manage modules and the entity declarations that are visible within the file,
- statements that declare entities
- facts to be added to an agent's knowledge base

7.1 Namespaces

Within a SPARK-L source file, entities such as functions and predicates are named by paths. The *namespace* of a file is the mapping from paths to entity declarations that is effective for that file. An entity declaration is *visible* in a file if it is in the namespace of the file.

SPARK-L files are grouped into *modules*. Modules are named by paths and are loaded by loading the file with the same name, the *root* file. Thus to load module `a.b.c`, SPARK would load the file with path name `a.b.c` which would map to a physical file `a/b/c.spark`. If the module contains more than one file, the root file must load in the other files using statements of the form `include:` *PATH* where the *PATH* names a file. Each of those files must include a statement of the form `module:` *PATH*, e.g., `module: a.b.c`. The `module:` statement is optional for the root file.

The root file of a module can make declarations visible in that file available to any other file using `export:` statements. An export statement is of the form `export:` *(ENTITY)**. For example, in file `d.e.f` the statement `export: one two three` makes the declarations of `one`, `two`, and `three` available for other files to use. Another file can add those entity declarations to its own namespace using a statement of the form `importfrom:` *PATH (PATH)**. For example, in a file `a.b.c`, the statement `importfrom: d.e.f one two` adds to the namespace of file `a.b.c` the declarations of entities `one` and `two` exported from the module `d.e.f`. The degenerate form of the `importfrom:` statement that leaves out the specific entities to import. This form imports *all* exported entity declarations. Thus `importfrom: d.e.f` would import all three entity declarations.

In general, within a SPARK file the visible entities are named simply by their id, that is a *PATH* with no prefix, for example `one`. However, it is possible to import an entity with the same id as one that is declared in the module or imported from another module. In this case the use of the id alone is ambiguous and instead, a prefix must be used to resolve the ambiguity, e.g., `a.b.c.one` or `d.e.f.one`.

7.2 Declarations

7.2.1 Procedures

Procedures are defined using "defprocedure" statements. The syntax of procedure definitions is as follows:

{`defprocedure` *ENTITY*
`cue:` *CUE*
`precondition:` *PRED*
`body:` *TASK*
(`doc:` *STRING)?*
(`features:` *TERMLIST)?*
(`roles:` *TERMLIST)?*}

Where *TERMLIST* is a list of *TERM*s, i.e., `[`*(TERM)**`]`.

Each procedure has a *CUE* that specifies when to invoke the procedure which is of the following form [9]:

- `[newfact:` *PRED*`]` – instances of this procedures should be invoked (in new intentions) if a fact of the form *PRED* is added to the knowledge base.
- `[do:` *ACT*`]` – this procedure shows one possible way of decomposing the action *ACT*.
- `[achieve:` *PRED*`]` – this procedure shows one possible way of making *PRED* become true if it is not already true.

The body of the procedure is a task network expression that is to be executed. A procedure can optionally specify a documentation string that describes the procedure. You can also associate features and roles with the procedure that provide information about the procedure and its variables to the advice mechanism to help with the selection of which procedures to use.

7.2.2 Predicate Declarations

A predicate declaration is of the form

{`defpredicate (`*ENTITY (VAR)**`)`
(`imp:` *TERM)?*
(`determined:` *TERMLIST)?*
(`doc:` *STRING)?*
(`properties:` *TERMLIST)?*}

The `doc:` argument provides a documentation string for the predicate.

[9] Other cue types will be added in the future.

The `properties` argument provides a way of associating arbitrary information with the predicate.

The `determined:` argument specifies argument modes (whether the argument is a valid data value or a pattern to match data values) for which the predicate returns a unique solution. This specification is a list of strings, where each string is a sequence of "+"s and "-"s – each character corresponds to an argument. "+" indicates that the argument must be a data value. "-" indicates that the argument is allowed to be a pattern. For example the determinism of the equality predicate could be defined as follows:

```
{defpredicate (= $arg1 $arg2) determined: ["+-" "-+"]}
```

This indicates that if `$arg1` is a data value (`"+-"`) or if `$arg2` is a data value (`"-+"`) then there is at most one solution to `(= $arg1 $arg2)`.

Note that for any predicate, if all the arguments are data values rather than patterns, then the predicate can only return at most one solution. Thus if no `determined:` *predinfo* is supplied, the default of `determined: ["+...+"]` is assumed.

The `imp:` argument specifies how the predicate is implemented. The default implementation, if this is not supplied, is to represent the predicate explicitly by a set of facts in the agent's knowledge base. For such predicates, the `determined:` argument affects what happens when a new fact is concluded: If the existence of a prior fact would cause the predicate to now be non-deterministic in conflict with the determined mode declarations, then that prior fact is retracted. For example, if we declare:

```
{defpredicate (location $object $location) determined: "+-"}
```

and the fact `(location bill kitchen)` is in the knowledge base, then concluding `(location bill garage)` would cause `(location bill kitchen)` to be removed from the knowledge base.

You can specify a predicate closure as the implementation for a predicate. This acts as a rule for defining the predicate in terms of other predicates. For example:

```
{defpredicate (GrandParent $x $y)
    imp: {pred [$x $y] (Parent $x $z) (Parent $z $y)}
```

You can also specify that the predicate is implemented by a Python or Java function or method. The following defines the `>` predicate in terms of the `__gt__` function of the Python module `operator`:

```
{defpredicate (> $x $y)
  imp: (py_predicate "++" (py_mod "operator" "__gt__"))}
```

7.2.3 Function Declarations

A function declaration is of the form:

{`deffunction` (*ENTITY (VAR)**)
   (`imp:` *TERM)?*
   (`doc:` *STRING)?*
   (`properties:` *TERMLIST)?*}

The `doc:` argument provides a documentation string for the predicate.

