A Formal Framework for Combining Natural Instruction and Demonstration for End-User Programming[*]

Christian Fritz[†]
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 USA
cfritz@parc.com

Yolanda Gil
Information Sciences Institute
University of Southern California
Marina del Rey, California. USA
gil@isi.edu

ABSTRACT
We contribute to the difficult problem of programming via natural language instruction. We introduce a formal framework that allows for the use of program demonstrations to resolve several types of ambiguities and omissions that are common in such instructions. The framework effectively combines some of the benefits of programming by demonstration and programming by natural instruction. The key idea of our approach is to use non-deterministic programs to compactly represent the (possibly infinite) set of candidate programs for given instructions, and to filter from this set by means of simulating the execution of these programs following the steps of a given demonstration. Due to the rigorous semantics of our framework we can prove that this leads to a sound algorithm for identifying the intended program, making assumptions only about the types of ambiguities and omissions occurring in the instruction. We have implemented our approach and demonstrate its ability to resolve ambiguities and omissions by considering a list of classes of such issues and how our approach resolves them in a concrete example domain. Our empirical results show that our approach can effectively and efficiently identify programs that are consistent with both the natural instruction and the given demonstrations.

1. INTRODUCTION
Allowing users with no programming background to specify procedures that can automate tasks is a longstanding goal of AI and intelligent user interfaces research. The problem is challenging because the intended behaviors can be very complex and it can be difficult for non-programmers to describe the correct procedure in a natural way while having the system learn it correctly.

Programming by demonstration (PbD) approaches [16] allow end users to demonstrate the intended behavior to the system on a number of specific examples, and the system in turn tries to infer the intended behavior from these. This allows a much larger group of people to "program", as it does not demand specialized programming skills of the user. The shortcoming of PbD, however, is that it is often hard for the system to generalize from linear examples when the intended behavior has complex structures. Furthermore, in the existing approaches it is difficult for the user to control what is being learned or to recover from errors that were made during demonstration.

An alternative approach is to allow users to specify a procedure by explicit instruction (PbI) (e.g. [5]). A user can provide a general description in natural language regarding particular steps of the target procedure or its high-level structure. It is hence easy to express complex program structure. The disadvantage with PbI is that this type of instruction is prone to ambiguities and omissions.

Ideally one would want to combine PbD and PbI, as they have complementary strengths. It is also desirable because humans typically combine both methods when teaching complex procedures, either demonstrating first and then describing the general case or describing the procedure first and then demonstrating it. There have been some promising approaches for making PbD systems more robust to errors and allowing user edits interleaved with demonstrations (e.g., [19, 4]). CHINLE [4] is a system that generates domain specific PbD systems from declarative interface specifications. Via a compelling visualization of procedure hypotheses, the system allows users to discard individual steps of the learned (linear) procedure or explicitly add training examples in support of particular hypotheses. This systems however, limit the types of editing operations the user is allowed to perform and, in particular, does not allow the manual specification of the high-level program structure, which might be most helpful to the PbD part of the system. Some approaches propose the use of situated instruction, where PbI is grounded in an example state [12]. Some recent work on combining PbD and PbI has focused on reinforcement learning frameworks [17, 21, 13]. However, there are no general approaches for incorporating broader forms of instruction and at the same time only require one or very few examples. For the purpose of learning web procedures, some recent approaches have put more emphasis on the natural-language processing aspects of the problem of matching demonstrations with explicit descriptions of what is being done, e.g., [2]. Besides single actions, Allen et al. also describe a way for learning

[*]We gratefully acknowledge support from the US Defense Advanced Research Projects Agency under grant number HR0011-07-C-0060.

[†]Work done during post-doc at USC/ISI.

IUI'11, February 13–16, 2011, Palo Alto, California, USA.
Copyright 2011 ACM 978-1-4503-0419-1/11/02...$10.00.

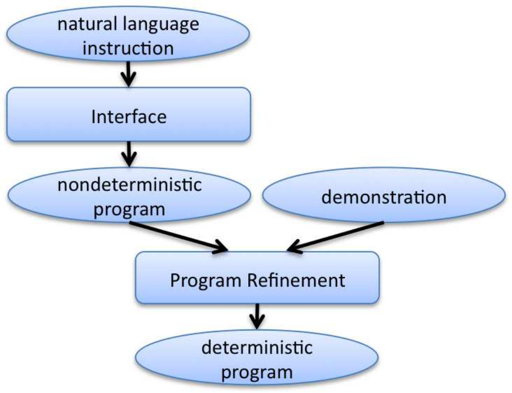

Figure 1. Schematic data-flow in our framework.

iterations but for that rely on explicit indication of loop start and end by the user.

We approach this problem by developing a formal, comprehensive framework for representing, updating, and executing program hypotheses. Our goal with this framework is to enable any learner to output its acquired knowledge in this form, allow for the combination of (sets of) program hypotheses, so that different input/learning methods can be combined, and enable the execution of partially learned programs if necessary, hence enabling learning from experience or explicit user feedback as well. Our framework is inspired by the recent approaches for combining PbD with user edits and integrates PbD with PbI in a more flexible manner. The key ideas of our approach are as follows:

1) We can represent (infinite) sets of program hypotheses using *non-deterministic programs*, in particular we can represent the incomplete procedures that can result from a PbI system. These are used to represent uncertainty about the target program being learned. In particular, we use the logical programming language *Golog* [15] which readily allows us to represent and reason about such non-deterministic programs. A Golog program can be understood to represent sets of deterministic programs, namely all those that would result by resolving the non-determinism in one way or another. We have developed a simple, controlled grammar based interface that generates Golog programs from English instructions. This PbI system uses Golog's non-deterministic programming constructs in places where instructions are ambiguous or incomplete.

2) We can refine these program hypotheses learned through PbI based on an example from a PbD system. We extend Golog's semantics to implement the update function that removes from a set of hypotheses all those that are inconsistent with newly given demonstrations. This is implemented as *refinements* to the program representing the version space, resolving some of the uncertainty, i.e., making some non-deterministic parts of the program deterministic. In many cases, a single example can be sufficient to resolve all non-determinism, resulting in the data-flow depicted in Figure 1.

3) We can integrate any two program hypotheses into a single one, for example to integrate the output of a PbI system with the output of a PbD system that has learned from several demonstrations. We accomplish this by defining a mechanism for Golog *program synchronization* that implements a provably sound and complete means of computing *symbolic intersection* of (possibly infinite) sets of hypotheses.

An important feature of this framework is that it can accommodate incremental learning over time while allowing the learner to test its current procedure hypothesis by executing it. In essence, a procedure hypothesis in our framework is akin to a *procedure version space* in that it represents many possible hypotheses about the procedure in a very compact manner. By virtue of basing the framework on Golog and its semantics in the situation calculus, we get as a side-effect the ability to execute, i.e., try-out, program hypotheses at any time, even while there still remains a lot of uncertainty. This exploits Golog's integration with automated planning and further facilitates learning from experience as well.

