

An Analysis of Procedure Learning by Instruction

Jim Blythe

USC Information Sciences Institute
4676 Admiralty Way,
Marina del Rey, CA 90292 USA
blythe@isi.edu

Abstract

Many useful planning tasks are handled by plan execution tools, such as PRS, that expand procedure definitions and keep track of several interacting goals and tasks. Learning by instruction is a promising approach to help users modify the definitions of the procedures. However, the impact of the set of possible instructions on the performance of such systems is not well understood. We develop a framework in which instruction templates may be characterized in terms of syntactic transforms on task definitions, and use it to explore the properties of coverage, ambiguity and efficiency in the set of instructions that are understood by an implemented task learning system. We determine what kind of ambiguity is affected by the instruction set, and show how context-dependent interpretation can increase efficiency and coverage without increasing ambiguity.

Introduction

Task management systems are becoming increasingly important in a number of areas, for example intelligent office assistants, task-aware user interfaces and autonomic computing. End users need to customize the procedure definitions used in these systems, but find them difficult to modify for a number of reasons. First, users do not know the syntax and semantics of the procedure definitions, or the terms in ontology. Second, the definitions, which are combined to produce task behavior, are highly interrelated, so that changes to one procedure definition may produce unexpected effects when other procedures are executed. Third, successful modifications to a procedure base often require keeping track of a sequence of smaller modifications.

Among techniques that can help users create or modify process descriptions are programming by demonstration (PBD) and expectation-based systems. PBD, which learns procedure definitions inductively from user-provided examples, has the advantage that users don't need to worry about implementation details (Lau, Bergman, Castelli, & Oblinger, 2004; Lieberman, 2001). However PBD typically requires concrete examples and access to all the relevant information, and several training examples may be required in rich environments. Expectation-based techniques also shield the user from the procedure syntax but may still require her to make implementation-level

decisions and keep track when a modification requires several changes (Blythe, 2001; Kim & Gil, 1999).

Procedure modification by instruction is a complementary technique (Huffman & Laird, 1995), that has recently shown promise for helping end users modify the procedures used in a task execution system (Blythe, 2005). Figure 1 shows how a procedure definition is modified following a user instruction. First, the instruction is matched to a *template* from a library. The template denotes a class of potential modifications, and has slots for the required information. Next, the template is *instantiated* based on the instruction. An instantiated template completely specifies a modification to the procedure definition. The choices during instantiation depend on the form of the current procedure and the instruction. The modification system may propose several candidate modifications, and may analyze potential problems introduced in the global problem solver and suggest remedies to the user.

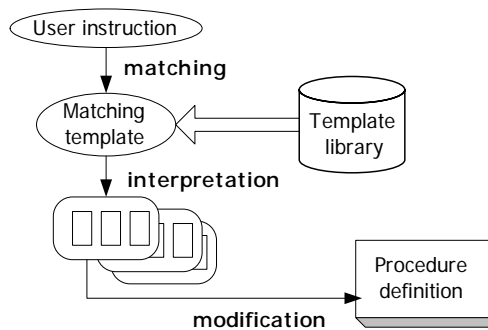


Figure 1. An instruction to modify a procedure is matched to a template. Then the template is instantiated and the resulting modification is tested.

Modification by instruction has several advantages. First, the user can refer to a modification at the operational level rather than the implementation level, for example saying 'do A before B' rather than 'add a sequence consisting of A followed by B into the original parallel construct'. Second, the approach allows the user to refer to conditions and tasks directly, rather than through examples. For example, with the instruction 'email me if it costs more than \$400', the user indicates both a task that the system is to add and the key condition under which it should be performed.

Third, the user can refer to the task and condition in their own terms, and an instruction-based tool can attempt to match to the domain ontology by search, e.g. 'send <message> <recipient> <mode>=email' and condition, e.g. 'totalPrice(chosenBid(laptop1)) < 2000'. Finally, the tool can be used either on a desktop or through speech, since it relies primarily on a sentence as input, and can easily be embedded in other systems.

