SPARK Reference Manual

For Version 0.8

Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, California 94025

August 4, 2006

The writing of this guide was supported by SRI International and by DARPA Contract Number NBCHD030010.

Abstract

SPARK is a new agent framework being developed at the Artificial Intelligence Center of SRI International. SPARK is a Belief Desire Intention (BDI) Agent framework in the procedural reasoning style and has been strongly influenced by its predecessor the Procedural Reasoning System (PRS). It is goal directed and event driven, supports flexible plan execution, and has meta-level reasoning and introspection capabilities. 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.

Contents

1 Introduction

This document provides a description of SPARK, an 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)[4].

The SPARK agent framework allows the development of active systems that interact with a constantly changing and 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 that 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* 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.

SPARK addresses some key limitations of earlier PRS predecessor. SPARK has built in support for user advisability of agent. SPARK is also 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 PRS.

1.1 The SPARK Agent Architecture

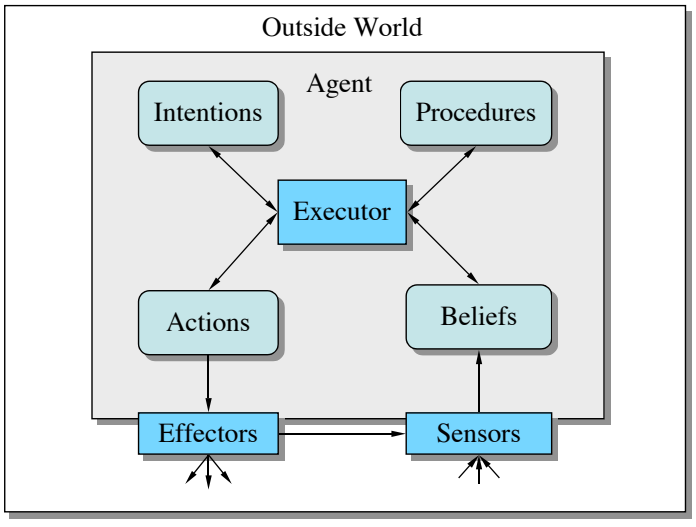

Figure 1: A SPARK Agent

Figure 1 illustrates the SPARK agent architecture. Each agent is embedded in the world and interacts with the world through sensors and effectors to perform some tasks. Each SPARK agent consists of (1) a knowledge bases of beliefs, including both ground facts and rules; (2) a set of *actions*, both primitive and non-primitive; (3) a library of *procedures*, describing how hierarchical networks of actions and tests may be performed to complete certain tasks or respond to certain events; (4) *intentions* containing the task instances and the corresponding procedure instances chosen for execution; (5) the *Executor* whose role is to manage the execution of intentions, thus updating the agent's database.

1.1.1 Beliefs

Each SPARK agent maintains a knowledge base (KB) of beliefs about the world which is represented as a simple logical database with ground facts and rules, and it carries a logic programming semantics [10]. Beliefs about the world are updated by the agent's sensors and through the agent performing tasks.

The beliefs can also describe the internal state of the agent, which is referred to as *metalevel facts*. Metalevel facts describe the current goals, intentions and other aspects of the internal state of the agent. Metalevel facts are updated by agent's internal events

In addition, procedures are also first-order objects in SPARK knowledge base, allowing procedures to be added, modified at run-time.

1.1.2 Actions

SPARK actions are hierarchical - they can be either primitive or non-primitive. Primitive actions cause effects through the effectors, which may directly or indirectly change the agents knowledge base. Non-primitive actions are expanded by the executor according to the procedures the agent has available, similar to task decomposition in hierarchical task network (HTN) planning.

1.1.3 Procedures

Knowledge about how to perform tasks or how to react to certain events is represented as procedures in SPARK. Each procedure consists of (1) a *cue* which specifies what task, goal, or event triggers this procedure; (2) a set of *preconditions* that specify when each procedure is applicable. and (3) a *body* of task network that specifies how the the task may be completed. Note that unlike planning systems that subgoal on preconditions, SPARK does *not* subgoal on its preconditions. The procedure bodies use an extended HTN representation including sequencing, parallel, iteration and conditional branching. Each procedure is executed one step at a time and the execution commonly consists of testing the agent's beliefs followed by executing some action based on the result of the tests.

There can also be metalevel procedures that are used to encode decisions that influence the operations of SPARK. For example, metalevel procedures can be used to choose among multiple applicable procedures or to modify the intentions. Details of meta-level reasoning are discussed in section 8.3.

1.1.4 Intentions

The set of procedure instances that the agent is currently executing are called the *intentions* of the agent. The *intention structure* of a SPARK agent consists of forest of trees of procedure instances where each procedure instance has a triggering *event* that caused it to be intended – the triggering event is what the *cue* matched. Multiple intentions may be active simultaneously.

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.

1.1.5 The Executor

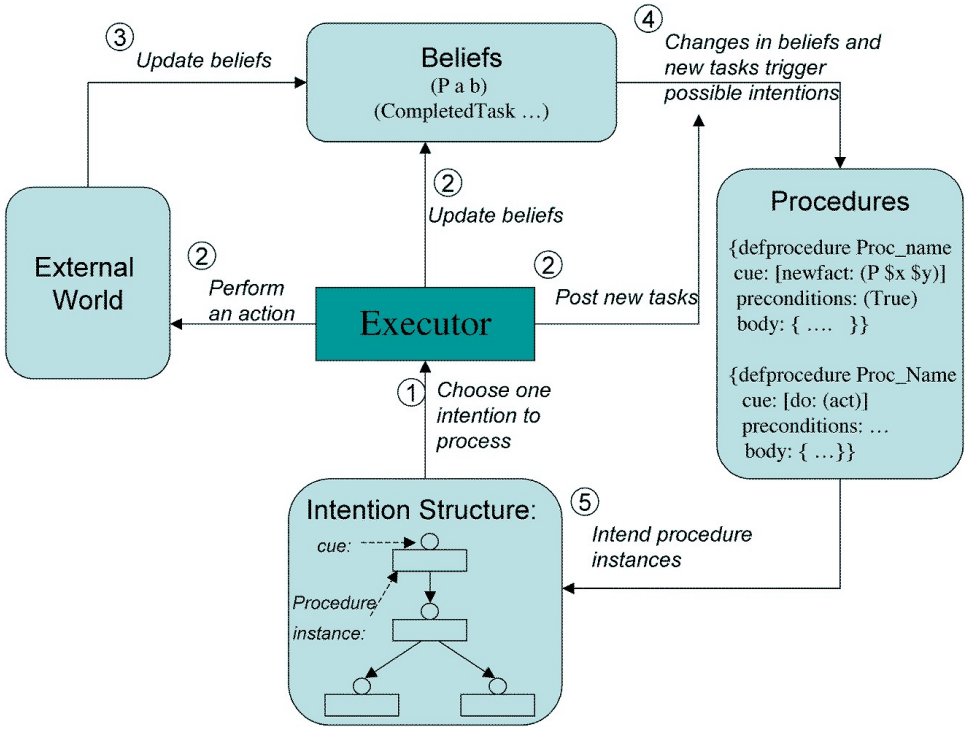

Figure 2: SPARK Interpreter Loop

At SPARK's core is the executor whose role is to manage the execution of intentions. The executor process one intention at a time (1), and as an intention is processed, new tasks may be posted

or beliefs may be updated (2). In addition, external events may also cause beliefs to be updated (3). These changes trigger the creation of new intentions based on available procedures through matching the new tasks or beliefs with the *cue* of procedures. A subset of the applicable procedure instances (i.e. procedures whose preconditions are met) are added to the intention structure. This subset is determined according to SPARK's default strategies unless this decision is over-written by meta-level reasoning procedures discussed in 8.3. Once new intentions are added to the intention struture, the executor again chooses one intention to process, and performs a single step of the chosen procedure instance, resulting in the execution of some action (primitive or non-primitive), which in term posts new tasks and updates the agent's beliefs.

1.2 SPARK-L: The SPARK Agent Language

The SPARK agent language (SPARK-L) is designed to bridge the gap between agent languages that emphasize on formal properties (semantically well-grounded, suited to formal analysis, but do not scale well) [9, 5, 6], and agent languages that emphasize on application development (expressive control mechamisms, suited to demanding application, but not suited to formal analysis) [8, 1, 3, 7].