We will show that our system is able to resolve the following kinds of issues commonly occurring in human instruction [8], when combined with a demonstration of the target program in an example scenario:

- mapping of objects to action arguments,
- missing action arguments,
- ambiguous references like "him" or "it",
- ambiguous scoping of conditionals and iterations, and
- unknown terms to refer to known actions or functions.

The next section describes our evaluation domain and goals. It is followed by a review of the situation calculus and Golog. Section 4 describes how we map natural language instructions into Golog programs. In Section 5 we present our approach for refining these programs given an example demonstration, and our empirical evaluation. Section 6 describes our approach for program synchronization.

2. LEARNING COMPLEX GAME PLAYING PROCEDURES

To evaluate our approach's ability to resolve types of omissions and ambiguities that are common in human instruction we consider procedures in the open-source real-time strategy game Stratagus/Wargus[1]. In this game, the goal of the player is to defeat all of the opponent's agents using his own footmen. Footmen units can be built in *barracks*, which in turn can be built by *peasants*. In order to provide food for his units, the player also needs sufficient *supplies*, which are provided by *farms*. Farms can, again, be built by peasants and each farm provides enough supply for four units. As an example, we consider a family of scenarios where initially the player only has one peasant, and the opponent, controlled by the computer, has N footmen. In order to win, the player has to first use his peasant to build barracks where he then can build footmen. In order to do so, he also needs to build farms to create the supplies for the footmen. The screen-shot

[1]`http://wargus.sourceforge.net/`

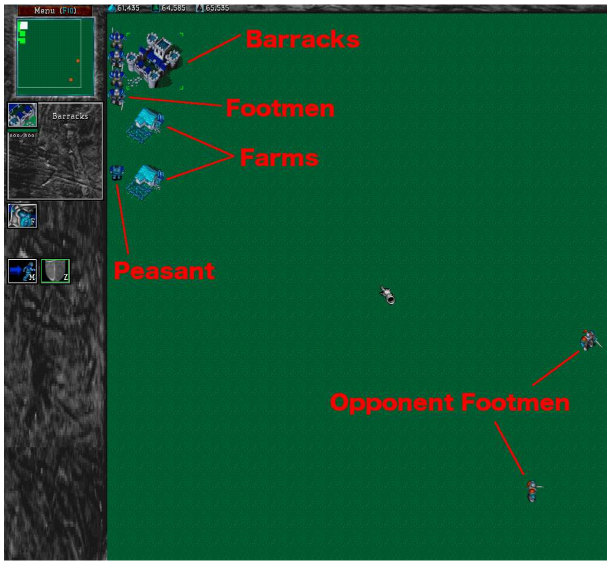

Figure 2. Screen-shot from the Stratagus/Wargus game.

in Figure 2 shows the situation where the player completed building barracks, farms, and four footmen (twice as many as the opponent's) and is now ready to attack the opponent. Outnumbering the opponent by a factor of two he is certain to win the game.

As a running example in this paper, we will show how a user can teach this general procedure to the computer by combining natural instruction and a demonstration in an example scenario with two opponent footmen. The procedure that will be taught will be applicable to any scenario where there is at least one peasant and any number of opponent footmen. The strategy is to always build twice as many footmen before attacking the opponent. This procedure contains several loops and iterations over sets.

Instructions generally contain several types of omissions and ambiguities simultaneously. Consider the following text, which is understood by our interface. These instructions omit action arguments, lack scopes for loops and iterations, use unknown terms, and contain ambiguous references.

Example 1. *"Build barracks using bestPeasant then wait until bestPeasant is ready, while noOfFootmen is less than noOfOppFootmen times 2, if supply is less than 1 then build farm, wait until bestPeasant is ready, create footman, wait until bestBarracks is ready, while there is an opponent who is alive, take the closestOpponent, forall footman attack him, wait until he is dead."*

Our system is given these utterances and a demonstration of the intended behavior using the Wargus interface shown in Figure 2. A video of the demonstration can be found on our web site [1]. Our goal is to produce the program of Figure 3. In our empirical results of Section 5.4 we consider instructions describing this procedure with varying amounts of omissions and ambiguities (Example 1 is one of them). We will see that our framework allows us to (a) represent trillions of program hypotheses compactly, and (b) use a

```
build( bestPeasant, barracks) ;
waitfor( ready(bestPeasant)) ;
while noOfFootmen < noOfOppFootmen · 2 do
    if supply < 1 then
        build( bestPeasant, farm) ;
        waitfor( ready(bestPeasant) );
    build( bestBarracks, footman );
    waitfor( ready(bestBarracks) );
while (∃x) opponent(x) ∧ alive(x) do
    set( o, closestOpponent);
    foreach a, footman(a) do
        attack( a, o)
    waitfor( dead(o));
```

Figure 3. Our target program.

demonstration to effectively and efficiently identify which of these hypotheses are consistent not only with the natural language instructions, but also the demonstration. This will produce the above target procedure.

While the focus of this paper is not on the interface, we study the naturalness of the user experience when teaching using a natural, controlled grammar in a separate line of work, where we implement a system for teaching scientific workflows. This system, which extends the Wings Workflow System [10], will make it ever easier for non-programmers to author new workflows or modify existing ones.

3. PRELIMINARIES

We use the situation calculus to formalize our approach. The situation calculus is a sorted logic for specifying and reasoning about dynamical systems [20]. In the situation calculus, the state of the world is expressed in terms of *fluents*, functions and relations relativized to a *situation* s, e.g., $F(\vec{x}, s)$. A situation is a history of the primitive actions a performed from a distinguished initial situation S_0. The function $do(a, s)$ maps an action and a situation into a new situation thus inducing a tree of situations rooted in S_0. We abbreviate $do(a_n, do(a_{n-1}, \ldots, do(a_2, do(a_1, s)))$ to $do([a_1, \ldots, a_n], s)$ or $do(\vec{a}, s)$. We denote the set of actions by $\mathcal{A}$. In the situation calculus, all actions have deterministic effects and this is what we are assuming in this paper.

In the situation calculus, background knowledge of a domain (e.g., Wargus) is encoded as a *basic action theory*, $\mathcal{D}$. It comprises four domain-independent foundational axioms and a set of domain-dependent axioms. Details of the form of these axioms can be found in [20]. Included in the domain-dependent axioms are the following sets:

Initial state axioms, $\mathcal{D}_{S_0}$: a set of first-order sentences relativized to situation S_0, specifying what is true in the initial state, e.g., *Peasant*$(1, S_0)$ and *PlayerId*$(1, S_0) = 0$ state that in the initial situation agent number 1 is a peasant and belongs to player number 0 (us).

Successor state axioms: provide a parsimonious representation of frame and effect axioms under an assumption of the completeness of the axiomatization. There is one successor state axiom for each fluent, F, of the form $F(\vec{x}, do(a,s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is a formula with free variables among $\vec{x}, a, s$. $\Phi_F(\vec{x}, a, s)$ characterizes the truth value of the fluent $F(\vec{x})$ in the situation $do(a,s)$ in terms of what is true in situation s.