However there are several challenges that need to be addressed for modification by instruction to have its full impact. These include understanding (1) what set of templates may be needed to guarantee one can make any required modification to a procedure definition, (2) how often more information is required to resolve an ambiguous instruction and (3) whether the templates capture required changes succinctly. This paper develops a framework to address these questions, and explores them in the context of an implemented tool for modification by instruction. Our framework considers the properties of a set of instruction templates, which characterize the capabilities of the tool. In particular, we provide a framework for estimating:

- (1) the *coverage* of a set of instruction templates: the proportion of the desired modifications to a procedure knowledge base that can be made,
- (2) the *ambiguity* in a template set: the average amount of feedback required from the user to uniquely identify a desired modification,
- (3) the *efficiency* of the template set: how well it compresses and simplifies the descriptions of desired modifications.

We begin by considering the possible modifications to procedures at the syntax level, defining transform operators that capture these modifications. We choose a complete set of transform operators as a reference, and estimate the completeness of a set of instruction templates in terms of how well it covers this reference set. We address ambiguity by examining conditions under which an instruction template may represent different modifications at the behavioral level. We relate the efficiency of a set of instruction types to the average number of transform operators that must be combined to express them.

We illustrate the approach with Tailor (Blythe, 2005), an implemented tool for modifying procedure definitions by instruction developed for the SPARK task execution system (Morley & Myers, 2004), which is inspired by PRS (Georgeff & Lansky, 1987). We use the framework we develop to explore the coverage and efficiency of the set of instruction templates intended for Tailor, and show that a relatively small set can cover the possible modifications in SPARK. Our approach is applicable to a broad range of modification and task execution systems.

In the next section we describe a short scenario illustrating the use of SPARK and Tailor. We follow with a simplified description of the syntax of SPARK, and describe a reference set of syntactic modifications to procedure knowledge. Next we describe the set of templates used in Tailor and provide their interpretations in terms of

transform operators. Only a subset of the templates is currently implemented. In the following section we discuss the ambiguity, completeness and coverage of this template set. We show examples of context-dependent interpretation of templates, where the template may entail different modifications depending on the context of the current procedure definitions. We finish with a discussion of our current framework and planned future work.

An Example of Task Modification by Instruction

Consider an office purchase task, where a user needs to customize the authorization procedure for a new department. The user gives the following instruction: “don’t get authorization if it costs less than \$2000”, which is recognized as an instance of the instruction template “[*don’t*] do <task> [*if* <condition>]”. The modification system instantiates the instruction by mapping the words “get authorization” into an existing task and the words “it costs less than \$2000” into a logical expression that is valid at the point in the plan when the task is considered. If either of these mappings requires making a choice, the system may ask the user for clarification, or pick a default based on generic reasoning about processes. Once the modification is agreed, the new procedure knowledge is provided to SPARK in executable form.

In this instruction, the user refers to several features of the solution without having to know the precise terms in the ontology. The user also does not need to know the syntax of the procedure language, or precisely how the modification will be implemented when there are several choices at the syntactic level. In addition, some instructions can compactly represent several modifications. For example, the instruction “use instant messaging instead of email” replaces one step with another that achieves the same goal, and may correspond to the two steps of deleting the email step and inserting an instant messaging step.

There are also some potential disadvantages of the technique. For example, in the instruction above it is not clear what the “cost” refers to. There may be several objects that could have a cost, and knowledge of the current situation is needed to know whether such a statement is ambiguous. Furthermore, given the instruction “send email before you get authorization”, it is not clear whether the user is requesting a new email step to be added or wishes to reorder an existing emailing step.

In the following sections we consider other instruction templates that the modification system may recognize. For example, the template “do <task> before/after/during/instead of <task> [*if* <condition>]” recognizes the inverse instruction “get authorization before placing the order”. The template “use <object1> instead of <object2> [*for* <task>] [*if* <condition>]” recognizes similar requests made in terms of task parameters rather than direct tasks, such as “use screws instead of nails to fix the planks”. This expression’s interpretation is context-dependent. For example, if one ‘fix’ procedure can have

screws or nails as an argument, and currently uses nails, the instruction can be interpreted as changing the argument of the procedure. However, if there are separate procedures, ‘nail’ and ‘screw’, each using objects of the respective types to fix target objects together, then the instruction can be interpreted as replacing a ‘screw’ task with a ‘nail’ task. Context-dependent interpretation is a useful property, since it is unambiguous in each case. Therefore the user need not be concerned with the way the procedures have been represented.