Usually, an agent's core knowledge comes from files containing SPARK-L source code (although other SPARK-L code can be read from other sources at run time). The SPARK-L source files 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 function or that `located` is define 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

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

1.2.1 SPARK Data Types

SPARK works with a variety of data types. These are divided into the *core* data types and the *non-core* data types. Values constructed from the core data types have textual representations that SPARK can parse directly and these values are the most widely used values in SPARK. These core data types (together with example values represented in SPARK-L) are divided into *primitive types*:

- *Integers* (e.g., 1 and -42)
- *Floats* (e.g., 1.0 and -1.07e-4)
- *Strings* (e.g., "one"). The backslash character \ is used to include various characters that would otherwise be hard to represent in the string. Two strings are equal if they contain the same sequence of characters.
- *Symbols* (e.g., one) act as identifiers or names. Symbols ordinarily consist of alphanumeric characters and characters from ._*/%&@!?<>=+-, but not starting with +, -, or a digit. You can have other symbols, but in his case, the symbol characters need to be written between | characters, for example |has space characters|. As in the case of strings the \ character can be used to escape any occurrences of the | character in the symbol name. Two symbols are equal if they contain the same sequence of characters.
- *Variables designators* (e.g., $one) that represent variables. These start with one or more $ characters followed by one or more alphanumeric characters.

and *compound types*:

- *Lists* (e.g., [1 2 3]) consist of a sequence of *elements*. Two lists are equal if their elements are equal.
- *Structures* (e.g., (foo 1 2 3)) consist of a *functor* symbol and a sequence of *arguments*. Two structures are equal if they have the same functor and their respective components are equal. For syntactic convenience, SPARK-L also allows constructs of the form {foo X Y Z}, bar: X Y Z, 'X, ,X, +X, and -X however these simply correspond to structures with unusual functor symbols – (|foo{}| X Y Z), (|bar:| X Y Z), (|'#| X), (|,#| X), (|+#| X), and (|-#| X) respectively.

Non-core SPARK data types include closures, failures, and arbitrary Python or Java objects.

All 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 means that all the core data values are immutable – they cannot change. You cannot modify a string by adding or deleting characters, you cannot change the elements of a list, you cannot change the functor or components of a structure, and so on.

It is possible to create "mutable" objects, such as queues, that can be passed around as SPARK data values. However, the above restriction requires that two "mutable object" values are equal if and only if they refer to the exactly the same object. Thus two different queues may contain the same elements at some time, but they will never be considered 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.

1.2.2 SPARK-L Syntactic Structures

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 the agent's knowledge base and procedures that define how to respond to events. These specifications and facts are represented syntactically by core data values, similar to the way that programs are represented by data structures in Lisp and Prolog. For example, the SPARK-L source for multiplying 1 and 2 is represented by a structure with functor symbol * and arguments 1 and 2, i.e., (* 1 2).

The ability to represent SPARK-L source as SPARK core data values is very powerful however it can lead to confusion. In this paper, when we wish to represent SPARK core data values using the SPARK-L syntax, we will use a sans serif font, for example 3. When we wish to write examples of SPARK-L source, we will use a typewriter font, for example `(* 1 2)`.

The SPARK-L language is made up of different types of *expressions* including the following:

- *Variables* (*VAR*).
- *Terms expressions* (*TERM*) describe data values and the application of functions over these data values. For example, `(+ 1 $x)` is a term expression representing one more than the value of variable `$x`. Variables are instances of term expressions.
- *Logical expressions* (*LOG*) that describe relationships between data values. For example, `(and (Member $x $list) (P $x))` is a logical expression that is true when the value of variable `$x` is both a member of the value of of variable `$list` and satisfies the predicate `P`.
- *Action expressions* (*ACT*) describe parameterized actions that have some effect on the agent or its environment. For example, `(fix $x)` is an action expression that describes applying the action `fix` to the value of variable `$x`.
- *Task network expression* (*TASK*) specify actions to be performed and conditions to be achieved. For example, `[forin: $x $list [do: (fix $x)]]` is a task network expression that will cause the action `(fix $x)` to be performed with `$x` bound to each element of `$list` in turn.
- *Statements* provide means for top level declarations of predicates, actions, functions, and so on. For example, `{defpredicate (P $x)}` declares `P` to be a predicate that takes one parameter.

1.2.3 Packages

SPARK deals with objects such as predicates, actions, functions, and constants. In the SPARK-L language these objects are named by identifiers, such as `foo`. To be able to reference an object using an identifier, there must be a declaration of the object. The declaration states that there exists a named object and states how that object is used (e.g., `foo` is an action and it takes three arguments). These declarations are immutable; they cannot be changed once execution has begun.

SPARK-L files can share declarations of identifiers. Each file has a *package* associated with it. Any identifier beginning with an underscore is considered *private* and can only be seen within the file, but all other identifiers are considered *public* and can be used in any other file with the same package. A file specifies the package it is using via a statement of the form "`package:` *NAME*" (or "`module:` *NAME*" for backwards compatibility). The `package:` statement should come at the start of the file. If the `package:` statement is missing, then the package name is taken to be the same as the file name.

A public identifier is a unique name within a package, but different packages may use the same identifier to refer to different objects. To distinuish between the object with identifier `foo` declared in package `a.b` from that declared in package `c.d`, each named object has a fully qualified name that is a distinct symbol, in this case `a.b.foo` and `c.d.foo` respectively.

Packages can share declarations with other packages by *exporting* the identifiers. A file in one package makes the symbol `foo` available to other packages using a statement of the form `export: foo`. A file in another package can add that declaration to its own package (and thus make it available to all other files in the same package) using a statement of the form "`importfrom: a.b foo bar`" to import specific identifiers or "`importall: a.b`" to import all identifiers from package `a.b`[1].

1.3 An Example

Figure 3 shows a simple file written in SPARK-L. This file declares a predicate `InterestedIn`, an action `forwardMessage`, two procedures `forwardMessageUnlessSpam` and `reportSpam` that represent possible ways of performing `forwardMessage` action, and three facts about people's interests. This file also imports entites `IsSpam`, `subjectOf`, and `sendTo` from the `message` package, and imports all entities in the `subject` and `person` packages, and the `export:` statement allows other files to import declarations of `InterestedIn` and `forwardMessage`. The `+` prefix before the `$message` variable in the `defaction` (and the cues of the procedures) indicates that the `$message` parameter is an input parameter, and the corresponding actual parameter must be bound before calling the action.

`(InterestedIn Bill implementation)` states that `Bill` is interested in the subject `implementation`, where `Bill` and `implementation` are *constants* and imported from other packages. The procedure `forwardMessageUnlessSpam` can be read as "when you want to perform the task `[do: (forwardMessage $message)]` where the `$message` is not a spam, then forward this message to everyone who is interested in the subject of this message".

