A Portable Process Language

Peter E. Clark[1], David Morley[2], Vinay K. Chaudhri[2], Karen L. Myers[2]

[1]M&CT, Boeing Phantom Works, PO Box 3707, Seattle, WA 98124
[2]Artificial Intelligence Center, SRI International, Menlo Park, CA 94025
peter.e.clark@boeing.com, {morley,chaudhri,myers}@ai.sri.com

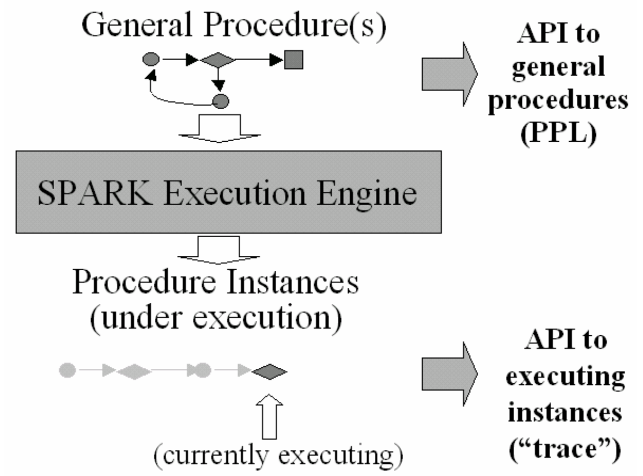
Abstract

Process representation languages designed to support execution have evolved to support specialized reasoning capabilities like action selection and task decomposition, but do not readily support inferences that one might need for explanation or question answering. In this paper, we report on a process language, PPL, that we have designed to serve as a bridge between a representation designed for execution and a representation designed for applications such as question answering and explanation generation. Through its use of a propositional-style representation of process structure, PPL can enable the use of generalized reasoning methods for those purposes. PPL is novel in that it directly encodes the process "flow chart" in a neutral, KIF-like syntax, allowing other modules to introspect on the process structure.

Introduction

SPARK (SRI Procedural Agent Realization Toolkit) [Morley and Myers 04] is an agent framework that builds on a Belief-Desire-Intention (BDI) model of rationality. SPARK provides a flexible plan execution mechanism that interleaves goal-directed activity and reactivity to changes in its execution environment. SPARK's procedural language has a clear, well-defined formal semantics that can support reasoning techniques for procedure validation, synthesis, and repair. The SPARK representation language (called SPARK-L) is an extended form of Hierarchical Task Network (HTN) representation [Erol et al. 04]. Extensions beyond standard HTN include iteration, conditional branching, and certain runtime-specialized constructs (WAIT, TRY).

SPARK includes a comprehensive API for monitoring executing procedures—called *procedure instances*—allowing external modules to view the "trace" of a procedure's execution. For example, by using this API an external module can find the specific task(s) currently being executed, and trace back to find specific previously executed tasks and their details. However, while allowing access to how a general procedure is playing out, this API does not allow access to the general "flow chart" of the procedure itself, for example, to find possible future tasks, choice points, cycles, and alternative ways the procedure may play out. However, many tasks require that a system be able to introspect on its own general procedures, in particular language understanding, question answering, and explanation. This has been particularly true in the context of CALO, an integrated cognitive assistant.[1] As a result, we have enhanced SPARK with a second API allowing it to export (an abstraction of) its general procedures also. These exported procedures are expressed in logic in Portable Process Language (PPL), a language we have designed for this purpose, complementing SPARK-L, the native process language of SPARK. The goals of PPL are to make explicit the steps, their arguments, and their relationships in the procedures, and to abstract away some of the details critical for execution but not needed for introspection.