Action precondition axioms: specify the conditions under which an action is possible. There is one axiom for each action a of the form $\mathit{Poss}(a(\vec{x}), s) \equiv \Pi_a(\vec{x}, s)$ where $\Pi_a(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$. For instance, $\mathit{Poss}(\mathit{Build}(x, \mathit{Footman}), s) \equiv \mathit{Supply(s)} > 0 \wedge \mathit{Barracks}(x)$, states that one can build a footman using x only if x is of type barracks, and there is enough supply for at least one more unit.

Given the semantics, situations compactly describe *traces* of state-action pairs and can hence be used to represent demonstrations of procedures.

We have implemented a basic action theory that specifies the available actions in the Wargus domain together with their arity, as well as basic concepts that can be useful in teaching various behaviors, such as "number of footmen" (noOfFootmen), "number of opponent footmen" (noOfOppFootmen). In the basic action theory we also define the space of possible "values" that can be used as action arguments. Beyond that we could have incorporated more details about actions, including their preconditions and effects as describe above. Such background knowledge can improve a systems ability to disambiguate between possible candidate programs a user is trying to teach. However, since this is not focus of this paper, we used a very simple domain encoding for Wargus, that does not describe any action preconditions or effects. These may, however, play a role in future work, when considering demonstrations that only describe the evolution of the state of the world, without explicit mentioning of action occurrences. In those cases we would need to be able to conjecture action sequences that would explain "what happened", as it has been described in the literature (e.g., [18]).

3.1 Golog

Golog [15] is a programming language defined in the situation calculus. It allows a user to specify programs whose set of legal executions defines a sub-tree of the tree of situations of a basic action theory. Golog has an Algol-inspired syntax extended with flexible *non-deterministic constructs*. Golog programs are created using the following constructs:

nil	empty program
$a \in \mathcal{A}$	primitive action
$\phi?$	test condition ϕ
$[\delta_1; \delta_2]$	sequence
if ϕ **then** δ_1 **else** δ_2	conditional
while ϕ **do** δ'	loop
$(\delta_1 \mid \delta_2)$	non-deterministic choice
$(\pi v)\delta(v)$	non-deterministic choice of argument
δ^*	non-deterministic iteration

In addition, Golog enables the definition of procedures. The semantics of a Golog program δ is defined in terms of macro expansion into formulae of the situation calculus. $Do(\delta, s, s')$ is understood to denote a formula expressing that executing program δ in situation s is possible and may result in situation s'. This is defined inductively over the program structure. For instance for primitive actions: $Do(a, s, s') \stackrel{\text{def}}{=} \mathit{Poss}(a[s], s) \wedge s' = do(a[s], s)$, where $a[s]$ denotes the action a with all its arguments instantiated in situation s. For simple non-determinism: $Do(\delta_1 \mid \delta_2, s, s') \stackrel{\text{def}}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$. The complete semantics can be found in [15]. In addition to the standard constructs, for the purpose of this paper we also use a convenience construct $\textbf{foreach}\, v, \varphi(v)\, \textbf{do}\, \delta(v)$, which is an iteration over all possible bindings for variable v such that $\varphi(v)$ is true, executing program δ for each such binding. We call a Golog program *deterministic* if it contains neither $(\delta_1|\delta_2)$ nor (πv) nor δ^*. While deterministic constructs enforce the occurrence of particular actions, these non-deterministic constructs define "open parts" that allow for several ways of interpretation/execution.

In the context of this paper we will interpret non-deterministic parts of a Golog program as uncertainty about the precise target procedure being learned. We will see how this uncertainty can be resolved via consideration of demonstrations. Roughly, non-deterministic constructs can be compared to the "union" operator and all other constructs to "join" operators often used in version space algebra based approaches (e.g., [14]). Hence, just like version space algebras hierarchically combine simpler spaces into more complex ones, we can use the inductive definition of the Golog language to combine several sub-programs into more complex ones. The availability of the concept of procedures is very helpful in this respect: it allows us to define and reuse sub-version spaces and give them names.

4. MAPPING INSTRUCTIONS INTO GOLOG PROGRAMS

We have created a simple, controlled-grammar based input interface for users to write English instructions, describing a program they would like the system to learn. These instructions are then translated into a Golog program, using non-determinism in places where details were omitted or things were ambiguous. The grammar and translation driving this interface is domain independent and only takes a basic action theory, i.e., a domain description, as input in order to recognize and conjecture action names and functions, fluents, and constants. The interface, deployed via a web application, give feedback to the user in terms of which parts of the typed-in instructions are being understood and how they are interpreted. Nevertheless, the degree to which the language supported by our grammar and the use of this interface seems natural to the user is not focus of this paper. Instead, we evaluate our work with respect to the types of omissions and ambiguities our framework can resolve by means of considering one or more example demonstrations.

In the remainder of this section we will discuss the omissions and ambiguities our framework can handle, some of which are contained in above instructions, and will describe how they can be represented in Golog. In the next section, we will then describe how our approach can take such a Golog program as input and then "refine" it by incorporating the information contained in the given example demonstration.

This will produce the above target program.

Ambiguous mapping of action arguments:
When a user says: *"attack the closestOpponent with bestFootman"*, it is not clear which of these arguments shall be first and which second. To accommodate for this ambiguity, we use non-deterministic choice to model all possible orderings of the stated arguments. We hence represent this sentence as:

$$\big(attack(closestOpponent, bestFootman) \mid \\ attack(bestFootman, closestOpponent)\big)$$

Upon consideration of an example demonstration, as described in the next section, our refinement algorithm will discover which of the two possibilities actually explains the demonstration (is consistent with it) and will hence only include that possibility in the refined program. This will always be possible as long as the two arguments do not have the same value in the given demonstration. Note, however, that the demonstration does not mention functions or fluents, but only concrete values. Hence, instead of $attack(bestFootman, closestOpponent)$ the demonstration will state, e.g., $attack(1, 5)$, if agent number 1 is currently the $bestFootman$ and agent number 5 is the $closestOpponent$.

Missing action arguments:
Human teachers often omit action arguments when they seem obvious from the context or by common sense. Since we assume that the domain definition contains an exhaustive list of all possible actions in the domain of interest together with their respective arity, it is easy to detect whether an argument is missing. But the system still needs to figure out what value (function, fluent, or constant) to use for the missing argument. For instance, in Wargus it seems natural to say *"build barracks"*, whereby one omits the first required argument of the "build" action, which specifies the peasant that shall be used to build the barracks. However, if, e.g., there is only one peasant, it is really obvious to a human what that argument should be.

Omissions of this kind are represented in our framework using Golog's π construct. This introduces a new, existentially quantified program variable, representing the unknown value, e.g.:

$$(\pi v)[value(v)?; build(v, barracks)].$$

The test, $value(v)$, restricts the variable to be of type "value". During refinement, our algorithm considers all possible such bindings that evaluate to the same concrete value as the one appearing in the demonstration in the respective current state of the demonstration.

Ambiguous references:
It is common for a human to refer to previously introduced terms using pronouns ("he", "it", "him", etc.) where again, to a human it is often clear from the context or common sense which object is referred to. In our interface, we allow for the explicit introduction of reference objects in the context of quantification. For instance, when the user says *"take the closestOpponent, all footmen, attack him"* he both omits the subject of the attack and uses the ambiguous reference "him". To a human, who knows that footmen attack, it might be clear that in each case of the implicit iteration, the considered footman shall be used as the subject of the attack and the closestOpponent as the object. The demonstration for such a statement would involve multiple instances of the attack action, all of which share the same object but have different subjects. As we will see, this is exploited by our refinement algorithm to disambiguate the instruction.