[1]For backwards compatibility, the degenerate case of "`importfrom:` *NAME*" with no identifiers specified is taken to be equivalent to "`importall:` *NAME*".

```
importfrom: message IsSpam subjectOf sendTo
importall: subject
importall: 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 3: A module written in SPARK-L

2 Term Expressions and Functions

The syntactic structures of the SPARK language that describe data values are terms expressions (*TERM*s). In this section, we will describe the syntax and semantics of term expressions, and functions which are applied to data values to produce other data values.

There are two things that SPARK can do with a term expression:

- SPARK may *evaluate* the term expression to produce a value, based on the values that any variables in the term expression have been bound to. For example, if `$x` has been bound to 2, then the term expression `[$x]` can be evaluated to give the result [2], a list containing one element, 2. In general, a term expression can only be evaluated if the variables in it have already been bound.
- SPARK may *match* a term expression to an existing value, in the process binding any variables that have not yet been bound. For example, if we match the term expression `[$x]` with the list [2], then the match will succeed and in the process bind `$x` to the value 2.

*TERM*s are of the following form:

- An integers, floating-point number, or string – evaluate to their values.
- Variable designators such as `$x` evaluate to the value that the variable has been previously bound to.
- Symbols such as `Bill` are treated as identifiers relative to the current package and evaluate to a fixed value.
- Structure term expressions, such as `(+ 1 2)` interpret the functor symbol as an identifier for a *function* and the structure arguments as parameters to pass to the function. For example, `(+ 1 2)` evaluates to the integer 3. By convention, function names begin with a lowercase letter.
- List term expressions such as `[1 (+ 1 2)]` are evaluated by evaluating the elements and creating a list of the resulting values.

2.1 Semantics for term expressions

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 $z]]` can be matched to the value [1 [2 3]], binding `$x`=1, `$y`=2, and `$z`=3.

2.2 Functions

A function computes a value from its inputs and the current state of the world, and it is supposed to have no effect on the world. Functions can be *static* or *dynamic*. Functions whose result does not depend upon the state of the knowledge base (e.g., the functions `*` and `min`) 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 example, the function `(getCurrentTime)`, which returns the current time of the system, is a dynamic function.

Functions can be implemented in a number of ways. The default implementation is for the function to construct/deconstruct a record-like structure, in the same way that functors in logic programming languages are used to construct terms. A function can also be implemented via a function closure (expressing it as a term expression), a Python function, or a Java method. More details of function implementations can be found in Section 7.3, and section 5 describes different kinds of closures.

Some functions are *invertible* in that the input parameters can be uniquely determined from the result. The default, structure constructing implementations, and some functions implemented as Python/Java code are invertible. Term expressions constructed from such functions can be used as patterns to match values, binding variables in the process. For example, suppose we have a structure constructing function `foo`, then the term `(foo 1 $x)` could be matched to the value `(foo 1 2)` binding `$x`=2 in the process.

Not all functions have term expressions to evaluate as parameters. Some special functions interpret their parameters in different ways. For example:

- `‘`*X* which is "syntactic sugar" for `(|‘#|` *X*`)` does not evaluate *X* as a term expression. Instead it returns *X* as a value. For example, `‘(+ 1 2)` evaluates to the structure value `(+ 1 2)` not the integer value 3. Similar to the LISP language, it is possible to insert evaluated values into the result using `,`*Y* (which is syntactically equivalent to `(|,#|` *Y*`)`). For example, `‘[1 ,(+ 1 1) 3]` evaluates to `[1 2 3]`.
- Closures such as `{pred [+$x] (and (P $x)(Q $x))}`, which are described in Section 5 represent functions, predicates, actions, etc., that can be constructed on the fly and passed around as data values. The special function `|pred{}|` takes an argument list and a logical expression as its parameters.
- `(if` *LOG X Y*`)` returns *X* if the logical expression *LOG* has a solution, or returns *Y* otherwise. For example, `(if (Father John Fred) "yes" "no")` returns `"yes"` if `Fred` is the father of `John`, and returns `"no"` otherwise.
- Another example of complex function is `(solutionspat [`*(VAR)**`]` *LOG X*`)` where `[`*(VAR)**`]` is a list of variables local to *LOG*. This function returns a list of the values of term expression *X* for each set of bindings for the variables in `[`*(VAR)**`]` that are solutions to logical expression *LOG*. For example, given the three facts `(P 1 2)`, `(P 3 4)`, and `(P 3 5)`, the expression `(solutionspat [$x $y] (P $x $y) (+ $x $y))` returns `[3 7 8]`.

3 Logical Expressions

The syntactic structures of the SPARK language that describe relationships between data values are *logical expressions* (*LOG*s). These are represented by structures such as `(P 1 2)`, where the functor symbol is an identifier (relative to the current package) that identifies a *predicate* and the structure arguments are the parameters of that predicate. By convention, predicate names usually begin with an uppercase letter.

There are four things that SPARK can do with a logical expression:

- SPARK may *test* the logical expression to find bindings of the currently unbound variables that satisfy the logical expression. After testing a logical expression, all the free variables (those that are not explicitly quantified) are bound. Sometimes SPARK only needs a single set of bindings and will take the first one it finds. For example, the `if` special function only looks for one solution to its logical expression argument. Sometimes SPARK needs to find all solutions. For example, the `solutionspat` special function finds all solutions to its logical expression argument.
- SPARK may *conclude* a logical expression into the knowledge base, adding a new fact as being true.
- SPARK may *retract* a logical expression, removing a fact from the knowledge base.
- SPARK may attempt to *achieve* the logical expression, executing procedures to affect the world so that the logical expression becomes true.

Just as in the case of functions, most predicates take term expressions as parameters. However, some special predicates take other expression types as parameters. These include the following (note: we use *(*`X`*)** to represent zero or more occurrences of an expression of category `X` and *[*`X`*]* to represent an optional expression of category `X`.):

- `(and` *(LOG)**`)` is the conjunctive logical connective, e.g., `(and (P $x) (Q) (R))`.
- `(or` *(LOG)**`)` is the disjunctive logical connective, e.g., `(or (P $y) (Q) (Z))`.
- `(not` *LOG*`)` is logical negation, e.g., `(not (P 1))`.
- `(exists [`*(VAR)**`]` *LOG*`)` – where `[`*(VAR)**`]` is `[`*(VAR)**`]` – is the existential quantifier, e.g., `(exists [$x $y] (P $x $y))`

In this section, we will first introduce logic programming with predicates as the logical expressions in SPARK are based on concepts from logic programming. We will then discuss logical expressions in SPARK, including how to test a logical expression in SPARK, and various ways logical expressions can be implemented in SPARK.

3.1 Logic Programming with Predicates

The logical expressions of SPARK are based on concepts from logic programming languages such as Prolog. A predicate expression represents some condition, where variables represent unknowns. The act of testing a predicate expression finds bindings (i.e., values) for those variables for which the condition is true. A set of bindings that makes a predicate expression true is called a *solution* for the predicate expression. Sometimes, SPARK only needs to find the first solution (if there is one), sometimes it needs to find all solutions.

Consider the predicate `(HasParent $child $parent)`, which holds when `$child` is bound to a value representing a person who has a parent represented by the value that `$parent` is bound to. We can represent the fact that "Alice" has parents "Betty" and "Charles" and that "Charles" has parents "Denise" and "Edward" by a set of facts

```
(HasParent "Alice" "Betty")
(HasParent "Alice" "Charles")
(HasParent "Charles" "Denise")
(HasParent "Charles" "Edward")
```

If we pose a query `(HasParent $c $p)` (where neither `$c` nor `$p` previously bound to anything), SPARK will identify four solutions:

- $c="Alice",$p="Betty",
- $c="Alice",$p="Charles",
- $c="Charles",$p="Denise",
- $c="Charles",$p="Edward",

We can pose a more specific query by replacing one or more of the variable arguments with a constant. For example, the query `(HasParent $c "Betty")` has only one solutions, $c="Alice".

Alternatively, if one of variables has already been bound to a value, then this will restrict the possible solutions. For example, if we pose the query `(and (HasParent $c $p) (HasParent $p $g))`, then SPARK will look for solutions to `(HasParent $c $p)` and for each solution, it will try to find a solution to `(HasParent $p $g)` where `$p` now has been bound. There will be no solutions for `(HasParent $p $g)` when `$p` is bound to `"Betty"`, `"Denise"`, or `"Edward"`, but when `$p` is bound to `"Charles"`, there are two solutions, `$g="Denise"` and `$g="Edward"`. Thus SPARK will find two solutions to the compound query:

- $c="Alice",$p="Charles",$g="Denise" and
- $c="Alice",$p="Charles",$g="Edward".

3.2 Logical Expressions in SPARK

It is important to note that functions and predicates (logical expressions) have no effects on the state of the world, they only bind variables: if something has an effect on the world, then it is an action. This distinction is important because during the execution of a logical expression, SPARK promises that the state of the world does not change. For example, one consequence is that repeated occurrences of the same term expression within a logical expression should evaluate to the same value, since they are meant to be evaluated with the world in a fixed state.

When testing a logical expression there may be multiple alternative sets of bindings for the variables that make the logical expression true. However, once the SPARK executor starts to execute some action, it must select and commit to one set of bindings for the variables.

After successfully testing a logical expression in SPARK, all the (free) variables that appeared in the term expression arguments passed to the expression must be bound to data values. For example, if `$x` is already bound, and we wish to test the logical 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 instances of the `P` predicate. Once the logical 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[2].

3.2.1 Testing Compound Logical Expressions

When dealing with compound logical 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 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.

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

- SPARK currently applies the Closed World Assumption to predicates and uses negation-as-failure [2] to test negated predicate expressions, that is, `(not (P 1))` is equated with the failure to prove `(P 1)`[3].

  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.

3.3 Different Types of Logical Expressions

As with functions, we have *static* and *dynamic* predicates. Predicates whose solutions do not depend upon the state of the knowledge base are called *static* predicates. For example, predicate `(Append $L1 $L2 $L)`, which appends lists `$L1` and `$L2` and puts the result in list `$L` is a static predicate – the result is not affected by when you test it, so long as `$L1`, and `$L2` have the same values. A predicate fluent whose solutions depend deterministically upon the state of the knowledge base is called *dynamic*. For example, predicate `(Salary $personid $amount)` about a person's salary is a dynamic predicate – the salary may change over time.

Predicates may also have different types of implementations. The default representation is for a predicate to be represented by a set of ground facts in the agent's KB. Similar to functions, predicates can also be represented as predicate closures, Python function, or Java methods. More details of predicate implementations can be found in Section 7.2, and section 5 describes different kinds of closures.

[3]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.

4 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*. Syntactically, each task network expression *TASK* is a list of keyword-structures, usually a single keyword structure, such as `do: (foo)` or `achieve: (P $x)`. The empty list, `[]`, is equivalent to the basic task `[succeed:]` which acts as a no-op.

Syntactically, a task network expression is a list of *task components*, usually one, where each task component is a structure such as `do: (foo)` or `achieve: (P $x)`.

A task component may represent a *basic task*, which performs some action or achieves some state. For example, the `do: (forwardMessage $m)` is a basic task component that calls for the action `(forwardMessage $m)` to be performed.

A task component may also represent a *compound task network* that is a combination of simpler task network expressions. For example,