The `properties` argument provides a way of associating arbitrary information with the function.

The `imp:` argument specifies how the function is implemented. The default implementation is for the function to construct a simple record structure that can be disassembled by pattern matching. This is similar to the use of functors in Prolog. As with predicates, the implementation can also be specified as a closure or in terms of Python or java code.

7.2.4 Action Declarations

An action declaration is of the form:

{`defaction` (*ENTITY (VAR)**)
   (`imp:` *TERM)?*
   (`doc:` *STRING)?*
   (`features:` *TERMLIST)?*
   (`properties:` *TERMLIST)?* }

The `doc:` argument provides a documentation string for the action.

...

7.2.5 Constant declarations

A constant declaration is of the form:

{`defaction` *ENTITY*
   (`doc:` *STRING)?*
   (`properties:` *TERMLIST)?* }

...

7.2.6 Advice Declarations

...

7.3 Facts

The module includes facts to be added to an agent's knowledge base when the agent loads the module. These appear as expressions of the form `(P 1 (somefun 2))` where `p` is a predicate symbol and the arguments contain no free variables.

8 Future Extensions

There are a number of extensions planned for SPARK:

- A framework for communication between SPARK agents.
- Predicate expressions that refer to conditions that hold over a time interval rather than just at a time point.
- Temporal constraints between tasks.
- A compiler to create code that runs much faster.
- An optional type system to provide support for static type checking where desirable.
- The incorporation of temporal projection techniques to support incremental planning.

9 Comparison with PRS

The key differences between PRS and SPARK are[10]:

- SPARK guarantees that conditions are tested immediately prior to task execution, avoiding delays that can lead to race conditions.
- SPARK has built in support for user advisability of agents.
- PRS is written in Lisp, whereas SPARK is currently implemented in Python/Jython. As a result, SPARK runs on a greater variety of platforms than PRS and has fewer restrictions on distribution.

[10]Not all of these features are available in the initial release of the system.

- For modularity, SPARK has a module system to avoid name clashes among independently developed parts of an agent.
- SPARK was designed so that agents could be compiled into very efficient code (although the current implementation it is interpreted).
- In SPARK, procedures, predicate expressions, and functions are first class objects that can be reasoned about and passed around. The procedure library can be modified on the fly to allow the agent to incorporate new procedures by concluding and retracting facts in the knowledge base.
- Like PRS, SPARK does not require type information, but SPARK will have an optional type system to provide compile-time detection of errors.

10 Acknowledgements

Development of SPARK has been supported by DARPA Contract NBCHD030010 and internal funding from SRI International.

10.1 Timing, Atomicity, and Immediacy

SPARK is very particular about when changes in the knowledge base are allowed to occur. In the descriptions of the various tasks, modifiers and effects that follow in later subsections, constraints on the timing of various events are specified.

In SPARK the notion of immediacy is based on changes to the agent's knowledge base. We use the phrase "A occurs *immediately* after B" to mean that no changes to the agent's knowledge base occur in-between A and B. We also use the phrase "A occurs *atomically*" to mean that no changes to the knowledge base occur during A (other than those caused by A itself).

Some tasks, such as `succeed:` and `fail:`, execute atomically. However in general, tasks can take some nontrivial amount of time, during which the knowledge base may be updated.

Tests of predicate expressions always occur atomically – nothing in the agent's knowledge base, neither beliefs nor procedures, is allowed to change during a test. Not only that, where there a task is conditional on some predicate expression test, execution of the task starts immediately after the test, ensuring that nothing has invalidated the result of the test.

This atomicity restriction even applies to fluents and predicates that are implemented via procedural attachments. Although the execution of a task expression is allowed to take an extended time period, the evaluation of a function or testing of a predicate expression is meant to be effectively instantaneous (at least with respect to the knowledge bases).

References

[1] K. Clark. Negation as failure. In Gallaire and Minker, editors, *Logics and Databases*, pages 293–322. Plenum Press, 1978.

[2] O. Despouys and F. F. Ingrand. Propice-plan: Toward a unified framework for planning and execution. In *ECP*, pages 278–293, 1999.

[3] 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.

[4] R. J. Firby. Adaptive execution in complex dynamic worlds. Technical Report RR-672, 1989.

[5] M. P. Georgeff and F. F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.

[6] M. J. Huber. JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 236–243, Seattle, WA, May 1999.

[7] F. F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pages 43–49, Minneapolis, 1996.

[8] J. Lee, M. J. Huber, P. G. Kenny, and E. H. Durfee. UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications. In *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS)*, pages 842–849, Houston, Texas, 1994.

[9] K. L. Myers. A procedural knowledge approach to task-level control. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 158–165. AAAI Press, 1996.

Copyright

```
#* THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR        *#
#* IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED            *#
#* WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE    *#
#* DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT,        *#
#* INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES        *#
#* (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR        *#
#* SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)        *#
#* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,       *#
#* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING     *#
#* IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE        *#
#* POSSIBILITY OF SUCH DAMAGE.                                               *#
#****************************************************************************#
```