Ambiguous scoping:
Specifying the scope of an if-then condition, a while-loop, or an iteration over a set is something humans rarely remember to do when explaining a program, and in fact generally does not confuse human students all that much as humans are good at figuring out from context or common sense where the respective instruction block logically ends. Our interface does not require the user to specify the scope of such blocks either, but as a result, there are often numerous combinations of possible scopes. Consider, e.g., the instruction: *"Take the closestOpponent, forall footman, attack him with the footman, wait until he is dead. "* The iteration in this sentence does not have an explicit scope. It is hence not clear whether or not the waiting action is part of the body of the iteration or not. This can be represented in Golog as:

$$\begin{aligned}&\big[set(o, \textit{closestOpponent});\\ &\ \big(\big[\,\textit{foreach}(a, \textit{Footman}(a), \textit{attack}(a, o));\\ &\quad \textit{waitfor}(\textit{dead}(o))\big] \;\big|\\ &\quad \textit{foreach}(a, \textit{Footman}(a),\\ &\qquad\qquad [\textit{attack}(a, o); \textit{waitfor}(\textit{dead}(o))])\,\big)\big]\end{aligned}$$

Using unknown terms: In the Wargus domain, there is an action called "build" that can be used to build new units (e.g., footmen). When a user is unfamiliar with the correct action names, he may however say: *"create a footman at the barracks"*. He may even at the same time omit an argument: *"create a footman"*. To represent this sentence in Golog and make these ambiguities explicit, we create a non-deterministic choice between all possible actions in the domain that have the right number of arguments or more. For actions that have more arguments, we insert existentially quantified variables, as described above. Unknown terms appearing in the place of values are handled analogously.

Putting it all together: The instructions of Example 1 contain all of these types of omissions and ambiguities. The Golog program representing these instructions is too large to show in this paper but can be found on our web site: [1]. While large, the program is still rather compact compared to the set of 486400 candidate programs, which it describes. This program can be used as input to the program refinement approach we describe in the next section, to produce the sought target procedure, when given the example demonstration.

5. REFINING GOLOG PROGRAMS

In this section we describe how the existing Golog semantics can be extended to refine a program given an example. This will allow us to take the Golog program representing the user instruction and refine it using any provided demonstrations. We begin by defining the problem of programming by demonstration and what counts as a solution in our setting. We assume the target program to be deterministic. However, in the absence of sufficient training data, we may not be able to specify all the details of this program and hence still end up with a non-deterministic program after considering a number of examples. This is reflected in the definition.

Definition 1 (PbD Problem)**.** A *PbD problem* is a tuple $\langle\delta, \{(S_1, S'_1), \ldots, (S_n, S'_n)\}\rangle$, where δ is a Golog program and each (S_i, S'_i) is a tuple of situation terms such that $S'_i = do(\vec{a}, S_i)$ for some sequence of actions $\vec{a}$. A *solution* to this problem is any program δ' such that:

1. for any pair of situations S, S', if $\mathcal{D} \models Do(\delta', S, S')$, then also $\mathcal{D} \models Do(\delta, S, S')$; and
2. for $(S_i, S'_i) \in \{(S_1, S'_1), \ldots, (S_n, S'_n)\}$:
$$\mathcal{D} \models Do(\delta', S_i, S'_i) \land (\forall s).Do(\delta', S_i, s) \supset s = S'_i$$

That is, a solution to a PbD problem is a program that does not admit any executions not admitted by the original program, but does (at least) admit the given set of demonstrated executions. Intuitively, δ is a non-deterministic Golog program describing a space of possible programs, and δ' is a specialization of this program that enforces behavior according to the given examples. We show how such a solution can be obtained via simulation of the given program over the demonstrations while keeping track of the non-deterministic choices being made during this simulation. Note that even though we assume the target program to be deterministic, a solution to a PbD problem is not necessarily deterministic. This is in particular the case when not enough demonstrations were given to rule out any remaining uncertainty.

5.1 Defining Program Refinement

Given a starting situation S and a program δ, the originally intended use of Golog was to create a constructive proof that $\mathcal{D} \models \exists s'.Do(\delta, S, s')$, hence obtaining as a side-effect a situation term s' that is a sequential *plan* for how the program can be executed successfully. We, however, will use the semantics in a different way: given *two* situation terms S, S' and a program δ with non-determinism, verify that executing δ starting in S is possible and can result in S'. In doing so, we heavily exploit the logical underpinnings of the presented framework, in order to actively exploit the content of S' to infer how decisions need to be made, rather than following a trial-and-error approach.

Recall that situations are sequences of actions and also, given a basic action theory $\mathcal{D}$, completely describe the state of the world. Therefore, S and S' can be used to represent a demonstration. Hence, without any modification necessary, the existing Golog semantics can be used to *verify* that a given program is consistent with a given demonstration. This verification can be done efficiently, as the sequence of actions in S' guides the interpretation of the program, hence often limiting the amount of search necessary to one-step look-ahead. However, the program can actually also be *refined* during this process such that all choices regarding non-determinism in the program that were necessary in order to explain the given demonstration are recorded. The refined program represents the version space updated with the considered example.

We accomplish this by extending the original Golog semantics as shown in Figure 4, where we use an additional forth argument that "returns" the refined program. Intuitively, the refined program is the same as the original program for all deterministic constructs, and for non-deterministic constructs it contains the specific choice that was made in order to "execute" the program. The latter, however, is only the case for those non-deterministic choices that are actually visited. For instance, choices occurring in the 'then' or the 'else' branch of an if-then-else, are only resolved if that branch applied to the considered example. Choices not visited during program execution are left as is. In the next section we will see that this refinement can be used to identify those specializations of the original program that are consistent with a given program demonstration. This is accomplished by fixing the second situation term (s'), which forces the program to execute in compliance with the actions in that situation term. Hence, by keeping track of the choices made during execution, we obtain a program that could be used to reproduce the demonstration described by the given s'. Further, since only those choices are made that are actually required to execute the program, no candidate programs are ruled out.

5.2 Computing Program Refinement

Intuitively, the new, forth argument in Do' "returns" the refined program that results from making the necessary choices during execution of the program in order to reach s' from s. Hence, we can achieve our goal of refining a Golog program δ using a given demonstration (S, S') by constructively proving that:

$$\mathcal{D} \models (\exists\delta').Do'(\delta, S, S', \delta')$$

where $\mathcal{D}$ is the basic action theory describing the domain. This will provide us with one out of potentially several possible refined programs δ'. As with the original Golog, the provided definition of Do' lends itself to a rather straightforward Prolog implementation, casting the problem of constructing this proof into a search problem. This implementation is available on our web site. It can be used to obtain all possible refined programs δ'.