```
[forall: [$person] (InterestedIn $person (subjectOf $message))
             [do: (sendTo $person $message)]]
```

is a compound task network that calls for the action `(sendTo $person $message)` to be performed for all people interested in the message.

4.1 Basic Task Components

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 components include:

- `do:` *ACT* – where *ACT* is `(`*ID (TERM)**`)` and *ID* 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:` *LOG*
  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 with a cue 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[4].

- `context:` *LOG(TERM)**
  e.g., `[context: (HasBoss $employee $boss) "Can't find the boss of %s" $employee]`

  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 and if the optional format string and argument terms exist, a formatted message is printed out.

- `conclude:` *LOG*
  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.

- `retractall:` `[`*(VAR)**`]` *LOG*
  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:` *LOG*
  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: []` *LOG*

Note: The value of dynamic functions depend upon the time that the functions are evaluated. Since basic tasks of the form `do:` *ACT* and `achieve:` *LOG* may take an arbitrarily long time to execute, the state of the knowledge bases may change over the execution of these tasks, and the value of any dynamic functions used in the parameters may change. Therefore, to avoid unpredictable results, it is recommended that only static functions should be used in parameters to these basic tasks. This recommendation is neither tested nor enforced by SPARK.

4.2 Compound Task Components

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

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

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[5].

Compound task networks 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 [6]. If the execution of a subtask 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:` *(LOG TASK)**
  e.g., `[select: (Tired) [do: (sleep)] (Hungry) [do: (eat)]]`

  The arguments to the `select:` compound task network expression are alternating *LOG*s and *TASK*s. Each pair represents a possible alternative execution path. Execution of the `select:` task network expressions requires testing the *LOG*s *in order*, finding the first *LOG* that has a solution, choosing a variable binding corresponding to one solution, and then executing the corresponding *TASK*. If no *LOG* 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 *LOG* 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 rather than 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 (*LOG*-*TASK* pair) cannot be used after the `select:` task network expression.

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

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

- `wait:` *(LOG 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 *LOG*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 *LOG* 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:` `[`*(VAR)**`]` *LOG 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:` [*(VAR)**] *LOG 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 *TASK* 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.

- `forallp:` [*(VAR)**] *LOG TASK*
  e.g., `[forallp: [$x] (Wall $x) [do: (paint $x blue)]]`

  forallp is similar to forall, except that it executes the subtask network for each solution in parallel.

- `forin:` *VAR TERM TASK*
  e.g., `[forin: $person $list [do: (sendmail $person)]]`

  In a forin task network, we bind *VAR* to each element in the list *TERM*, and executes the task network that follows in sequence.

4.3 Execution of Task Network Expressions

In general, the execution of a task network expression consists of executing some sequence of basic task components, where the order and timing of those basic task components is directed by the compound task components.

4.3.1 Normal Execution of Task Network Expressions

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 logical expression. There may be many possible solutions, however the executor commits to one solution by binding the local variables appropriately, and then proceeds to execute the relevant task. As more expressions are tested and basic tasks executed, additional variable bindings are made. Variables that are bound in a loop body are local to the loop body and are considered fresh variables on each iteration. Once the executor has committed to a variable binding, that variable binding will remain valid for the entire scope of the variable unless a task network expression in which the variable is bound raises an exception. Consider the following procedure:

```
{defprocedure ExampleProcedure
 cue: [do: (foo)]
```

```
precondition: (P $a)
body:
 [seq: [context: (Q $a $b)]
       [do: (bar $a $b $c)]
       [forin: $x $c [do: (act1 $x)]
       [try: [seq: [do: (act1 $x)] [do: (act2 $y)]]
             []
             [seq: [do: (act3 $x)] [do: (act4 $y)]]
             []]]
}
```

The body of this procedure is a compound task network. There are many local variables in this procedure, including `$a, $b, $c, $x, $y`. Variable `$a` is bound through testing the precondition of this procedure. During the normal execution of this procedures, we first test the logical expression `(Q $a $b)`. If the test is successful, we may find one or more bindings for variable `$b`, but SPARK commits to only one binding. The executor then proceeds to the next task `[do: (bar $a $b $c)]` where variables `$a` and `$b` are bound, but `$c` is unbound. If this task completes successfully, it is expected to bind variable `$c`; otherwise, if it fails, it is expected to return some reason for failure. In the forin task network, variable `$x` is local to the forin loop, and is bound to different values on different iterations of forin. In the `[try:]` task network, SPARK first attempts to execute `[seq: [do: (act1 $x)] [do: (act2 $y)]]`: if it succeeds, variables `$x` and `$y` are bound, and [] gets executed and `[try:]` succeeds; but if it fails, then the bindings for variables `$x` and `$y` are undone, and task network `[seq: [do: (act3 $x)] [do: (act4 $y)]]` gets executed, rebinding variables `$x` and `$y`.

4.3.2 Exceptions: Failures and Errors

As we see in the previous example, it is possible for tasks (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 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.

More advanced failure-handling mechanisms are capable using SPARK's meta-Level, which is discussed in Section 8.

5 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 actions that operate on other functions, predicates and tasks.

5.1 Function Closures

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 [`*(VAR)**`]` *TERM*}

Where `[`*(VAR)**`]` is a list of variables that are formal parameters to be bound when the closure is applied to some arguments. The formal parameters of a function closure do not take mode annotations, since they are all input parameters. You apply a function closure to arguments using the `applyfun` function, thus

```
(applyfun {fun [$x] (- $x 1)} 9)
```

would evaluate to 8.

5.2 Predicate Closures

Predicate closures are data values that represent predicates. Basic predicate closures are of the form:

{`pred [`*(ARG)**`]` *LOG*}

Note that a predicate closure takes an argument list which can include mode annotations. 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. If this closure encloses a dynamic predicate, it is always tested against the agent's knowledge base at the time the closure is applied.

5.3 Task Network Closures

Task network closures are data values that encapsulate a task network expression to produce an action. Task network closures are of the form:

{`task [`*(ARG)**`]` *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)`.

5.4 Variable Scoping in Closures

The scopes of variables in closures are determined by braces {}. Each layer of brace {} (i.e., each expression whose functor is of the form `|...{}|`) introduces 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: thus `$$$x` would refer to the variable `$x` two scopes out.

Consider the following obtuse logical expression:
`(and (Member $x [1 2]) (= $y (applyfun {fun [$z] (* $z $$x)} 4)))`
the `$$x` in the closure refers to the earlier `$x` in the `Member` expression. For each solution to the `Member` expression, `$y` is bound to a new function closure. The whole logical expression has two solutions: `$x`=`1`, `$y`=`4` and `$x`=`2`, `$y`=`8`:

6 Statements

SPARK-L files can contain statements that affect the namespace of the file and introduce loading dependencies. These are

- `package:` *PATH* – where *PATH* is the name of a package
  e.g., `package: foo.bar`. This indicates that identifiers in this file should be interpreted with respect to the *PATH* package. This also introduces a load dependency, requiring that the file with logical name *PATH* and files it depends upon, be loaded when this file is loaded.

- `importall:` *PATH* – where *PATH* is the name of a package
  e.g., `importall: spark.lang.list`. This indicates that all the identifiers exported from the package *PATH* should become visible within the package of this file. This also introduces a load dependency, that files of package *PATH* must be loaded when this file is loaded.

- `importfrom:` *PATH(ID)** – where *PATH* is the name of a package and *(ID)** is a sequence of identifiers
  e.g., `importfrom: spark.lang.list Member Length`. This indicates that the identifiers *(ID)**, which must be exported from the package *PATH*, should become visible within the package of this file. This also introduces a load dependency.

- `requires:` *PATH* – where *PATH* is the name of a file
  e.g., `requires: foo.bar2`. This introduces a load dependency, requiring that file *PATH* be loaded when this file is loaded.

- `export:` *(ID)** – where *(ID)** is a sequence of identifiers
  e.g., `export: MyPred myAction`. This indicates that the identifiers listed should be exported from the package of this file (enabling them to be imported into another package)

- `exportall:`
  e.g., `exportall:`. This indicates that all non-local identifiers declared in this file should be exported from the package of this file (enabling them to be imported into another package). The locally declared identifiers (i.e., those whose name starts with "_") are not exported, nor are identifiers declared in different files.

7 Declarations and Implementations

As mentioned earlier, SPARK domain knowledge is grouped into files, where each file contains a sequence of statements that declare and define entities. A symbol *declaration* explicitly declares the existence of a symbol and states how that symbol should be used: as a constant, function, predicate, action, procedure, and so on. As part of the declaration, an identifier may have a definition, called an *implementation* – for example that `+` is defined by a Python function or that predicate `Located` is defined by a set of knowledge base facts, and so on. This section describes different types of declarations in SPARK and their implementations.

We have adopted the following naming conventions to help distinguish different types of entities.

- function names are written with mixed case where the first letter is lowercase, e.g., `someRandomFunction`
- predicate names are written with mixed case where the first letter is uppercase, e.g., `SomeRandomPredicate` (often a category or a declarative verb or clause, e.g., `Person`, `LikesToEat`)
- action names are written with mixed case where the first letter is lowercase, e.g., `someAction` (usually an imperative verb or clause, e.g., `eatFish`)
- module names are written with all lowercase letters and digits where underscores are allowed if necessary, e.g., `foo.examples`
- procedure names are written with mixed case

7.1 Constant Declarations and Implementations

A constant declaration is of the form:

{`defconstant` *ID*
  [`doc:` *STRING*]}

This declares *ID* to be a constant that evaluates to a symbol. Here are some examples of constant declarations that are used with the example shown in section 1.3.