General Procedure(s)
API to general procedures (PPL)
SPARK Execution Engine
Procedure Instances (under execution)
API to executing instances ("trace")
(currently executing)

As a close analogy, consider a system wanting to introspect on a piece of software, but only having an API to the trace of that software executing. The system could, of course, literally try to parse the source code representation of the software and try to work out what it did (in fact, some software analysis tools do exactly this), but this requires the system to have a complete internal model of the programming language, both syntax and semantics. Instead, it would be nice if the software could export a high-level summary of its behavior—literally, a flow-chart-like data structure—that other systems could

[1] See http://www.ai.sri.com/project/CALO

then manipulate, expressed in some relatively implementation-independent language. Our goal is that PPL be such a language, in the specific context of SPARK-like planning systems.

In this paper, we describe a simplified version of the SPARK language and how it is executed, the reasoning requirements, PPL, and experiences in using the language in the context of an agent system. We conclude with directions for future work.

Description of SPARK Process Language

The SPARK process language provides a hierarchical representation of processes. A library of *procedures* provides declarative representations of activities for responding to events and for decomposing complex tasks into simpler tasks. Each procedure has a *precondition* stating conditions under which it can be applied, and a *task network expression* describing how to respond to an event or to decompose a non-primitive action. The hierarchical decomposition of tasks bottoms out in primitive actions that bring about some change in the outside world or the internal state of the agent. A SPARK agent specification includes declarations of predicate and task symbols, facts about the initial state, and a library of procedures. The key syntactic structures include term expressions, logical expressions, actions, task expressions, and procedures.

A *term expression* represents a value. Atomic term expressions are constants such as `42` and `"Hi"`, and variables of the form `$x`. These are combined to form compound term expressions, including lists such as `[1 2 3]`.

Logical expressions are constructed from predicate (fluent) symbols applied to term expressions, e.g., `(= 1 $x)`, `(True)`, logical operators, and quantifiers. These expressions can be used to alter the flow of execution or to bind variables for use later in the procedure.

An *action* is constructed from an action symbol and term expressions as parameters, for example, `(laptop_query $criteria $quotes)`. The action may be primitive, which is performed by executing some procedural attachment, or nonprimitive, which is hierarchically expanded into subtasks using procedures. The parameters of an action may include free variables that are not bound at the time the action is attempted. These variables are bound by the successful execution of the action and provide a means of returning values from executing the action.

A *task network expression* is a pattern of activity that when attempted may either succeed or fail. Task network expressions include such constructs as

`[set:` *V T*`]` - Set variable *V* to the value *T*.
`[do:` *A* `]` – Perform action *A*.
`[seq:` *N1 N2* … `]` – Attempt task networks *N1*, *N2*, … sequentially.
`[parallel:` *N1 N2* … `]` – Attempt task networks *N1*, *N2*, … in parallel.
`[select:` *P1 N1 P2 N2* … `]` – Execute the task expression *Ni* corresponding to the first logical expression *Pi* that has a solution. Fail if none has a solution.
`[wait:` *P1 N1 P2 N2* … `]` – As select, but wait instead of failing.
`[while:` *P N*`]` – While *P* has a solution execute *N*.

At its simplest, a SPARK *procedure* has the form `{defprocedure` *name* `cue:` *trigger* `precondition:` *P* `body:` *N*`}`

This indicates that if *P* is true when *trigger* occurs, then executing *N* is a valid way of responding. The cue may be of the form `[newfact:` *P*`]` to respond to the fact *P* being added to the KB, or `[do:` *A*`]` to expand the action *A*.

For example, the following procedure, `Get_Bid`, describes one way of expanding the task of performing action find_bids. It is applicable if the condition `(Online "DM4QR")` holds. It performs a `laptop_query` action that binds variable `$temp_quotes` and then depending upon whether or not this list is empty, either performs `relax_and_redo_query`, binding `$quotes`, or binds `$quotes` to `$temp_quotes`.

```
{defprocedure Get_Bid
  cue:
   [do: (find_bids $item $criteria $quotes)]
  precondition: (Online "DM4QR")
  body:
  [seq:
    [do: (laptop_query $criteria
                             $temp_quotes)]
    [select:
      (=  $temp_quotes [])
        [do: (relax_and_redo_query $criteria
                                   $quotes)]
      (True)
        [set: $quotes $temp_quotes]]]}
```

