Towards the Integration of Programming by Demonstration and Programming by Instruction using Golog

Christian Fritz and **Yolanda Gil**
Information Sciences Institute
University of Southern California
Marina del Rey, California. USA.
{fritz,gil}@isi.edu

Abstract

We present a formal approach for combining programming by demonstration (PbD) with programming by instruction (PbI)—a largely unsolved problem. Our solution is based on the integration of two successful formalisms: version space algebras and the logic programming language Golog. Version space algebras have been successfully applied to programming by demonstration. Intuitively, a version space describes a set of candidate procedures and a learner filters this space as necessary to be consistent with all given demonstrations of the target procedure. Golog, on the other hand, is a logical programming language defined in the situation calculus that allows for the specification of non-deterministic programs. While Golog was originally proposed as a means for integrating programming and automated planning, we show that it serves equally well as a formal framework for integrating PbD and PbI. Our approach is the result of two key insights: (a) Golog programs can be used to define version spaces, and (b) with only a minor augmentation, the existing Golog semantics readily provides the update-function for such version spaces, given demonstrations. Moreover, as we will show, two or more programs can be symbolically *synchronized*, resulting in the *intersection* of two, possibly infinite, version spaces. The framework thus allows for a rather flexible integration of PbD and PbI, and in addition establishes a new connection between two active research areas, enabling cross-fertilization.

1 Introduction

In large parts, AI is concerned with the problem of making machines behave in particular ways intended by users. The problem is challenging because the intended behaviors can be very complex, it can be difficult to humans, in particular non-programmers, to describe the correct behavior explicitly, and in many cases there are plenty of exceptions or special circumstances to consider. Therefore life-long learning is the norm and there always remains some uncertainty as to what the right behavior should be.

Programming by demonstration (PbD; e.g., (Lieberman 2001)) has shown a lot of promise in end-user programming. In PbD a user demonstrates the intended behavior to the system on a number of specific examples, and the system in turn tries to learn the correct behavior from this. This allows a much larger group of people to "program", as it does not demand specialized programming skills of the user. The shortcomings of PbD, however, are 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.

On the other hand there are approaches that allow users to program by explicit instruction (PbI). Using, say, natural language a user can provide direct input regarding particular parts of the target program or its high-level structure. It is hence easy to express complex program structure. The disadvantages, however, are that this type of instruction is prone to omissions and imprecision—it is easy for a user to forget a condition or a boundary case— and also, as with explicit programming, the user needs to put more effort into structuring the instructions correctly.

Ideally one would want to combine both, PbD and PbI, as they seem to complement each other well. But this integration is challenging. There have been some promising approaches for making PbD systems more robust to errors and allowing user edits interleaved with demonstration of examples. Oblinger, Castelli, and Bergman (2006) describe the augmentation-based learning (ABL) algorithm that is used by the DocWizards system (Prabaker, Bergman, and Castelli 2006). The algorithm allows users to manually edit procedure hypotheses explicitly interleaved with demonstration and ensures that such edits are not undone by future demonstrations. Chen and Weld (2008) present CHINLE, 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. These systems however, limit the types of editing operations the user is allowed to perform and hence only cover part of the spectrum between PbI and PbD. In particular, they do not allow the manual specification of the high-level program structure, which might be most helpful to the PbD part of the system.

The problem is that there is no principled, comprehensive framework for representing, updating, and executing procedure hypotheses. Such a framework should enable any learner to output its acquired knowledge in this form, allow for the combination of (sets of) procedure hypotheses, so

that different input/learning methods can be combined, and enable the execution of partially learned procedures if necessary, hence enabling learning from experience or explicit user feedback as well.

Inspired by the recent approaches for combining PbD with user edits, we propose a formal framework that 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 procedure hypotheses using *nondeterministic programs*. These are used to represent uncertainty about the target procedure being learned. In particular, we use the logical programming language Golog which readily allows us to represent and reason about such nondeterministic programs. Such a program can be understood to represent the entire set of possible deterministic programs described by it. For the purpose of this paper, we assume that there is a learning component available that generates such programs from user instruction. As such, we believe the language to be well suited for this purpose, as its nondeterministic constructs can be used in places where instructions were ambiguous or omitted. 2) 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. 3) Given Golog's formal semantics, we can define a notion of *program synchronization* that provably implements a *symbolic intersection* of (possibly infinite) sets of hypotheses.

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 procedure 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. Perhaps most importantly however, the approach facilitates the reuse of a large body of research regarding situation calculus and Golog that could allows us to extend PbD for other settings, including for environments with only partially observable states, or uncertain action effects.

The contributions of this paper are of mainly theoretic nature rather than describing or evaluating a complete system. A prototype implementation of the framework however exists and will be made available. It could serve as a back-end to systems that focus on the actual user interface and we are actively using and extending this system.

1.1 Motivation

Some possible ways of integrating PbD and PbI are as follows: i) A novice programmer defines the rough, high-level structure of a program using PbI and then starts demonstrating specific examples of the program. She may later go back-and-forth between PbI and PbD to both get the structure but also all specific boundary cases right. ii) An expert programmer designs a template program by instruction that ensures a basic, required behavior. This template is then customized by each end-user individually via demonstration. iii) Two programmers each sketch their (partial) knowledge of the same program using PbI. These two program candidates are then *synchronized*, by which they complement each other and produce a more specific candidate. Similarly, two sets of hypotheses provided by different learning systems could be symbolically combined.