```
{defconstant Bob doc: "person Bob"}
 {defconstant documentation doc: "an email category"}
```

7.2 Predicate Declarations and Implementations

A predicate declaration is of the form:

{`defpredicate (`*ID (MVAR)**`)`
  [`imp:` *TERM*]

[`doc:` *STRING]*
[`properties:` `[`*(TERM)**`]`*]}*

The arguments of the predicate are specified by a sequence of *MVAR*s, where each *MVAR* is of the form `+`*VAR*, *VAR*, or `-`*VAR*. A *MVAR* of the form `+`*VAR* indicates that whenever the predicate is tested, the corresponding argument must always be evaluable. SPARK will do static checking of procedures to ensure that this is the case and will report errors if it is not. A MVAR of the form `-`*VAR* indicates that whenever the predicate is tested, the corresponding argument must not be evaluable. SPARK does not currently use this information to check the use of predicates. A *MVAR* of the form *VAR*, has no constraint.

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

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

The `imp:` argument is a term expression that specifies how the predicate is implemented. There are three types of predicate implementations:

1. Extensional (default) - the predicate is defined by a set of ground facts in the agent's KB.
2. Closure - the predicate is defined in terms of other predicates via a predicate closure.
3. Code - a python function or java static method is used to test a ground fact or to generate solutions if there are unbound variables in the parameters.

If the `imp:` argument is not specified, then it defaults to an extensional implementation. Occasionally, you may want to impose *functional* constraints on an extensional implementation that forbid there being two facts with the same "key" value. For example, consider a predicate to represent people's salaries: `(Salary $personid $amount)`. A new fact about `$personid` should replace any previous facts – there is a constraint that when the first parameter is supplied, the predicate must return no more than one solution. To achieve this effect we use `imp: (determined "+-")` as in:

```
{defpredicate (Salary $personid $amount) imp: (determined: "+-")}
```

The argument `"+-"` to `determined` is a *mode string*. A mode string is a sequence of "`+`"s and "`-`"s, with each character corresponding to an argument. In this situation, the "`+`" arguments in the mode string constitute the "key". To be precise, the mode string specifies conditions under which the predicate returns a unique solution: "`+`" indicates that the argument needs to be bound to a ground value. "`-`" indicates that the argument need not be ground.

To cater for multiple functional constraints, `determined` allows any number of mode string arguments. For example, `(determined "+-" "-+")` means that there is at most one solution if either the first or second parameter is given. The default behavior when `imp:` is not specified is equivalent to `(determined)`, meaning that when a new fact is concluded nothing is retracted.

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, the following declares the `GrandParent` predicate in terms of the `Parent` predicate using a predicate closure:

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

You can also declare that SPARK should use a Python function or Java method to test a predicate or generate solutions. In this case, you must specify precisely which parameters of the predicate must be evaluated and passed to the function, and what to do with the result of the function. If in a particular use of the predicate, the specified parameters are not evaluable, then a runtime exception will be raised.

You specify which parameters must be bound using a mode string. As mentioned before, a mode string contains a "`+`" for each parameter that must be evaluable and a "`-`" for each parameter that need not be bound (if fact, other characters are also allowed in mode strings, but discussion of these complexities will be left to the end of the subsection). Here, each "`+`" parameter is an input parameter whose value will be supplied to the function, and each "`-`" parameter is an output parameter that is to be bound by testing the predicate. The input parameters are passed to the function in the same order as they appear in the logical expression.

You specify the Python function to use with the form `(pyMod` *PythonModuleString FunctionName*`)`. For example, `(pyMod "a.b" "foo")` refers to the function `foo` in the Python module `a.b`. Alternatively, if the first input parameter is a Python object, you can specify a method of the object to call using the form `(pyMeth`*MethodName*`)`. The remaining input parameters are passed in as parameters to the method. For example, `(pyMeth "items")` refers to the method `items`. When running SPARK using Jython, you can access Java static and non-static methods the same way: `(pyMod "x.y.Class" "staticMethod")` and `(pyMeth "someMethod")`.

The interpretation of the returned value is the most complex part of defining a SPARK predicate in terms of Python or Java code. It depends upon the number of output parameters and whether the function is generating at most one solution or any number of solutions:

- zero output parameters, at most one solution – The function needs to return whether or not the predicate is satisfied by the input parameters. The return value is interpreted as a Boolean: True for one solution, False for no solution.
- one output parameter, at most one solution – The function needs to return whether there is a solution and if so, what the binding of the output parameter is. The returned value is interpreted as a value for the output parameter, with the value `None` representing no solution.
- one output parameter, multiple solutions – The function needs to return all the values of the output parameter for which the predicate is true. The returned value is interpreted as a sequence of alternative values for the output parameter. A zero length sequence represents no solutions.
- multiple output parameters, at most one solution – The function needs to return whether there is a solution and if so, what the bindings of the output parameters are. The returned

value is interpreted as a sequence of values for the output parameter, with the value `None` representing no solution.

- multiple output parameters, multiple solutions – The function needs to return all sets of output parameter values for which the predicate is true. The returned value is interpreted as a sequence of alternative solutions, with each element/solution being a sequence of values for the output parameters.

The number of output parameters is given by the mode string. If the function is to return a single solution, you declare the predicate {`defpredicate` *Name* `imp: (pyPredicate` *Mode Fun*`)`}. If the function is to return multiple solutions, you declare the predicate {`defpredicate` *Name* `imp: (pyPredicateSeq` *Mode Fun*`)`}.

For example, the following defines the `>` predicate, taking two arguments as input. As there are no output parameters, the function must return a Boolean, which is exactly what the `__gt__` function of the Python module `operator` does:

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

For a more complex example, we can define a predicate `DictContents`, which has three parameters, a Python dictionary, a key in that dictionary, and the value for that key. The `items` method of a Python dictionary returns a list of key-value tuples. We can use this method directly to implement a predicate that enumerates all the keys and values in the dictionary. We use `pyPredicateSeq` to indicate that the function returns a list of different solutions. The fact that there are two output parameters means that each element of the solution list is a interpreted as a sequence containing a value for `$key` and a value for `$value`:

```
{defpredicate (DictContents $dict $key $value)
    imp: (pyPredicateSeq "+--" (pyMeth "items"))}
```

You can specify different functions to call for different modalities using `multiModal`. For example, if `identity` were a Python function defined as `def identity(x): return x` in module `module`, then we could define the following predicate which would return `$x`=1 for both `(equal 1 $x)` and `(equal $x 1)`:

```
{defpredicate (equal $x $y)
    imp: (multiModal
            (pyPredicate "+-" (pyMod "module" "identity"))
            (pyPredicate "-+" (pyMod "module" "identity")))}
```

Sometimes, the Python function needs access to the Python object corresponding to the agent. In this case, you can start the mode string with the capital letter "`A`", indicating that the agent object

should be passed to the function along with the input parameters. Thus a mode `"A+-+"` means that the Python function will be passed three arguments: the agent object, the predicate's first parameter, and the predicate's third parameter. Since the agent object is always the first argument, if you use a Python method as the implementation it will be a method on the agent object.

7.3 Function Declarations and Implementations

A function declaration is of the form:

{`deffunction (`*ID (VAR)**`)`
[`imp:` *TERM]*
[`doc:` *STRING]*
[`properties:` *TERMLIST]*}

The arguments of the function are specified by a sequence of *VAR*s. It is assumed that all the arguments passed to a function must be evaluable.

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

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

The `imp:` argument specifies how the function is implemented. There are three types of function implementations:

1. Structures (default) - the function construct/deconstructs a record-like structure, similar to the terms of Prolog.
2. Closure - the function is defined in terms of other functions via a function closure.
3. Code - a simple python function or java static method is used to evaluate or match a value.

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. Thus given `{deffunction (tree $left $right)}`, the term `(tree 1 2)` would build a structure with functor `tree` and arguments `1` and `2`. The term `(tree $x $y)` would match this structure, binding `$x`=`1` and `$y`=`2`.

The function implementation can also be specified as a function closure. For example, the following function declaration implements function `plus1` as a function closure.