SPARK Process Execution

Figure 1 shows the architecture of each SPARK agent. Each agent is embedded in the world and interacts with the world though sensors and effectors. Each agent has its own knowledge base of beliefs and library of procedures. The initial state of the beliefs and procedure library are initially loaded from files written in the SPARK language. The beliefs are updated by the agent's sensors and through the agent executing procedures. The set of procedure instances that the agent is executing at a given time form the intentions of the agent. At any time an agent may be executing multiple intentions. At SPARK's core is the executor whose role is to manage the execution of intentions. The executor can post procedure subtasks as actions. Primitive actions cause effects through the effectors. Non-primitive actions are expanded by the executor according to the procedures the agent has available.

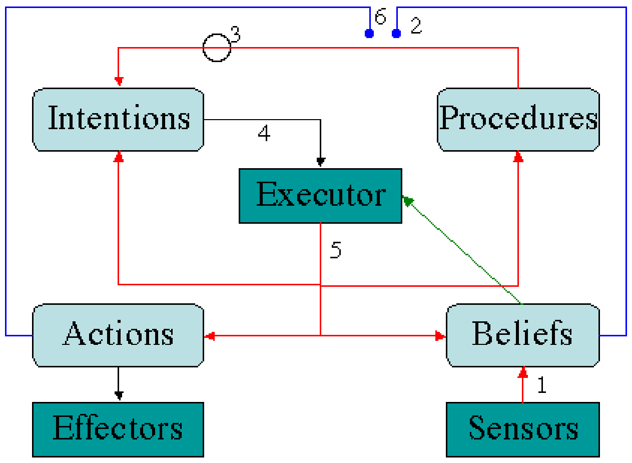


Figure 1: The Architecture of each SPARK agent.

SPARK process execution proceeds as follows: (1) Sensors update the agents beliefs. (2) These belief changes trigger the instantiation of procedures - those with a cue that matches the event and a precondition that is satisfied with respect to the agent's beliefs. (3) A subset of these applicable procedures is selected to be added to the intentions. (4) The executor selects one intended procedure instance to progress by executing a single step. (5) This may cause updates to the beliefs or the posting of new subtasks. (6,2) This in turn triggers the instantiation of procedures, and so on.

The step by step execution of each procedure instance results in the binding of values to variables in the procedure instance. Each predicate that is tested or action that is executed results in all free variables becoming bound. Usually when a logical expression (such as may appear in a context: or select: task network expression) is tested, SPARK needs to commit to a single solution without the possibility of backtracking.

The approach that SPARK and many other BDI agent systems take to the hierarchical expansion of tasks is based on an expectation that the agent is working in a highly dynamic environment: the expansion of tasks is performed *at the time the task is to be executed* and the choice of the procedure to use is based on the *current state of the agents beliefs*. BDI agent systems have been integrated with planning systems (e.g., [Myers 99, Lemai and Ingrande 04]) however SPARK currently has no planning component.

Reasoning Requirements

We consider here the reasoning requirements for three applications, and how introspection on the process model itself is necessary to support them: (1) dialog interpretation, (2) question answering, (3) explanation generation.

Reasoning Requirements for Dialog Interpretation

Robust language processing is challenging in that what the user says can be ambiguous, incomplete, and erroneous. In the context of the CALO system that we are developing, one class of dialogs (between the user and computer) that we have been studying is "purchase dialogs", where the user is directing CALO to purchase an item on the user's behalf. In these dialogs, these problems can all occur, and resolution of them requires the system to have strong expectations, hence requiring knowledge of the processes themselves.