As we design and implement procedure modification systems based on instructions, we wish to understand how well a set of instruction types serves this goal by balancing context-dependent interpretation, abstraction, naturalness and coverage with ambiguity. We quantify these properties by relating the instruction types to transformations on the procedure definitions made at the syntactic level. At this level there is no ambiguity and coverage is easy to determine, but users are forced to make implementation decisions and reason about the syntax.

Syntactic Transform Operators

We consider a set of syntactic transformations that can be made to SPARK procedures. Tailor interprets instructions by mapping them to transformations on SPARK. We begin by reviewing the syntax of SPARK. More details can be found in (Morley & Myers, 2004).

A SPARK knowledge base includes a set of procedures that define how to achieve user goals. A procedure consists of a *cue*, which is an event for which the procedure is appropriate, a *precondition*, which is a logical expression that must be true at the time of applying the procedure and a *body*, which defines the tasks to be executed in order to execute the procedure. These are built from term expressions, logical expressions, actions and task expressions. For simplicity, we only consider procedures with action cues and ignore database update actions.

A *term expression* represents a value. These include constants, variable symbols and function applications. A function application consists of a function name and a set of term expressions, e.g. (+ \$x 1). The set of constant types, function and predicate symbols depends on the domain in which SPARK is applied. For example, the office domain may include people, e.g. John-Smith, and functional relations such as office-of(John-Smith). A *logical expression* is constructed from a predicate symbol and a list of terms, (e.g. has-window(office312)), logical operators and existential quantifiers (e.g. (exists [\$x] (not (p \$x)))).

An *action* consists of an action symbol and a list of terms, e.g. (send-email \$message John-Smith). A *task expression* may take the form [do: <action>], or may be one of the expressions shown in Table 1, where t1, t2 and t3 are task expressions and p is a logical expression. The seq: and parallel: constructs may take an arbitrary number of tasks.

```
[noop:]      do nothing
[fail:]      fail
[seq: t1 t2] do t1 and then t2
[parallel: t1 t2] do t1 and t2 in parallel
[if: p t1 t2] if p is true, do t1, otherwise t2
[try: t1 t2 t3] do t1, and do t2 unless t1
                fails, in which case do t3
[wait: p t1] wait until p is true, then do t1
```

Table 1: Rules for constructing task expressions

To develop metrics for a set of modification instructions, we consider sets of *transform operators* over the syntactic structures of task expressions. Each transform operator can be described as a pair of patterns, matching the original and new syntactic structures respectively. For example, the following pattern captures the transformation of adding a task to the end of a seq construct:

```
[seq: t+] -> [seq: t+ t']
```

where t+ denotes a sequence of one or more task descriptions, and t' denotes a new task to be added. We restrict new tasks to be one of [noop:], [fail:] or [do: <action>]. This transform operator is reversible: in reverse, it removes the last task from a seq: construct that has two or more tasks. A set of templates for user instructions generates a set of transform operators.

It is not hard to construct a set of transform operators that is complete in the sense that they can be combined to convert any task expression into any other. One approach is to mirror the recursive definitions by which compound tasks are constructed, as shown in Table 2. Each transform is reversible. We assume that arbitrary logical expressions can be added by the operators producing if: and wait: constructs. This assumption could be lifted by including a set of rules for logical expressions.

```
[seq: t+] <-> [seq: t+ t']
[seq: t] <-> t
[parallel: t+] <-> [parallel: t+ t']
[parallel: t] <-> t
[if: p t1 t'] <-> t1
[try: t1 ta' tb'] <-> t1
[wait: p t1] <-> t1
[do: <act>] <-> [noop:]
[noop:] <-> [fail:]
```

Table 2: A complete set of transform operators

The set shown in Table 2 is complete, since any task expression must be built following the constructs in Table 1, and for each construct we can find a corresponding transform in Table 2. The operators are reversible, and therefore can be used to transform any task expression to [noop:] and then transform [noop:] to any other task expression. Of course, this is usually a very inefficient process in terms of the number of operators applied.

This set of operators is by no means unique. For example, an alternative set might add tasks at the beginning of the seq: and parallel: constructs rather than at the end.

This would alter the order in which the transforms should be applied for a given modification and may significantly change the length of the optimal sequence.

Interpreting Instruction Templates as Transform Operators

We give interpretations of the instruction templates in terms of the syntactic transform operators described in the last section. As discussed earlier, the meaning of the instructions is context-dependent, so for some of the instructions we provide a set of alternative interpretations and describe the conditions under which they apply. In the following, a `<condition>` can be either a logical expression, e.g. `(and (cost $item $cost) (< $cost 2000))`, or “if `<action>` succeeds/fails”.