```
{deffunction (plus1 $x) imp: {fun [$x] (+ 1 $x)}}
```

As with predicates, SPARK functions can be defined in terms of Python functions and Java methods. For a function, the mode consists entirely of `+`s:

```
{deffunction (plus $x $y)
  doc: "Return the sum of $x and $y."
  imp: (pyFunction "++" (pyMod "operator" "__add__"))}
```

Similar to `pyFunction`, `pyReversible` takes an additional argument that implements the inverse function and can be used for matching. In the following example we see that function `addOne` is implemented as Python method `add_one`, and the inverse of function `addOne` is implemented as Python method `subtract_one`. This allows us to use the SPARK function `addOne` in the form `(= $x (addOne 1))` and `(= 2 (addOne $y))`:

```
{deffunction (addOne $number)
  doc: "Add one to a number"
  imp: (pyReversible "+" (pyMod "spark.util.number" "add_one")
                         (pyMod "spark.util.number" "subtract_one"))}
```

The Python function for implementing the inverse of the SPARK function must take a single SPARK value as its input and identify (i) whether or not some values for the parameters of the SPARK function yield the given result value and if so (ii) what those values are. We use the same interpretation of the returned value of the Python function as would be used for a SPARK predicate taking the same number of output parameters

- If the SPARK function has no parameters, the inverse function should return a Boolean – whether or not the result of the function call matches the value.
- If the SPARK function has exactly one parameter, the inverse function should return the value for that parameter or `None` if there is no parameter for which the SPARK function returns the given result.
- If the SPARK function has multiple parameters, the inverse function should return a sequence of values for the parameters for which the SPARK function returns the given result, or `None` if there is no such sequence.

Thus, in the above example, `subtract_one` should return a simple value.

As with predicates, the mode string may start with "`A`", in which case the agent object is passed to the Python function as the first argument. When using `pyReversible`, the second, inverse Python function must accept two arguments, the agent and the object to match.

7.4 Action Declarations and Implementations

An action declaration is of the form:

{`defaction` (*ID (MVAR)**)
  [`imp:` *TERM]*

[`doc:` *STRING]*
[`features:` *TERMLIST]*
[`properties:` *TERMLIST]*
[`roles:` *TERMLIST]* }

The action declaration arguments are similar to the predicate declaration arguments, but also include *features* and *roles* that provide information for meta-level reasoning about actions and procedures. The `features:` argument provides a way of associating an action with a category of actions. The `roles:` argument associates meaning with the parameters of the action.

The `imp:` argument specifies how the action is implemented. There are three types of action implementations:

1. Procedural (default) - the action is performed by expansion into other tasks via procedures.
2. Closure - the action is defined in terms of other tasks via a task closure.
3. Code - a simple python function or java static method is used to test the perform the action.

If no implementation is specified, the action is expanded using procedures that are cued on the action. If a task closure is specified, then it will be executed when the action is performed.

Actions can also be implemented using Python functions and Java methods. These functions should execute quickly since all other intentions stop running during its execution. As an example, the SPARK action `print` is declared below as a Python method `prin` defined in the `spark_example` Python module as `def prin(format, args): print format%args`:

```
{defaction (print $format $values)
  doc: "Formatted print - $format is a format string and
                          $values is a list of values."
  imp: (pyAction "++" (pyMod "spark_example" "prin"))}
```

Exceptions in Python are translated into failure in SPARK. For one or more output parameters, the same interpretation of the function result is used as for predicates – a single value for one output parameter and a sequence for multiple output parameters, with `None` being interpreted as failure. For the zero output parameter case, the natural Python equivalent of a SPARK action is a function that has no explicit return value. In Python, `None` is returned in this case, so we should not use `None` to represent failure. Instead, failure can be signaled by raising an exception.

As with predicates, the mode string may start with "`A`", in which case the agent object is passed to the Python function as the first argument.

7.5 Procedures Declarations and Implementations

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

{`defprocedure` *ID*
`cue:` *CUE*
[`precondition:` *LOG]*
`body:` *TASK*
[`doc:` *STRING]*
[`features:` *TERMLIST]*
[`properties:` *TERMLIST]*
[`roles:` *TERMLIST]*}

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

- `[newfact: (`*IDARG*`)LIST]` – instances of this procedures should be invoked (in new intentions) if a fact matching `(`*ID*`[`*(ARG)**`])` is added to the knowledge base.
- `[synchronous: (`*ID*`[`*(ARG)**`])]` – instances of this procedures should be invoked (pausing execution of the triggering intention) if a fact of the form `(`*ID*`[`*(ARG)**`])` is added to the knowledge base. This is only useful for meta-level predicate events (see Figure 5).
- `[do: (`*ID*`[`*(ARG)**`])]` – this procedure shows one possible way of decomposing the action `(`*ID*`[`*(ARG)**`])`.
- `[achieve: (`*ID*`[`*(ARG)**`])]` – this procedure shows one possible way of making `(`*ID*`[`*(ARG)**`])` true if it is not already true.

Each *ARG* denotes a formal argument to match against the parameters of the given fact or task and can be one of the following forms:

- *VAR* – a variable that may or may not be bound to a specific value at run time
- `+`*TERM* – *TERM* is a term expression to match the corresponding actual parameter which must be evaluable
- `-`*TERM* `-` *TERM* is a term expression that will be evaluable after executing the body of the procedure and the corresponding actual parameter will be matched to the value that results from evaluating this expression.

These *mode annotations* on the formal parameters let SPARK know which variables in the body of the procedure will always be bound at run time. This is vital for knowing if the expressions within the body satisfy the modality specified in the declarations of the predicates and actions.

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. For example:

```
{defprocedure Choose_Item_Minimum
  cue: [do: (chooseItem $quotes $selection)]
  precondition: (True) # not strictly necessary, this is the default
  body: [do: (chooseMinimumCost $quotes $selection)]
  features: [(mode "automated") (minimize "cost")]
  roles: [(options $quotes) (choice $selection)]
}
```

8 Meta-Level Reasoning

The SPARK meta-level reasoning provides mechanisms for modifying the execution of SPARK, including modifying the intention state of SPARK as well as influencing procedure selection. For example, when SPARK is faced with a choice of procedures instances applicable for a given goal it must select one; by default, an arbitrary procedure is selected. You can use meta-level procedures to provide a different mechanism for selection using your own specified criteria.

In order to understand the meta-level reasoning mechanisms, it is important to understand how SPARK represents its intention state (*intention structure*, *tframes*, and *events*), as well as the SPARK execution cycle. Both of these are described below in more detail. We will also discuss meta-level predicates and meta-level procedures.

8.1 Intention Structure, Tframes and Events

The *intention structure* of a SPARK agent consists of forest of trees of *tframes*. A tframe is some executing process, be it a procedure instance or a task closure instance, or an action implementation, that can be executed incrementally. Each tframe has a triggering *event* that caused it to be intended. Thus, for procedure instance tframes, the triggering event is what the cue matched.

Procedures can be either synchronous or asynchronous. Synchronous procedures are triggered by the adoption of tasks such as `[do: (foo)]`, or adoptions of goals such as `[achieve: (bar)]`). Changes in the agent's knowledge base generally trigger asynchronous (`newfact:`) procedures. However, some meta-level events that reflect changes in the intention structure can also have synchronous (`synchronous:`) meta-level procedures respond to them, allowing the meta-level level procedure to act before the triggering intention continues execution.

Some of the events triggering synchronous procedures are classified as *solver events*. Such tasks are expected either to complete successfully and bind variables in the parent tframe or to fail and return some reason for failure. Generally there are multiple tframes applicable for a solver event, and SPARK commits to one at any given time. In contrast, when there are multiple tframes applicable for an asynchronous event, SPARK intends all of the applicable tframes at once.

Tframes with asynchronous triggering events, such as would be matched by `cue: [newfact: (P)]`, appear as the roots of the trees. Tframes with synchronous triggering events, such as would be matched by `cue: [do: (foo)]`, are mostly caused by the execution of a *parent* tframe, and appear in the intention structure as children of their triggering event's parent tframe. Execution of the parent tframe is suspended until all the children have completed and have been removed from the intention structure.

8.2 SPARK Inner Execution Cycle

SPARK agent architecture is both goal directed and event driven. To accomplish this, SPARK maintains an intention structure and an event list to be responded to. During each inner execution cycle, SPARK processes one leaf tframe of the intention structure as well as responds to all events in the event list.

In addition, SPARK employs a multi-level meta-reasoning mechanism where higher level meta-procedures can overwrite the normal execution cycles of lower level procedures. In the course of this multi-level meta-reasoning, SPARK generates meta-level SOAPI events corresponding to new facts such as `(SOAPI $event $tframes)`, where `$tframes` represent the set of applicable procedure instances for the triggering event `$event`. SOAPI events are only used within multi-level meta-reasoning, and are not inserted to the event list.

Figure 4 illustrates the details of SPARK inner execution cycle. In steps 1 through 3, SPARK processes one leaf tframe in the intention structure, possibly adding events to the event list. In step 4, SPARK processes all the events in the event list. For each event, SPARK uses a bottom-up approach to find all applicable meta-level procedures. In step 4(c)i, SPARK looks for applicable procedure instances (or more generally, tframes) for event $E[level]$, and the total number of applicable tframes are recored in variable $N[level]$ in step 4(c)ii. In step 4(c)iii, if SPARK has found some applicable tframes at this level, SPARK then generates a meta-level SOAPI event $E[level+1]$ which corresponds to all applicable tframes for the event $E[level]$. At the next level of meta-reasoning, SPARK continues to look for applicable procedure instances for the newly generated SOAPI event. This process continues until SPARK can't find any more applicable meta-level procedures at two consecutive levels (step 4(c)iv) or it exceeds the allowed maximum level of meta-level reasoning. If SPARK has found some applicable tframes at any meta-level in the previous step, then in step 4d, SPARK chooses one tframe, preferring higher meta-level tframes over lower meta-level and object level tframes, and then intends this tframe. Otherwise, if SPARK can not find any applicable tframes and the event e being processed is a solver event, then in step 4e, SPARK must inform the parent of e of failure to find a procedure.

An object-level tframe corresponds to a procedure instance that responds to a new belief in SPARK knowledge base, while a meta-level tframe is typically triggered by SOAPI events, and it ensures that some lower-level tframes are intended according to some criteria, thus modifying the intention state of SPARK. For example, if the selected event e in step 4a is an asynchronous event, a *built-in default* meta-level procedure for asynchronous events will get intended, and this meta-lvel procedure ensures that *all* applicable tframes to asynchronous event e are intended.

Note that events may be added to the event list in steps 4d and 4e, and the newly added events are processed within the same execution cycle inside the while loop in step 4. This may result in infinite metal-level recursion, and we need to be cautious when we write meta-level procedures to avoid infinite recursion.

1. Pick a leaf tframe, t, in the intention structure
2. If t has completed, remove t from the intention structure and (if triggered by a solver event) inform the parent tframe of the success or failure of t
3. Else perform one step of t
4. While there are events in the event list:
   (a) Select an event, e, from the event list to process
   (b) $E[1] \leftarrow e$
   (c) For $level$ from 1 to maximumLevelOfMetaReasoning, do:
      i. $tflist[level] \leftarrow$ list of tframes applicable to $E[level]$
      ii. $N[level] \leftarrow length(tflist[level])$
      iii. If $N[level] \neq 0$, $E[level+1] \leftarrow event(\text{SOAPI } E[level]\ tflist)$
      iv. Else if $level > 1$ and $N[level-1] = 0$ then break
   (d) If $level > 2$, then add an arbitrary element of $tflist[level-2]$ as a new leaf of the intention structure
   (e) Else (i.e. $level \leq 2$) if e is a solver event, then inform the parent of e of failure to find a procedure

Figure 4: SPARK Inner Execution

8.3 Meta-level Events and Predicates

In addition to the events generated when concluding new facts into the knowledge base, execution of SPARK procedures also generates internal "meta-level" events. These meta-level events can be used to trigger "meta-level" procedures that can perform additional tasks or change the execution. For uniformity, these events are treated as if they were assertions of facts into the knowledge base, except that those facts then vanish and are unable to be tested[7] once the current execution cycle completes. The significant meta-level events are represented by predicates in Figure 5.

8.4 Meta-level Procedures

Meta-level procedures can be used for many purposes. For example, they can help choose which procedure to use if there are multiple applicable procedure instances, or they can assist in performing event logging, or for failure handling. We present a few examples of meta-level procedures, but SPARK's meta-level procedures are by no means limited to these functionalities.

[7] We call these *ephemeral* predicates

- `(SOAPI $event $tframes)` - indicates that `$tframes` is the list of procedure instances that were found to match the triggering event `$event`. Meta-level procedures triggering on this can be used to handle getting multiple or getting no applicable procedure instances in a way that differs from the default mechanism.
- `(AppliedAdvice $task $advice_label)` - indicate that advice with label `$advice_label` was used in the selection of procedures for `$task`.
- `(AdoptedTask $task)` - indicates that task `$task` has been adopted.
- `(CompletedTask $task)` - indicates that task `$task` has successfully completed.
- `(FailedTask $task $reason)` - indicates that task `$task` has failed with reason `$reason`.
- `(StartedProcedure $tframe)` - indicates that procedure instance `$tframe` has been intended.
- `(CompletedProcedure $tframe)` - indicates that the procedure instance has completed successfully.
- `(FailedProcedure $tframe $reason)` - indicates that the procedure instance has failed with reason `$reason`.

Figure 5: SPARK Metal-Level Events and Predicates

8.4.1 Meta-level Procedures for Selecting Among Multiple Applicable Procedures Instances

Consider the following meta-level procedure. If an event satisfies `EventIsSolver`, it means that the event requires a procedure to return bindings and indicate success or failure. If we have multiple applicable procedure instances, the default behavior is to select and intend only one procedure instance. The choice is arbitrary. This new meta-level procedure will override that default behavior if one of the procedure instances satisfies the predicate `MyPreferredTframe` by intending one of those preferred procedure instances. Note that we use a `synchronous:` cue rather than a `newfact:` cue, since we want to suspend all processing of the tframe that triggered the `SOAPI` event until the meta-level procedure has worked out what new tframe to intend, intended it, and completed:

```
{defprocedure MyMetaProcedure
  cue: [synchronous: (SOAPI $event $tframes)]
  precondition: (and (EventIsSolver $event)
                     (> (length $tframes) 1)
                     (Member $tf $tframes)
                     (MyPreferredTframe $tf))
  body: [do: (intendTFrame $tf)]}
```

As a second example, we have defined two features `normal_meta` and `preferred_meta` through `deffunction`, and have also written two meta-level procedures using these features (NormalMeta

and PreferredMeta). The third meta-procedure `MetaIntendPreferredMeta` shown is a higher-level meta-procedure that intends lower-level meta-procedures – upon SOAPI events, among available tframes, it selects to intend a meta-procedure that has the `preferred_meta` feature instead of other meta-procedures.

```
# features
{deffunction (normal_meta)}
{deffunction (preferred_meta)}

{defprocedure NormalMeta
  cue: [synchronous: (SOAPI $event $tframes)]
  precondition: (and (not (EventIsSolver $event))
                     (> (length $tframes) 1)
                     (Member $tf $tframes)
                     (MyPreferredTframe_normal $tf))
  features: [(preferred_meta)]
  body: [do: (intendTFrame $tf)]}

{defprocedure PreferredMeta
 cue: [synchronous: (SOAPI $event $tframes)]
 precondition: (and (not (EventIsSolver $event))
                    (> (length $tframes) 1)
                    (Member $tf $tframes)))
 features: [(normal_meta)]
 body: [forin: $tf $tframes [do: (intendTFrame $tf)]]
 }

{defprocedure MetaIntendPreferredMeta
  cue: [synchronous: (SOAPI $event $tframes)]
  precondition: (and (EventIsSolver $event)
                     (> (length $tframes) 1)
                     (Member $tframe $tframes)
                     (= $features (features $tframe))
                     (Member (preferred_meta) $features)
                     )
  body: [do: (intendTFrame $tframe)]
}
```

8.4.2 Meta-level Procedures to Assist In Event Logging

Meta-level procedures are often used for event logging. The following procedure will conclude a fact in the knowledge base each time a task is adopted. The `Objective` predicate has two functions. Firstly it stops the procedure from responding to tasks in meta-level procedures (useful in avoiding infinite meta-level recursion) and secondly it finds the top level objective associated with `$task`:

```
{defprocedure Log_Adopted_Task_Event
  cue: [newfact: (AdoptedTask $task)]
  precondition: (and (Objective $task $objective)
                     (= $time (currentTime)))
  body: [conclude: (TaskWasAdopted $task $objective $time)]
}
```

Another example of event logging shown below is triggered when a `$task` has completed. In the precondition of this meta-procedure, we first test if this task is an objective level task, and then check if this task/goal has property `avgExecTime` associated with it. If so, we then concludes a fact in the knowledge base indicating the completion of this task. Predicate `ObjectId` finds the unique id associated with this task instance.