For example, while instructing CALO to buy a computer, if the user says "Find me an appropriate machine", the user actually means "You have authorization to start retrieving quotes from vendors for the laptop I want to purchase" (as opposed to, for example, start physically searching the building to find machines). To find the correct interpretation, CALO needs to have strong expectations about what sort of activities may occur in the future, and then match the user's utterance with those expectations. In this case, CALO contains a process model of how to purchase items, and as this process model is currently active (triggered by earlier statements by the user, such as "I want to purchase a laptop"), CALO should be able to look at future steps that could be construed as "finding a machine" to identify what the user is referring to. In this case, a subsequent step in the process is to retrieve quotes for the to-be-purchased computer, and a matching algorithm can identify this as the most likely thing to which the user is referring [Yeh et al. 03]. To do this, the system needs to introspect on its general purchase plan (procedure), to identify that a future step that can be construed as "finding" will occur.

Reasoning Requirements for Question Answering

As a general-purpose assistant, CALO is expected to field answers to a wide variety of questions from a user, including about its (CALO's) own knowledge of how to do things. While SPARK's execution trace API allows CALO to answer questions about specific things it has done (e.g., "When you purchased the laptop, did you request authorization?"), CALO also needs knowledge of the procedural flow chart itself for more general questions about procedures, for example,

1. How do you purchase a laptop?

2. Will you need to access to the Web during the purchase?
3. Is authorization required [i.e., is there an authorization step] to purchase a laptop over $2000?
4. What will happen if the authorizing manager is unavailable?
5. Email the quotes you find to my home email address.
6. Get authorization from Joe, not Steve.

The first four of these questions are independent of any specific execution of the procedure, and necessarily require access to the general procedure itself to answer the questions. The last two questions are in the context of a partially executed procedure, in which the user is making reference to a to-be-executed future step. Again with these two questions, the system needs a representation of the general procedure to identify the future step to which the user is referring (these are not in the execution trace, as they have not yet happened).

Requirements for Supporting Explanation

In a related piece of work reported elsewhere, we are working on explaining processes. Answering "why" questions particularly requires introspecting not just on what happened, but the specific tests and conditions that caused those things to happen, a particular form of meta-reasoning. Again, knowledge of the structure of the process flow chart is often required to answer these questions, including details of tests directing flow of execution at branch points, and details of how earlier steps support later steps. (In its current form PPL does not capture all this knowledge, but it is a step toward this.) Example questions from the user to CALO include

1. Why didn't you ask for authorization?
2. Why did you send the purchase request to Dallas?
3. Why haven't you started searching yet?

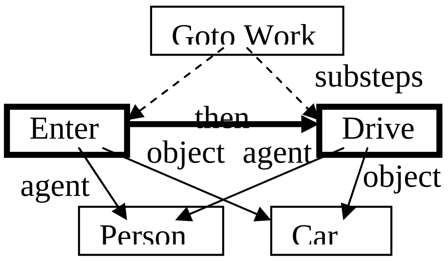
Portable Process Language (PPL)

Overview

To meet the reasoning requirements already identified, we must be able to introspect SPARK processes to obtain information as follows: What tasks are involved in task X? What task comes after task Y? Might task Z occur in process W? The goal of the Portable Process Language (PPL) is to represent just this kind of "flow chart" information. PPL captures a filtered view of the SPARK process models to include tasks, subtasks, task parameters, and task ordering. It does not currently capture the semantics of conditional tests, context, and other logical evaluations. Such extensions can be added in the future.

We now describe PPL, and then describe how it complements PSL (Process Specification Language), a language that captures the semantics but not the explicit structure of a process flow chart.

Consider a toy example of a two-step process—namely, going to work—with a trivial two-step flow chart consisting of (i) entering a car and then (ii) driving the car to work:



The PPL for a skolem instance process is follows:

```
;;; Steps in the flow chart
(instance-of goto-work1 goto-work-step)
(instance-of enter1 enter-step)
(instance-of drive1 drive-step)

(possibleTask goto-work1 enter1)
(possibleTask goto-work1 drive1)
(followedBy enter1 drive1)

;;; Parameters to those steps
(instance-of person1 person-var)
(instance-of car1 car-var)

(agent goto-work1 person1)
(object goto-work1 car1)

(agent enter1 person1)
(object enter1 car1)

(agent enter1 person1)
(object drive1 car1)
```