An Extended Example

Let us consider an example from the Wargus domain. In Wargus we repeatedly receive descriptions of the current state of the world from the game engine, which we can model as special actions in the situation calculus, whose effect is to set all fluents according to the information retrieved from the game engine. Hence, demonstrations are sequences of regular actions executed by the user and world-state descriptions that indicate state updates and happen while the user is waiting for orders to be completed. For instance, the action

$$
\begin{aligned}
&Do'(nil, s, s', \delta') \equiv s' = s \wedge \delta' = nil\\
&Do'(a, s, s', \delta') \equiv \mathit{Poss}(a[s], s) \wedge s' = do(a[s], s) \wedge \delta' = a\\
&Do'(\varphi?, s, s', \delta') \equiv \varphi[s]? \wedge s = s' \wedge \delta' = \varphi?\\
&Do'([\delta_1; \delta_2], s, s', \delta') \equiv (\exists s^*, \delta'_1, \delta'_2).Do'(\delta_1, s, s^*, \delta'_1)\wedge\\
&\qquad Do'(\delta_2, s^*, s', \delta'_2) \wedge \delta' = [\delta'_1; \delta'_2]\\
&Do'(\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, s', \delta') \equiv (\exists \delta'').\\
&\qquad (\varphi[s] \wedge Do'(\delta_1, s, s', \delta'') \wedge \delta' = \textbf{if } \varphi \textbf{ then } \delta'' \textbf{ else } \delta_2) \vee\\
&\qquad (\neg\varphi[s] \wedge Do'(\delta_2, s, s', \delta'') \wedge \delta' = \textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta'')\\
&Do'(\textbf{while } \varphi \textbf{ do } \delta, s, s', \delta') \equiv \neg\varphi[s']\wedge\\
&\qquad (\forall P).\big\{(\forall s_1, s_2, s_3, \delta_1, \delta_2, \delta_3)\big[\varphi[s_1] \wedge Do'(\delta_1, s_1, s_2, \delta_2) \wedge\\
&\qquad P(\textbf{while } \varphi \textbf{ do } \delta_2, s_2, s_3, \textbf{while } \varphi \textbf{ do } \delta_3) \supset\\
&\qquad P(\textbf{while } \varphi \textbf{ do } \delta_1, s_1, s_3, \textbf{while } \varphi \textbf{ do } \delta_3)\big] \wedge\\
&\qquad (\forall s_1, \delta_1)\neg\varphi[s_1] \supset P(\textbf{while } \varphi \textbf{ do } \delta_1, s_1, s_1, \textbf{while } \varphi \textbf{ do } \delta_1)\big\}\\
&\qquad \supset P(\delta, s, s', \delta')\\
&Do'((\delta_1|\delta_2), s, s', \delta') \equiv\\
&\qquad Do'(\delta_1, s, s', \delta') \vee Do'(\delta_2, s, s', \delta')\\
&Do'((\pi v)\delta(v), s, s', \delta') \equiv (\exists x)Do'(\delta(x), s,', \delta')\\
&Do'(\delta^*, s, s', \delta') \equiv (\forall P).\{(\forall s_1)P(s_1, s_1, nil)\wedge\\
&\qquad (\forall s_1, s_2, s_3, \delta_1, \delta_2)[Do'(\delta, s_1, s_2, \delta_1) \wedge P(s_2, s_3, \delta_2)\\
&\qquad \supset P(s_1, s_3, \delta_1; \delta_2)]\} \supset P(s, s', \delta')\\
&Do'(P(\vec{x}), s, s', \delta') \equiv \mathit{Proc}(P(\vec{x}), \delta) \wedge Do'(\delta, s, s', \delta')\\
&Do'(\textbf{waitfor } \varphi, s, s', \delta') \equiv\\
&\qquad (\varphi[s] \wedge s' = s \wedge \delta' = \textbf{waitfor } \varphi) \vee\\
&\qquad (\neg\varphi[s] \wedge (\exists x).Do'(\textbf{waitfor } \varphi, do(state(x), s), s', \delta')
\end{aligned}
$$

Figure 4. Axioms for refining programs.

sequence [*attack(1, 5), state(. . .), . . . , state(. . .), attack(1, 6), state(. . .), . . .*] represents a demonstration where the user first ordered agent number 1 to attack agent number 5, then waited for a while as the orders were carried out and the world kept changing. The user then ordered the agent to attack agent number 6, followed by more state updates. We here, and elsewhere in this paper, omit the contents of the lengthy state updates. In addition to this demonstration, the user utters the following instructions:

> *"While there is an opponent who is alive, take the closest-Opponent, attack him, wait until he is dead"*

These instructions can be translated into the following Golog program, which accounts for the ambiguous scope of the while-loop, and the missing argument for the attack action:

```
( while  (∃x) opponent(x) ∧ alive(x) do
      set( o, closestOpponent);
      (πv) [value(v)? ; (attack(v, o) | attack(o, v))];
      waitfor( dead(o))   |
  [while  (∃x) opponent(x) ∧ alive(x) do
      set( o, closestOpponent);
      (πv) [value(v)? ; (attack(v, o) | attack(o, v))]
   waitfor( dead(o)) ] )
```

Let us call this program P_1 and the above action sequence $\vec{A}_1$. Let us further assume that there is a situation S in which there are two opponents left alive, 5 and 6 with 5 being closest right now, and that there is a designated "bestFootman" which in S is agent number 1. Then, upon trying to prove

$$\mathcal{D} \models (\exists \delta').Do'(P_1, S_0, do(\vec{A}_1, S_0), \delta')$$

the algorithm will discover that both branches of the non-deterministic choice in P_1 explain the first action of the sequence (*attack*$(1, 5)$) when using $v =$ *bestFootman* and using the argument ordering (v, o) rather than (o, v). This is because the condition of the while loop evaluates to true in S and o is set to 5. There may be functions, fluents, or constants other than "bestFootman" evaluating to 1 in S and each of them will be considered as a possibility as well. Beyond this first step though, only the first program branch explains the next action in the sequence, *state*$(\dots)$, which is a state update and which can only occurring while the player is waiting. The second branch of the program P_1 is thereby ruled out, as it demands the execution of another attack action, which is inconsistent with the demonstration. The definition of Do' for primitive actions requires the next action in the demonstration to match the action being executed. If also the remainder of the sequence $\vec{A}_1$ can be explained using the first branch of P_1 with the values described above, the refinement will succeed with $\delta' =$

```
while  (∃x) opponent(x) ∧ alive(x) do
    set( o, closestOpponent);
    attack(bestFootman, o);
    waitfor( dead(o))
```

5.3 Formal Properties of Program Refinement

From a formal point of view, there are a few things to note about our definition and computation of refinement: First of all, the cases for if-then-else and while-loops are still in accordance with the original Golog semantics. We here have merely unwound the macro-definition (in terms of $(\delta_1|\delta_2)$ and δ^*). The reason for this is that we want to keep the if-then-else/while-loop in the refined program, rather than replacing them with the specific sequence of actions resulting from resolving the non-deterministic choices for the specific case considered.