Adding a condition to a task

This operation is characterized by the instruction template

```
[don't] do <task> [if <condition>] [1]
```

If `<task>` is currently in the task expression and `<condition>` corresponds to a logical expression, the instruction can be identified with the operator,

```
task -> [if: p task]
```

where `p` is the condition, `<c>`, or `(not <c>)` if the sentence includes “don’t”.

If the condition takes the form ‘`<task2>` succeeds/fails’, and `<task2>` is not found in the task expression, the appropriate operators are

```
task -> [try: task2 task] ('succeeds')
-> [try: task2 [noop:] task] ('fails')
```

With tasks reversed if the word ‘don’t’ appears.

If both `<task>` and `<task2>` are in the task expression, they are together replaced by the appropriate `[try:]` clause.

This may entail re-ordering the tasks relative to each other or other tasks in the expression.

Adding a new task into a task expression

The instruction template for this operation is

```
do <task1> before/after/in parallel with/instead of/as part
of < task2> [if <condition>] [2]
```

where `<task2>` matches some task already in the task expression being altered, say `t2`.

The interpretation of this template depends on whether `<task1>` is already present in the task expression being modified. First, suppose that no task matching `<task1>` is in the task expression but the description matches a new task, `t1`. If the word ‘before’ is used, then the appropriate operator adds `task1` just before `t2`:

```
t2 -> [seq: t1 t2]
```

with similar operators for ‘after’ and ‘in parallel with’. If there is a condition that is a logical expression, we use

```
t2 -> [if p [seq: t1 t2] t2]
```

and if the condition tests success of a task we use a similar `try: construct`.

The phrase ‘instead of’ is taken to signify that the task is being replaced:

```
t2 -> t1,
or t2 -> [if: p t1 t2]
```

The phrase ‘as part of’ signifies that `t1` should be included when `t2` is accomplished, or in other words should be included in the body of every procedure whose cue matches `t2`:

```
(proc cue: t2 pre: p body: t3)
-> (proc cue: t2 pre: p
    body: [parallel: t1 t3])
```

This is similar to ‘do `<task1>` in parallel with `<task3>`’ except that it is applied to every matching procedure in the knowledge base, rather than to a current task expression.

Reordering tasks in a task expression

The same template may be used to reorder tasks when `<task2>` matches an existing task `t2` and `<task1>` matches existing task `t1`. To save space, omit the cases where conditionals are specified. These mirror the cases in the previous section. In the operators below, `ta*`, `tb*` and `tc*` designate arbitrary sequences of tasks that do not match `<task1>` or `<task2>`.

Keyword: *before*: (we omit *after*., which is similar)

```
[seq: ta* t2 tb* t1 tc*]
-> [seq: ta* t1 t2 tb* tc*]
[parallel: ta* t1 t2]
-> [parallel: ta* [seq: t1 t2]]
```

Keyword: *in parallel with*:

```
[seq: ta* t1 tb* t2 tc*]
-> [seq: ta* tb* [parallel: t1 t2] tc*]
```

The template is ambiguous when both `task1` and `task2` match tasks in the current expression, since the user may either intend to reorder the tasks or to add a new task.

Waiting for a condition to be true

The instruction template:

```
wait for <condition>, then do <task> [3]
```

is used to add a `wait: construct` to a task expression. `<task>` should match a task in the current expression, `task`, and `<condition>` must match a logical expression `p`. If the task already depends on `p`, the condition is replaced by the `wait: construct`:

```
[if: p task] -> [wait: p task]
```

otherwise,

```
task -> [wait: p task]
```

Properties of Tailor’s Instruction Set

We have specified three major instruction templates to be recognized in Tailor, and shown how they are interpreted as transform operators. Currently, 4.1 and part of 4.2 are implemented in Tailor. Having provided definitions of the instructions in terms of syntactic transform operators, we

examine their properties, including coverage, efficiency and ambiguity.

Ambiguity

Ambiguity arises when an instruction may have several interpretations, and further information is required from the user to find which one is intended. Context-dependent interpretation is not ambiguous since, although there are two or more interpretations, the correct one is clear from the current procedure definition. We discuss ambiguity first because subsequent measures make adjustments for it, as we explain below.