In PPL, each of the steps in the flow chart is denoted by a *specific individual*. Note that these individuals are *not* instances of events in the world (e.g., the specific event of entering the car at a certain time); rather, they are instances of steps in a flow chart. Similarly, each parameter (e.g., the person, the car) for a flow chart step is denoted by a *specific individual*. Again, note that these individuals are *not* instances of objects in the world (e.g., a specific person or car); rather, they are instances of parameters (variables) in the flow chart. In other words, PPL is a somewhat literal logical depiction of flow chart steps, rather than of event sequences. We can relate flow chart steps to event types (classes), and flow chart parameters to object types (classes), with some simple correspondence statements:

```
(event-type-for goto-work-step goto-work)
(event-type-for enter-step       enter)
(event-type-for drive-step       drive)
(object-type-for person-var      person)
(object-type-for car-var         car)
```

where goto-work, enter, drive are types (classes) of actual events in the world, and person, car are types (classes) of actual objects in the world.

Given a PPL flow chart, in principle one could "execute" it to perform the plan it denotes. However, the AI planning community has long matured beyond these kinds of "toy" executions—in the real world, plan execution also involves plan monitoring, recovery in the case of failure, consideration of time and resource constraints, and so on. To adequately represent such executable processes, much richer languages are needed, and this is precisely the role of SPARK. Rather, one should view PPL as capturing a simple abstraction of a complex executable process, in a language-neutral form suitable for introspection, to support the kinds of tasks discussed earlier.

For contrast, a simple SPARK representation of the above procedure would be

```
{defprocedure Goto_Work_by_Car
  cue: [do: (goto_work $person $car)]
  precondition: (True)
  body:
  [seq: [do: (enter $person $car)]
        [do: (drive $person $car)]]}
```

While suitable for execution, this syntactic structure is more difficult to manipulate for introspection. Note also that some information in this SPARK representation is implicit: PPL makes explicit the step ordering (implicit in the ordering of lines of SPARK-L representation) and step-substep relations (implicit in the grouping of tasks within a single procedure in SPARK-L). This allows other tools easy access to the process itself.

Conditionals

PPL denotes conditional branches in a flow chart using a predicate

```
(conditionalFollowedBy <step> <test> <next-step>)
```

meaning that if execution is at <step>, and <test> evaluates to true, then the next step will be <next-step>. At present, <test> is the opaque (quoted) logical expression copied from the SPARK-L test itself, but our plan is to replace this with a transparent logical expression, whose variables include the parameter instances in the rest of the PPL. For example, from the SPARK procedure for purchasing a laptop, the step "relax-and-redo" follows "laptop-query" only if no quotes were found. This appears in PPL as

```
(conditionalFollowedBy laptop-query1
    (and "(= $temp_quotes [])")
    relax-and-redo1)
```

This allows the external systems to see that relax-and-redo1 is a possible next step in the plan, but not at present to understand details of conditions under which it will be executed (unless it was to parse the quoted logical expressions).

Additional Representational Properties

While it is not our intention that PPL capture the full details of the original SPARK process models, it is clear that there are additional details that should be captured. These include preconditions, "cue" conditions, and better handling of logical assertions and tests in the original SPARK. These are all items for future work.

How PPL is Generated

The PPL is generated by introducing specific individual names for each variable, for the cue task, and for each atomic step, such as [do: A] in the procedure. SPARK action symbols, such as enter, become event types. Type and role declarations for the actions are translated into object type statements and role statements. Thus, for an action enter with parameter roles agent of type person and object of type car, we translate

```
[do: (enter $person $car)]
```

into

```
(instance-of enter1 enter-step)
(event-type-for enter-step enter)
(agent enter1 person1)
(instance-of person1 person-var)
(object-type-for person-var person)
(instance-of car1 car-var)
(object-type-for car-var car)
(object enter1 car1)
```

Each of the specific individuals corresponding to the atomic steps is related to the individual corresponding to the cue by possibleTask. These tasks are considered only “possible” because conditional branching or unexpected failures may prevent the tasks from being attempted:

```
(possibleTask goto-work1 enter1)
```