Second, since we want to explicitly refer to programs as objects, in order to produce the refined program, we require the reification of programs as objects in the language. This property is shared with the so called transition semantics for Golog, as described by [6], and we assume programs are reified analogously. This also allows us to define the semantics in terms of a predicate rather than macro-expansion.

Finally, by analogy to the original semantics we have that:

Proposition 1. For any two situations S, S' and a Golog program δ without procedures:

$$\mathcal{D} \models Do(\delta, S, S') \text{ iff } \mathcal{D} \models (\exists \delta').Do'(\delta, S, S', \delta')$$

Soundness of Program Refinement

The following theorem states that program refinement leads to provably correct solutions of PbD problems:

Theorem 1. Let $M = \big\langle \delta, \{(S_1, S'_1), \dots, (S_n, S'_n)\} \big\rangle$ be a

PbD problem. Any program δ' such that:

$$\mathcal{D} \models (\exists\, \delta_1, \ldots, \delta_{n-1}).Do'(\delta, S_1, S_1', \delta_1) \wedge \\ Do'(\delta_1, S_2, S_2', \delta_2) \wedge \cdots \wedge Do'(\delta_{n-1}, S_n, S_n', \delta')$$

is a solution to M. — *Proof:* See our technical report [7].

The theorem gives rise to a host of possible algorithms for finding solutions. This includes the common filtering algorithm for updating version spaces: Given the first demonstration, generate the entire set of consistent version spaces (i.e., refined programs δ_1). Then, given subsequent examples, remove from this set all those spaces (programs) that are inconsistent with any of the examples. Note that our use of logic readily realizes the "lazy evaluation" approach that is popular in many version space algebra based approaches to PbD in order to handle infinite version spaces.

Other possible algorithms could follow a depth-first or a best-first search approach. The latter could, for instance, be realized by devising an evaluation function that ranks refined programs by some understanding of likelihood. For example, shorter and/or simpler programs could be explored before more complicated ones are considered. There is a host of related research on situation calculus and Golog from which such specifications and search strategies could be drawn (e.g., [3, 11]).

5.4 Empirical Results

While the ultimate goal of this work is to create a natural interface for users, we do not evaluate the naturalness of our current interface, since this is not germane to the contributions of this paper. Instead, in order to evaluate the utility of our approach for the goal of effectively combining programming by natural instruction with programming by demonstration, we consider a number of example instructions containing common issues arising in natural instruction. We evaluate how well our system can handle these issues, given a single demonstrations of the target program.

Setup

Intuitively, the more omissions and ambiguities instructions contain, the harder it is to refine them using an example demonstration, because there are more possible ways of "explaining" prefixes of the demonstration. While our existing implementation of the refinement algorithm based on the axioms of Section 5.1 has not been optimized, we nevertheless wanted to use it to gauge the empirical performance of our algorithm. To that end, we created a set of 18 example instructions, all describing the program from Section 2 with varying amounts of omissions and ambiguities of all the types described. These instructions were created by manually removing more and more detail from a perfect description of the target program. Example 1 shows one of the 18 instructions we used.

We then translated these instructions into Golog programs which we then refined using the previously described demonstration of the target program (cf. Figure 1). The demonstration and the complete set of instructions used in the experiments together with their automatically generated Golog

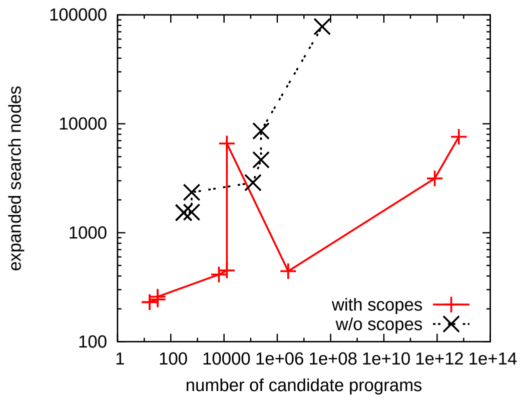

Figure 5. Number of search nodes expanded in program refinement vs. number of candidate programs described by instruction (log-log scale).

representation and the refined program produced by our algorithm can be found on our web site [1]. In all cases, the refined program was deterministic and equal or equivalent to the intended target program.

Results

Figure 5 shows the results. Each point in the plot corresponds to one example instruction. The x-axis denotes the number of programs that are consistent with the instruction. These are the candidate programs, each of which might be the program that was intended by the user. This set of candidates is compactly represented by the Golog program that we generate from the human instruction, as described in Section 4. On the y-axis we show the number of search nodes expanded during refinement of the Golog program, guided by the example demonstration. To improve readability, we have split the test cases (input sentences) into two classes, shown in the figure using two different lines. In the first class all sentences use explicit scoping of instruction blocks, such as loops, while the second class contains the same sentences but with all scoping information removed. All experiments were run on an Intel Xeon CPU, at 2.66GHz. Memory consumption was negligible due to the use of a depth-first based search approach.

From the figure, we can conclude that the amount of omissions and ambiguities in terms of action names, action arguments, and their ordering, increases the amount of search required, as one would expect. In addition, we can observe that missing scoping information over-proportionally increases the amount of required search in order to refine the non-deterministic program representing the natural instruction. While removing scoping naturally increases the number of possible target programs, it seems that its absence from the instructions is particularly difficult to compensate for using a demonstration. In other words, demonstrations seem very effective in disambiguating between possible actions, action arguments, ambiguous references and the like, but do not provide much guidance in terms of program structure. This is consistent with the general insight that demonstrations are poorly suited for describing complex program structure.

Overall, however, we can observe that our approach is able to effectively and efficiently exploit the information contained in an example demonstration to identify a consistent candidate program. In some cases, our approach only had to expand 7593 search nodes to identify a program that is consistent with both the human instruction and the demonstration, out of 6.554 trillion candidate program that would be consistent with the human instructions alone.

So far our algorithm only uses naive depth-first search. Its performance in terms of considered nodes could hence still be improved by drawing on the insights from the planning and search literature. In particular heuristic search approaches might improve performance by guiding the search to explore most likely candidate programs first.

6. INTERSECTING VERSION SPACES

Thus far we have made the assumption that demonstrations are given directly to our framework, and that only within our framework the learning from examples takes place. In practice, it would, however, be extremely valuable if we could combine our framework with existing algorithms for learning from examples as well. To this end, in this section we consider the problem of *intersecting* version spaces symbolically. The problem can be stated as follows: Given two version spaces for the same target program, determine a new version space that contains all and only those program executions that were contained by each of the two given spaces. In terms of our program based representation of version spaces, we can define the problem more precisely:

Definition 2 (Program Intersection)**.** Let δ_1, δ_2 be two Golog programs over action theory $\mathcal{D}$. The Golog program δ' is called an *intersection of* δ_1, δ_2 if for any two situations S, S' such that $S' = do(\vec{a}, S)$ for some sequence of actions $\vec{a}$: $\mathcal{D} \models Do(\delta', S, S')$ iff $\mathcal{D} \models Do(\delta_1, S, S')$ and $\mathcal{D} \models Do(\delta_2, S, S')$, and there is at least one such pair S, S' such that $\mathcal{D} \models Do(\delta_1, S, S')$ and $\mathcal{D} \models Do(\delta_2, S, S')$.