We distinguish two kinds of ambiguity, based on the source. Term ambiguity occurs with an unclear reference to a term in an instruction, either a task or a logical condition. For example, consider the instruction “send email before getting authorization if it costs more than \$2000”. The logical condition is ambiguous since it is not clear what ‘it’ refers to. In the task expression being modified, several variables may already be bound from prior queries or task parameters that may have a cost according to the ontology, and applying one or more relations from the ontology may yield others. Tailor has a bias to prefer objects already referred to or shorter chains of references, but in some cases must ask follow-up questions to resolve ambiguity. Similar work must be done to create a concrete new task matching ‘send email’, to fill the parameters for the recipient and message.

The second type is *template ambiguity*, which occurs when there are several possible interpretations of the instruction into transform operators. For example, the same instruction has template ambiguity if the expression already contains a task to send email after getting authorization. In this case it is not clear whether the user intends to add a new email task or to move the existing one earlier.

Neither form of ambiguity presents a serious problem for the approach, given an interactive environment where alternative interpretations can be presented. Both forms can make the approach less convenient, however, by requiring more feedback from the user. The choice of instruction templates does not impact term ambiguity, since it is independent of the templates themselves, while template ambiguity is directly affected. Template ambiguity is an important property of a set of instruction templates in its own right, and it also affects efficiency described below.

Coverage

We measure coverage by the proportion of transforms in a complete set, for example Table 2, that can be addressed by instruction templates. In terms of syntactic transform operators, our set of instruction templates described in the previous section does not provide complete coverage. For example, there is no reliable way to transform from [seq: t] to the operationally equivalent t using the instruction templates defined. However, all of the transforms from Table 2 are covered by transforms from

the instruction templates up to purely syntactic differences, so this set provides complete coverage.

Efficiency

Two measures are used to capture the efficiency of a set of instruction templates. First, *coverage efficiency* is measured as the ratio of the number of transform operators required to cover the set of templates to the number of templates. Second, *expression efficiency* is measured as the average number of transform operators from a reference set that are required to model the instruction templates.

For an absolute measure of *coverage efficiency*, we would need to know the smallest possible set of transform operators giving complete coverage. We might also wish to account for the difference in expressiveness of the templates, whose range is increased by the use of multiple keywords, and the fact that the transform operators are designed to provide syntactic coverage while the templates are designed to provide operational coverage. However, the value is more useful as a comparison between alternative sets of instruction templates, so we use a constant reference set of transforms, introducing a constant multiplier on the value for any set of templates. We increase the count of instruction templates to the smallest number that could be specified without template ambiguity, to avoid favoring smaller, ambiguous sets.

In the example from the previous section, we have 3 main templates, ignoring [3]. Since [2] has template ambiguity, we increase the count to 4, essentially positing a copy of [2] that is only used for existing tasks and one that is only used for new tasks. Our reference set of transform operators from Table 2 has 9 operators, yielding a coverage efficiency of 9/4, or 9/5 if template [3] is included.

Roughly half as many templates are required as operators for two reasons. First, the use of multiple keywords allows a template to represent multiple operators. For example the keywords *before* and *in parallel with* in [2] correspond to seq: and parallel: operators respectively. Second, the use of context-dependent interpretation also increases the range of a template.

The *expression efficiency* is the average number of transform operators required to interpret a template. Again, this depends on the set of transform operators used. In particular, if we used the set of all interpretations as the reference set, efficiency could not be greater than 1. However, this set may have incomplete coverage, or poor coverage efficiency. A reference set serves to compare alternative sets of instruction templates. This property also depends on the population of modifications that is chosen to compute the average. Here we include each case in which the templates may be used and a direct example of each use of a transform operator, all weighted equally.

From the cases outlined above, we have an efficiency of approximately 1.6 for the example set. The improvement can be attributed to the optional “if” clauses in the templates, and the direct expression of reordering, which is not directly implemented in the reference set of transforms.