Of more interest is the ordering relationship between the atomic steps, represented by the followedBy and conditionalFollowedBy predicates. Determining this relationship requires walking over the body task network expression, and keeping track of all the possible prior atomic steps and the sequences of conditions that must be satisfied for the current step to follow each of these. We have to consider multiple prior atomic steps when considering a step following a parallel or select construct. An atomic step following a context construct or within a select or wait will be executed only if the appropriate conditions holds, and there may be multiple conditions between the execution of one atomic step and another. To translate iterative constructs, we need to introduce "null" tasks to link the start and end of the loop.

For simplicity, we have ignored the possibility of alternative procedures for the same action. However, this can be represented by making each procedure a subtype of the event type for the action, and the making the cue an instance of a step of that subtype.

PPL and other Languages

PPL and PSL

A well-known standard for process representations is the Process Specification Language (PSL) [Gruninger 04], a logic-based standard for capturing the semantics of processes.

PPL is intended as a complement to, rather than competitor of, PSL, as it captures process knowledge in a different way. The most significant difference between PPL and PSL is that PSL's representation of the ordering of steps is in terms of actual events ("activity occurrences"), while PPL orders the abstract flow chart steps ("activities") themselves. For example, in PSL the above toy plan for going to work would be

```
% enter is a subactivity of going to work
(forall (?person ?car)
   (subactivity (goto-work ?person ?car)
               (enter ?person ?car)))

% driving is a subactivity of going to work
(forall (?person ?car)
   (subactivity (goto-work ?person ?car)
               (drive ?person ?car)))

% In all occurrences of going to work, a driving
occurrence
% follows a entering occurrence.
(forall (?occ ?person ?car)
  (implies (occurrence_of ?occ (goto-work ?person ?car))
     (exists (?occ1 ?occ2)
        (and (occurrence_of ?occ1 (enter ?person ?car))
           (occurrence_of ?occ2 (drive ?person ?car))
           (subactivity_occurrence_of ?occ1 ?occ)
           (subactivity_occurrence_of ?occ2 ?occ)
           (successor ?occ1 ?occ2)))))
```

The last axiom states that for all occurrences of going to work, there will be an occurrence of entering followed by an occurrence of driving. While this makes the semantics of the original flow chart explicit, the actual structure of the flow chart has been lost (the simple relationship "enter $\rightarrow$ drive" is expressed as a complex quantified logical expression).

In principle one could perhaps recover the original flow chart by reverse-engineering it from these PSL axioms, either by parsing the axioms themselves[1] or by theorem proving the general relationships (e.g., proving that driving always follows entering in goto-work). However, neither of these options is particularly easy. In contrast, our goal with PPL is to preserve the original flow chart structure so that it is directly accessible for other agents. One could imagine extending PSL to include some predicate "macros" that would allow the general flow chart relationships to be made explicit, and which would also expand to the traditional PSL axioms such as those shown above. Conversely, PSL makes explicit the actual semantics of the flow chart, and PSL could be generated from PPL if one wanted to make these semantics explicit (indeed, PPL could be a "straw man" candidate for such PSL "macros").

PPL and OWL-S

OWL-S is a OWL-based Web service ontology, which supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in an unambiguous, computer-interpretable form. Like PPL, OWL-S represents generic procedures themselves, and similarly uses individuals to denote process objects and parameters used by those processes (In this sense, PPL is more similar to OWL-S in approach than to PSL). Generally speaking, OWL-S process is not a program to be executed. It is a specification of the ways a client may interact with a service. A process can generate and return some new information based on information it is given and the world state, or it can produce a change in the world. This transition is described by the preconditions and effects of the process. Processes can be atomic or composite. The composite processes may have control structure such as sequence, parallel, split, join, if-then-else, etc.