As an example, consider a situation where the user is teaching how to behave when a single footman, denoted “F”, encounters a single opponent footman, “O”. The user utters: *“if strength of f is greater than strength of o, attack, else retreat”* and also demonstrates a few examples, where he always either attacked, or moved away from the opponent. The instruction omit the “attack” arguments, and use the unknown term “retreat”. This is captured by the program: **if** $\mathit{strength}(F) > \mathit{strength}(O)$ **then** $(\pi x, y)\mathit{attack}(x, y)$ **else** $((\pi x, y)\mathit{move}(x, y)|(\pi x, y)\mathit{build}(x, y)|(\pi x, y)\mathit{attack}(x, y))$. Assume that some PbD algorithm was able to generalize the demonstrations to the program: $(\mathit{attack}(F, O)|\mathit{move}(F, \mathit{pos}(F) - (\mathit{pos}(O) - \mathit{pos}(F))))$, where $\mathit{pos}(x)$ denotes the x-y-position of agent x. That is, the learner learned the concept of “away from the opponent”, but was not able to learn under which circumstances to do the one vs. the other. When taken together, one can of course find that that the correct program is: **if** $\mathit{strength}(F) > \mathit{strength}(O)$ **then** $\mathit{attack}(F, O)$ **else** $\mathit{move}(F, \mathit{pos}(F) - (\mathit{pos}(O) - \mathit{pos}(F)))$. This is what is achieved by the approach we define in this section for *synchronizing* two Golog programs.

We conjecture that the problem of (constructively) proving the existence of an intersection for general Golog programs is undecidable.[2] Therefore, we only consider a restricted form of Golog programs without while–loops and recursive procedures, and where (bounded) non-deterministic iteration is only allowed over primitive actions and expressions of the form $(\pi v)a(v)$ or $(\pi v)v$, i.e., non-deterministic choice of action arguments or actions themselves. For clarity, we refer to this language as Golog$^-$. For this class of programs we define the set of axioms shown in Figure 6, denoted Σ_{sync}, regarding a new predicate *sync* that can be used to constructively prove the existence of an intersection of two programs, i.e., generate a program that represents the intersection of the version spaces. The resulting representation is very compact, as it uses the non-deterministic constructs of Golog to represent the space of programs still considered to be a candidate for the target program. For ease of presentation we assume—without loss of generality—that all programs are sequences (e.g., $[A; nil]$ instead of A). We use the notation $\mathit{nondet}(Z)$, where Z is a set, to denote the non-deterministic choice between the elements of the set, i.e., for $Z = \{z_1, \ldots, z_n\}$, $\mathit{nondet}(Z) \overset{\text{def}}{=} (z_1 \mid \cdots \mid z_n)$. Our restrictions on non-deterministic iteration ensure that this set always turns out to be finite[3].

The axioms lead to a sound and complete means for computing program intersections, as stated by the following theorem. We have implemented but not yet evaluated this.

Theorem 2. Let δ_1, δ_2 be two Golog$^-$ programs over some action theory. If there exists a Golog$^-$ program δ' such that $\Sigma_{\mathit{sync}} \models \mathit{sync}(\delta_1, \delta_2, \delta')$, then it is an intersection of δ_1, δ_2. Further, if there exists an intersection of δ_1, δ_2, then there is a δ' s.t. $\Sigma_{\mathit{sync}} \models \mathit{sync}(\delta_1, \delta_2, \delta')$. — *Proof:* See [7].

Note that the theorem does not state that all possible intersections can be identified using *sync*. This is because there may be infinitely many possible intersections, all of which characterizing the same sub-set of the version space. Instead, only one such representation is found. This, however, does not cause practical limitations.

7. DISCUSSION

We have presented a framework for integrating programming by natural language instruction and programming by demonstration. The key idea was to represent ambiguous human instruction as non-deterministic programs, and then refine these programs via simulation of demonstrations. We have evaluated our framework in the Wargus domain, illustrating the types of omissions and ambiguities commonly occurring in human instructions that our framework can resolve when provided with an example. Our empirical results show that the framework is practically effective and efficient. We further presented an approach for combining the symbolic outputs of different learners via program intersection.

Our ultimate goal is to facilitate programming by end-users. Therefore we plan to do user studies to evaluate the nat-

[2]Our intuition stems from the problem’s similarity to the difficult problem of determining program equivalence.

[3]Intuitively this is because δ^*’s—which are the only source of infinity—are unified when they “meet” in *sync*.