```
{defprocedure Note_success_of_task
 cue: [newfact: (CompletedTask $task)]
 precondition:  (and (Objective $task $obj)
                     (= $goalSymbol (getGoalName $task))
                     (Properties $goalSymbol $props)
                     (= $time (getCurrentTime))
                     (Member (avgExecTime $t) $props))
 body: [seq:
        [context: (and (= (@ $goalnamestring) $goalSymbol)
                       (= $bindings (getBindings $task)
                       (ObjectId $task $id))]
        [conclude: (TaskSucceeded $id $goalnamestring $bindings $time)]]
 }
```

8.4.3 Meta-level Procedures For Failure Handling

The following example of meta-level procedure is used for failure handling. It is triggered when a procedure instance `$tframe` fails for some reason `$reason`. This procedure finds the task that triggered this tframe using predicate `(TaskContext $tframe $taskexpr)`, and retrieves the goal name (`getGoalName`) and arguments to the goal `(getBindings)`. Here we have to use a `synchronous:` cue, to make sure that the normal handling of the failed procedure is suspended while the meta-level procedure is running. In the body of the procedure, different recovery actions are taken depending on the failure reason.

```
{defprocedure FailureHandling
 cue: [synchronous: (FailedProcedure $tframe $reason)]
 precondition: (and (TaskContext $tframe $taskexpr)
                    (= (@ $goalnamestring) (getGoalName $taskexpr))
                    (= $bindings (getBindings $taskexpr)))
 body:
  [select: (FailValue $reason (process_failure "failure type one"))
```

```
            [do: (recoverActionOne $goalnamestring $bindings)]
            (FailValue $reason (process_failure "failure type two"))
            [do: (recoverActionTwo $goalnamestring $bindings)]]]
}
```

In summary, meta-level procedures are triggered by new meta-predicates corresponding to events in Figure 5. In the preconditions and bodies of meta-level procedures, a variety of predicates and functions are used to access different aspects of the internal state of SPARK, as well as properties, features, and roles of actions and other procedures. A list of such SPARK predefined predicates and functions are listed online at `http://www.ai.sri.com/ spark/doc/sparkdoc/index.html`.

9 Advice

When SPARK is faced with a choice of procedures instances applicable for a given goal it must select one. In the absence of any meta-level procedures, an arbitrary procedure is selected. You can use meta-level procedures to provide a different mechanism for selecting between different applicable procedure instances.

Another mechanism SPARK provides for choosing among multiple procedures instances is the *advice* mechanism. Advices allow a user to specify "preferences" in the selection of procedure instances, and can also be used to invoke other "consultation" procedures to determine which procedures instance to use. Advice and consultation are implemented as special types of meta-level procedures in SPARK. This section describe SPARK advice mechanisms.

9.1 Using Advice in SPARK

The advice module, `spark.lang.advice`, allows SPARK to select between procedure instances based on advice declarations. These advice declarations are of the form:

{`defadvice` *ID TERM VAR LOG VAR LOG*}

where *ID* is the name of the advice; *TERM* is one of the constants `strong_prefer`, `strong_avoid`, `prefer`, `avoid`, `weak_prefer`, or `weak_avoid`, indicating the strength of the advice and whether the situation is prefered or to be avoided; the first *VAR* is bound to the goal/task currently being expanded; the first *LOG* is a condition on that goal/task; the second *VAR* is bound to the procedure instance under consideration; and finally, the second *LOG* is a condition on both the goal and the procedure instance.

In general, one must do four things to use advice:

1. Declare the relevant features using `deffunction`. SPARK has a set of predefined features including `(selection)`, `(task_type $type)`, etc, which can be found in SPARK module task_manager.advice_metatheory).
2. Declare the existence of these features in relevant actions and/or procedures.
3. Define advice using `defadvice`.
4. Add fact `(AdviceActive` *STRING*), where *STRING* is the name of the advice. This is because advice declarations can exist in the system without being currently active, and advices are not active by default. So to make an advice active, one needs to assert the fact `(AdviceActive $advicename)`.

Figure 6 illustrates how to define and use advice in SPARK. First, notice that we are using SPARK predefined set of features including `(task_type $type)` and `(mode $mode)` in action `(resolve_CSP)`, and in procedures `AutomatedCSP` and `InteractiveCSP`. We then define two advices

for how to select between the two procedures. The first advice "`AutomatedSelectionsScheduling`" can be read as: if the goal/task (`$task`) currently being exapnded has feature that its task_type is "scheduling", then among all applicable procedure instances, prefer the one with feature indicating "automated" mode. The second advice is similar, except that it prefers the procedure with "interactive" mode. The above two advices can be used to select between two procedures AutomatedCSP and InteractiveCSP for the same goal `resolve_CSP`.

For example, we need to assert `(AdviceActive "AutomatedSelectionsScheduling")` or `(AdviceActive "InteractiveSelectionsScheduling")` in the knowledge base respectively to active the corresponding advices. Notice that these two advices conflict with each other, but by concluding different `AdviceActive`, we can easily switch between the two advices. In contrast to using meta-procedures to influence selections of procedures, using advices have the advantage that it allows user to turn advices on/off easily.

When SPARK is faced with more than one applicable procedure instance, each procedure instance is matched against the active advice rules. For each advice rule that matches, the procedure instance gains a weighting – positive if the rule is a prefer rule and negative if it is an avoid rule. `strong` preferences have large weightings that override all ordinary preferences, which in turn override all `weak` preferences. The procedure instances with the highest sum of weights is used. This is called the *maximally-preferred* procedure instance.

If there are multiple maximally preferred procedure instance, the default behavior is to choose one of these arbitrarily. However, this can be overridden with consultation advice.

9.2 Consultation

Consultation advice is supplied using "defconsult" declarations of the following form:

{`defconsult` *STRING VAR VAR LOG* `doc:` *STRING*}

The *STRING* is a name associated with the advice. The *VAR*s are variables to be bound to the task to be performed and a list containing the maximally preferred procedure instances respectively, and *LOG* is a condition on these variables. Consultation advice is activated by asserting the fact `(ConsultActive` *STRING*`)`, where *STRING* is the name of the consultation advice.

If at the end of the preference advice handling, there are multiple maximally preferred procedure instances, the consultation advice is tested. If the *LOG* of at least one of the active consultation advice tests true, then a goal `[do: (consultUser $advice $origtflist $tflist $tf]` is posted. `$advice` is bound to the list of applicable active consultation advice names, `$tflist` is bound to the maximally preferred procedure instances, `$origtflist` is bound to the list of *all* applicable procedure instances including those not maximally preferred. The result of this action should be to bind `$tf` to a single procedure instance that is the one to use.

The following examples defines a consultation called `"always"`, and it is actived. So whenever there are multiple maximally preferred procedure instances, consultation procedure

```
{defaction (resolve_CSP $rid $status $result)
 features: [(task_type "scheduling")]}

{defprocedure AutomatedCSP
 cue: [do: (resolve_CSP $rid $status $result)]
 precondition: (True)
 body: [...]
 features: [(mode "automated")]
}

{defprocedure InteractiveCSP
 cue: [do: (resolve_CSP $rid $status $result)]
 precondition: (True)
 body: [...]
 features: [(mode "interactive")]
}

{defadvice "AutomatedSelectionsScheduling" prefer
  $task (ContextFeature $task (task_type "scheduling"))
  $tf (Feature $tf (mode "automated"))
  doc: "Let the system make selection decisions for scheduling."
}

{defadvice "InteractiveSelectionsScheduling" prefer
  $task (ContextFeature $task (task_type "scheduling"))
  $tf (Feature $tf (mode "interactive"))
  doc: "Let the user make selection decisions for scheduling."
}

# Conclude one of the fact.
#(AdviceActive "AutomatedSelectionsScheduling")
 (AdviceActive "InteractiveSelectionsScheduling")
```

Figure 6: Using Advice in SPARK

`Trivial_consultUser` is activated to consult user in selecting a single procedure instance.

```
{defconsult "always" $event $tframes (and (Ground $event) (Ground $tframes))}

(ConsultActive "always")

{defprocedure Trivial_consultUser
  cue: [do: (consultUser $clabels $event $orig_tframes $pref_tframes
            $selected_tframe)]
  precondition: (Ground $orig_tframes)
  body: [seq:
         [do: (print "Consulting user about %s due to rules: %s"
                     [$event $clabels])]
         [do: (userSelect $selected_tframe $pref_tframes)]]}
```

10 Acknowledgements

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

References

[1] 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.

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

[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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] A. van Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Copyright