Clearly, the scope and the applicability of OWL-S is much different from either SPARK-L or PPL. In terms of expressiveness, OWL-S is comparable to SPARK-L as both are expressive process description languages. OWL-S, PPL, and SPARK-L all use similar representations for

[1] Mike Gruninger reports that a group has done this for a database application, but that this relies on an assumed syntactic regularity in the axioms in order to make parsing feasible [personal communication].

inputs, outputs, and results. Over and above this shared core, PPL provides a representation for the control structure in a process by using the followedBy relation. The current design of PPL is limited to just that. PPL is a subset of SPARK-L designed to support the requirements of reasoning applications. If one were to perform a similar reasoning over OWL-S processes, a PPL-like subset will need to be identified in order to avoid the potential overkill of using OWL-S in its entirety. In a similar vein, PPL has a simple KIF-like syntax, intended to be easily generated and processed by sending/receiving applications, both abstracting out some details and making some implicit semantics of SPARK-L explicit (e.g., event ordering). In contrast, OWL-S is specifically designed to support Web-based services, and hence uses an RDF-based syntax, clearly appropriate for Web-based applications but possibly more cumbersome to deal with in the wider context of process communication. Conversely, OWL-S (and similarly BPEL4WS, the Business Process Execution Language for Web Services) has gone further than PPL in defining different types of process ordering and parallelism. Some of these constructs would be useful to incorporate in PPL as it matures.

Experience Using PPL

PPL directly represents the steps in a procedure, the parameters of those steps, and their ordering. This representation allows a user's ambiguous utterances (e.g., ”find me”) to be matched against expected tasks in the procedure (e.g., ‘‘find_computer”) to identify the user's intent.

We have implemented a translator from the SPARK representation language to PPL. Using this translator, we exported SPARK process models. Using the exported process model, we ran a suite of tests on a dialog interpretation module, and for answering questions. In both cases, the system was able to resolve the user's utterances correctly in a simple, scripted dialog, illustrating proof of concept. Obviously, this is only a first step, but the demonstrated feasibility of the mechanism is encouraging.

Future Plans

The work we have reported here is just an initial attempt at developing a representation that will bridge the requirements of executing a process, and being able to answer questions about it. Clearly, more work needs to be done. First, the language needs to be extended to capture a larger subset of SPARK-L representation. The current PPL ignores many of the details of a process model, for example, the conditions. Second, we would also like to use PPL for specifying and/or modifying existing processes, e.g., through interaction with the user. This latter goal requires reversing the information flow so that PPL is used to generate SPARK-L, rather than the reverse. Currently PPL is too impoverished to support this, but with some small extensions this should be possible, so that either the PPL contains enough information to generate/modify a SPARK process, or suitable software can be written to “fill in the gaps” appropriately (and perhaps interactively) when passing information back to the execution environment.

Although PPL is still preliminary, the general idea of distinguishing representations for execution and representations for introspective reasoning has proved fruitful in our work, and one which we believe will continue to have value as our project progresses further.

Acknowledgement

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), or the Department of Interior-National Business Center (DOI-NBC).

References

Myers, K. L. CPEF: *A Continuous Planning and Execution Framework*. AI Magazine, vol. 20, no. 4, 1999.

Lemai, S., Ingrande, F. Interleaving Temporal Planning and Execution in Robotics Domains, *AAAI 2004*, July 25-29, 2004, San Jose, California, USA.

Morley, D. and Myers, K. The SPARK Agent Framework, in *Proc. of the Third Int. Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS-04)*, New York, NY, pp. 712-719, July 2004.

Yeh, P., Porter, B., and Barker. K. Using Transformations to Improve Semantic Matching. *Second International Conference on Knowledge Capture*, October 23-25, 2003.

Gruninger, Michael. Ontology of the Process Specification Language, in *Handbook of Ontologies*, pp575-592, Ed: S. Staab, R. Dtuder, Berlin: Springer (2004).

Erol, K., Hendler, J., and Nau, D. *Semantics for Hierarchical Task-Network Planning.* Technical Report CS-TR-3239, Computer Science Department, University of Maryland, 2004.