$$
\begin{aligned}
&\mathit{sync}(\delta_1, \delta_2, \delta') \equiv \mathit{sync}'(\delta_1, \delta_2, \delta') \vee \mathit{sync}'(\delta_2, \delta_1, \delta') \\
&\mathit{sync}'(nil, nil, y) \equiv y = nil \\
&\mathit{sync}'([a; b_1], [a; b_2], y) \equiv (\exists z).\mathit{sync}(b_1, b_2, z) \wedge y = [a; z] \\
&\mathit{sync}'([\varphi?; x], \delta_2, y) \equiv (\exists z).\mathit{sync}(x, \delta_2, z) \wedge y = [\varphi?; z] \\
&\mathit{sync}'([\textbf{if } \varphi \textbf{ then } \delta_a \textbf{ else } \delta_b; x], \delta_2, y) \equiv \\
&\qquad \big((\exists z_a).\mathit{sync}([\delta_a; x], \delta_2, z_a) \wedge \\
&\qquad\quad ((\exists z_b).\mathit{sync}([\delta_b; x], \delta_2, z_b) \wedge y = \textbf{if } \varphi \textbf{ then } z_a \textbf{ else } z_b) \vee \\
&\qquad\quad ((\nexists z_b).\mathit{sync}([\delta_b; x], \delta_2, z_b) \wedge y = [\varphi?; z_a])\big) \vee \\
&\qquad \big((\nexists z_a).\mathit{sync}([\delta_a; x], \delta_2, z_a) \wedge \\
&\qquad\quad (\exists z_b).\mathit{sync}([\delta_b; x], \delta_2, z_b) \wedge y = [\neg\varphi?; z_b]\big) \\
&\mathit{sync}'([(\delta_a | \delta_b); x], \delta_2, y) \equiv \\
&\qquad \big((\exists z_a).\mathit{sync}([\delta_a; x], \delta_2, z_a) \wedge \\
&\qquad\quad ((\exists z_b).\mathit{sync}([\delta_b; x], \delta_2, z_b) \wedge y = (z_a | z_b)) \vee \\
&\qquad\quad ((\nexists z_b).\mathit{sync}([\delta_b; x], \delta_2, z_b) \wedge y = z_a)\big) \vee \\
&\qquad \big((\nexists z_a).\mathit{sync}([\delta_a; x], \delta_2, z_a) \wedge (\exists z_b).\mathit{sync}([\delta_b; x], \delta_2, z_b) \wedge y = z_b\big) \\
&\mathit{sync}'([(\pi v)\delta(v); x], \delta_2, y) \equiv ((\exists v', x').\delta_2 = [(\pi v')\delta(v'); x'] \wedge \\
&\qquad (\exists \delta').\mathit{sync}(x, x', \delta') \wedge y = [(\pi v)\delta(v); \delta']) \vee \\
&\qquad \big((\nexists v', x').\delta_2 = [(\pi v')\delta(v'); x'] \wedge (\exists Z).y = \mathit{nondet}(Z) \wedge \\
&\qquad Z \neq \emptyset \wedge (\forall z).z \in Z \equiv (\exists q).\mathit{sync}([\delta(q); x], \delta_2, z)\big) \\
&\mathit{sync}'([a(v)^*; x], \delta_2, y) \equiv \big((\exists x').\delta_2 = [a(v)^*; x'] \wedge \\
&\qquad (\exists \delta').\mathit{sync}([a(v)^*; x], x', \delta') \wedge y = [a(v)^*; \delta']\big) \vee \\
&\qquad (\exists x').\delta_2 = [((\pi v')a(v'))^*; x'] \wedge \\
&\qquad (\exists \delta').\mathit{sync}([a(v)^*; x], x', \delta') \wedge y = [a(v)^*; \delta']\big) \vee \\
&\qquad \big((\nexists x')\delta_2 = [a(v)^*; x'] \wedge (\nexists x')\delta_2 = [(\pi v)a(v)^*; x'] \wedge \\
&\qquad (\exists Z).y = \mathit{nondet}(Z) \wedge Z \neq \emptyset \wedge \\
&\qquad (\forall z).z \in Z \equiv (\mathit{sync}([a(v); a(v)^*; x], \delta_2, z) \vee \mathit{sync}(x, \delta_2, z))\big) \\
&\mathit{sync}'([((\pi v')a(v'))^*; x], \delta_2, y) \equiv \big((\exists x').\delta_2 = [a(v)^*; x'] \wedge \\
&\qquad (\exists \delta').\mathit{sync}([a(v)^*; x], x', \delta') \wedge y = [a(v)^*; \delta']\big) \vee \\
&\qquad (\exists x').\delta_2 = [((\pi v)a(v))^*; x'] \wedge y = [((\pi v')a(v'))^*; \delta'] \wedge \\
&\qquad (\exists \delta').\mathit{sync}([((\pi v')a(v'))^*; x], x', \delta')\big) \vee \\
&\qquad \big((\nexists x')\delta_2 = [a(v)^*; x'] \wedge (\nexists x')\delta_2 = [(\pi v)a(v)^*; x'] \wedge \\
&\qquad (\exists Z).y = \mathit{nondet}(Z) \wedge Z \neq \emptyset \wedge \\
&\qquad (\forall z).z \in Z \equiv (\mathit{sync}([((\pi v')a(v')); ((\pi v')a(v'))^*; x], \delta_2, z) \vee \\
&\qquad \mathit{sync}(x, \delta_2, z))\big)
\end{aligned}
$$

Figure 6. The set of axioms Σ_{sync}.

uralness of our natural-language interface and how effective end-users can program using the described combination of instruction and demonstration. In [9] we present results regarding the use of natural instruction for designing new workflows in an interactive user interface. In that paper, our focus was more on the user experience, while in this one we focused on expressiveness of the language and the integration with demonstrations. In future work we would like to merge the results of this work. Other possible directions for future work include: considering the feasibility of PbD under incomplete knowledge of the state, or in the face of uncertain action effects, and investigating the effects of the availability of detailed background knowledge about the domain and its actions.

8. REFERENCES

1. `www.isi.edu/ikcap/wargus/`.
2. J. F. Allen, N. Chambers, G. Ferguson, L. Galescu, H. Jung, M. D. Swift, and W. Taysom. Plow: A collaborative task learning agent. In *Proc. of the 22nd AAAI Conf. on Artificial Intell.*, pages 1514–1519, 2007.
3. F. Bacchus, J. Y. Halpern, and H. J. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artif. Intell.*, 111(1-2), 1999.
4. J.-H. Chen and D. S. Weld. Recovering from errors during programming by demonstration. In *Proc. of the 2008 Intl. Conf. on Intell. User Interfaces (IUI)*, pages 159–168, 2008.
5. P. Clark, J. Thompson, K. Barker, B. Porter, V. Chaudhi, A. Rodriguez, J. Thomere, S. Mishra, Y. Gil, P. Hayes, and T. Reichherzer. Knowledge entry as graphical assembly of components. In *Proc. of the 1st Intl. Conf. on Knowledge Capture (K-CAP)*, 2001.
6. G. De Giacomo, Y. Lespérance, and H. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artif. Intell.*, 121(1–2), 2000.
7. C. Fritz and Y. Gil. Towards the integration of programming by demonstration and programming by instruction using golog: Extended version with proofs. In *AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR)*, 2010.
8. Y. Gil. Human tutorial instruction in the raw. 2010. Submitted for publication. Available at: `www.isi.edu/~gil/papers/gil-ker10.pdf`.
9. Y. Gil, V. Ratnakar, and C. Fritz. Tellme: Learning procedures from tutorial instruction. In *Proc. of the 2011 Intl. Conf. on Intell. User Interfaces (IUI)*, 2011.
10. Y. Gil, V. Ratnakar, J. Kim, P. A. Gonzalez-Calero, P. Groth, J. Moody, and E. Deelman. Wings: Intelligent workflow-based design of computational experiments. In *IEEE Intelligent Systems*, 2010.
11. H. Grosskreutz and G. Lakemeyer. Turning high-level plans into robot programs in uncertain domains. In *Proc. of the 14th European Conf. on Artificial Intell. (ECAI)*, pages 548–552, 2000.
12. S. B. Huffman and J. E. Laird. Flexibly instructable agents. *J. Artif. Intell. Res. (JAIR)*, 3:271–324, 1995.
13. E. S. Kim, D. Leyzberg, K. M. Tsui, and B. Scassellati. How people talk when teaching a robot. In *Proc. of the 4th ACM/IEEE Intl. Conf. on Human Robot Interaction (HRI)*, pages 23–30, 2009.
14. T. A. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1-2), 2003.
15. H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.
16. H. Lieberman, editor. *Your wish is my command: programming by example*. Morgan Kaufmann, San Francisco, CA, USA, 2001.
17. R. Maclin, J. W. Shavlik, L. Torrey, T. Walker, and E. W. Wild. Giving advice about preferred actions to reinforcement learners via knowledge-based kernel regression. In *Proc. of the 20th National Conf. on Artificial Intell. (AAAI)*, pages 819–824, 2005.
18. S. McIlraith. Explanatory diagnosis: Conjecturing actions to explain observations. In *Proc. of the Sixth Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR)*, pages 167–177, 1998.
19. D. Oblinger, V. Castelli, and L. D. Bergman. Augmentation-based learning: combining observations and user edits for programming-by-demonstration. In *Proc. of the 2006 Intl. Conf. on Intell. User Interfaces (IUI)*, pages 202–209, 2006.
20. R. Reiter. *Knowledge in Action*. MIT Press, Cambridge, MA, USA, 2001.
21. A. L. Thomaz and C. Breazeal. Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artif. Intell.*, 172(6-7), 2008.