For example, consider the instruction: “Send me email before seeking authorization if the airline is not based in the US”. Suppose that ‘seeking authorization’ matches task `auth` in the current task expression, ‘if the airline is not based in the US’ matches condition `not-US`, and ‘send me email’ matches a new task `email`. Then the instantiated interpretation of template [2] is

```
auth ->
  [if: not-US [seq: email auth] auth]
```

In the reference set of table 2, this is achieved by combining the following transforms:

```
auth -> [if: not-US auth auth]
auth -> [seq: email auth]
```

Discussion

The benefits of allowing users to modify procedure definitions through instruction depend on a number of factors, including how large a set of instruction templates must be mastered, how efficiently they capture the changes users want to make and how much follow-up work is required to resolve ambiguity. In this paper we introduced a framework for quantifying some of the properties of a set of instruction templates that determine these features. We show how the choice of templates can impact ambiguity, coverage and expression efficiency. The properties are illustrated in the context Tailor, which helps users modify task expressions in SPARK through instruction.

It is important to see how well a set of templates captures the modifications that a user typically wants to make. We are investigating this aspect, although it is beyond the scope of this paper. Blythe (Blythe, 2005) uses a set of task modifications from a web site as a measure of typical modifications. We plan to augment our analysis with user studies to explore the use of the templates in practice.

Many task-based systems have been designed to take instruction from users. In advice-taking systems such as Foo (Mostow, 1981), the instruction takes the form of high-level advice that must be operationalized by the system in order to be used, e.g. “try to flush out the Queen” in a game of Hearts. Myers (Myers, 1996) characterizes the possible forms of advice in an advice-taking system. We have attempted this for direct task-modification instructions, although there is some overlap. INSTRUCTO-SOAR (Huffman & Laird, 1995) is one of the broadest instruction-taking systems and accepts a range of instructions to modify its procedure definitions, generalizing from instructions to perform particular actions and also learning operator control knowledge from more general instruction, e.g. “never pick up green blocks”. However, properties of the set of recognizable instructions such as completeness or conciseness are not investigated. Lau et al. (Lau et al., 2004) and Liebermann (Lieberman, 2001) also consider inductive techniques for updating procedures through examples. The analysis of procedure transformations provided here would also be relevant to these systems, providing further information on coverage and machine learning bias.

Work on the TRIPS and TRAINS projects at Rochester has addressed many aspects of dialog management in the context of collaborative problem-solving, including the combined use of context and dialog to help resolve ambiguous instructions (Allen et al., 2001). Our work is complementary as it considers instructions in the context of an executable procedure model, using properties of the model for disambiguation and to investigate completeness.

Quantifying the benefits from a set of instruction templates is only one aspect of an instruction-based system, although an important one. By distinguishing term ambiguity from template ambiguity, we isolate an aspect that is independent of the template set but must still be studied and reduced as far as possible. The context provided by the current procedures, the current task and the modification being made is a powerful guide to help reduce ambiguity that we will explore further in future work.

Acknowledgments

I am very grateful for comments from Yolanda Gil, Karen Myers, Varun Ratnakar, Tim Chklovski, Jihie Kim and Melinda Gervasio. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA), through the Department of the Interior, NBC, Acquisition Services Division, under Contract No. NBCHD030010.

References

- Allen, J., Byron, D., Dzikovska, M., Ferguson, G., Galescu, L., & Stent, A. (2001). Towards conversational human-computer interaction. *AI Magazine*.
- Blythe, J. (2001). *Integrating expectations from different sources to help end users to acquire procedural knowledge*. In Proc. IJCAI, Seattle, WA.
- Blythe, J. (2005). *Task learning by instruction in tailor*. In Proc. Intelligent User Interfaces, San Diego, CA.
- Georgeff, M., & Lansky, A. (1987). *Reactive reasoning and planning*. In Proc. AAI, Seattle, WA.
- Huffman, S., & Laird, J. (1995). Flexibly instructable agents. *Journal of AI Research*, 3, 271-324.
- Kim, J., & Gil, Y. (1999). *Deriving expectations to guide knowledge-base creation*. In Proc. AAI.
- Lau, T., Bergman, L., Castelli, V., & Oblinger, D. (2004). *Sheepdog: Learning procedures for technical support*. In Proc. Intelligent User Interfaces.
- Lieberman, H. (2001). *Your wish is my command*. San Francisco: Morgan Kaufmann.
- Morley, D., & Myers, K. (2004). *The spark agent framework*. In Proc. AAMAS, New York, NY.
- Mostow, J. (1981). *Mechanical transformation of task heuristics into operational procedures*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh.
- Myers, K. (1996). *Strategic advice for hierarchical planners*. In Proc. Knowledge Representation.