For illustration purposes in this paper, we use examples from a domain in which a robot is being programmed to tidy up a room. One possible, incomplete procedure for this could be stated as follows: *While there remains an item on the floor, we are meant to pick it up and then, if it is a toy, put it in the box, if it is clothing, go to the closet and put it either on a hanger or in the drawer (but we don't yet know which one), and if it is anything else, we don't know what to do.* Then, after receiving a demonstration where a book is put on the shelf, the system may entertain two hypotheses regarding the uncertainty about other objects: (a) that everything else goes on the shelf, or that (b) there are more conditions to consider. We will use this as a running example.

We begin by reviewing the situation calculus and Golog. We then show how version spaces can be represented as Golog programs, can be updated using an extension of Golog's semantics, and can be intersected by *synchronizing* two programs. In Section 4 we describe how the presented framework can be used to flexibly integrate PbD and PbI. We discuss related and future work before we conclude.

2 Preliminaries

The situation calculus is a sorted logic for specifying and reasoning about dynamical systems (Reiter 2001). 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. The relation $\sqsubset$ provides an ordering on situations of the same branch, and $s \sqsubseteq s'$ abbreviates $s = s' \vee s \sqsubset s'$. We abbreviate $do(a_n, do(a_{n-1}, \ldots do(a_1, s)))$ 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. Nevertheless, there exist extensions that account for actions with uncertain outcomes (e.g., (Mateus et al. 2001)).

In the situation calculus, background knowledge of a domain (e.g., our cleaning robot domain) 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 (Reiter 2001). We write $s \sqsubset s'$ to say that situation s precedes s' in the tree of situations. This is axiomatized in the foundational axioms. 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., *OnFloor*$(Book_1, S_0)$ and *Book*$(Book_1)$ state that initially there is a book on the floor.

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, $\mathit{Poss}(\mathit{PickNextItem}, s) \equiv (\exists x).\mathit{NextItem}(s) = x \wedge \mathit{OnFloor}(x, s)$.

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

2.1 Golog

Golog (Levesque et al. 1997) is a programming language defined in the situation calculus. It allows a user to specify programs whose set of legal executions defines a subtree of the tree of situations of a basic action theory. Golog has an Algol-inspired syntax extended with flexible *nondeterministic constructs*. Golog programs are created inductively 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'	loops
$(\delta_1 \mid \delta_2)$	non-deterministic choice
$(\pi v)\delta(v)$	non-deterministic choice of argument
δ^*	non-deterministic iteration

In addition, Golog allows 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 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 (Levesque et al. 1997). While deterministic constructs enforce the occurrence of particular actions, non-deterministic constructs define "open parts" that allow for several ways of interpretation/execution. We call a Golog program *deterministic* if it contains neither $(\delta_1|\delta_2)$ nor (πv) nor δ^*.

3 Version spaces as Golog programs

In this section we illustrate how naturally Golog programs can be used to define version spaces and how the existing semantics can be augmented in order to refine a program given an example, hence realizing the update function for the version space. We further define a notion of *program synchronization* that allows us to symbolically intersect version spaces.

Intuitively, in the context of this paper, non-deterministic parts of a Golog program express uncertainty about the precise target procedure being learned and will be filled-in via demonstration. 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., (Lau et al. 2003)). 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. For instance, our cleaning robot example could be specified as:

$$
\begin{aligned}
P_1 \stackrel{\text{def}}{=} \textbf{while } & (\exists x).\mathit{OnFloor}(x) \textbf{ do } [\, \mathit{PickNextItem}; \\
& \textbf{if } \mathit{Toy}(\mathit{Item}) \textbf{ then } \mathit{Put}(\mathit{Item}, \mathit{Box}) \\
& \textbf{else if } \mathit{Clothing}(\mathit{Item}) \textbf{ then } [\mathit{goto}(\mathit{Closet}); \\
& \quad (\mathit{Put}(\mathit{Item}, \mathit{Hanger}) \mid \mathit{Put}(\mathit{Item}, \mathit{Drawer}))] \\
& \textbf{else } \mathit{Process}(\mathit{Item}) \,]
\end{aligned}
$$

We use the (non-deterministic) procedure $\mathit{Process}(x)$, defined as follows, to describe the space of possible (deterministic) procedures of how to stow away any other item:

$$
\begin{aligned}
& \mathit{Process}(x) \stackrel{\text{def}}{=} \\
& \quad (\, \mathit{Stow}(x) \mid \\
& \quad\quad ((\pi c)[\mathit{cond}(c)?; \textbf{if } c \textbf{ then } \mathit{Stow}(x) \textbf{ else } nil])^*) \\
& \mathit{Stow}(x) \stackrel{\text{def}}{=} [((\pi\, l)\mathit{goto}(l))^*;\ (\pi v)\mathit{Put}(x, v)]
\end{aligned}
$$

where $\mathit{cond}(\varphi)$ is a user-defined predicate that defines the set (version space) of conditions to consider. This is common practice in version space algebra based approaches and could, for instance, limit conditions to be simple fluent literals or conjunctions of such of up to a certain length. The system could then conjecture conditions that distinguish the place certain types of items are meant to be stowed at. The auxiliary procedure $\mathit{Stow}(x)$ non-deterministically chooses such a place and this non-determinism gets resolved as more examples are incorporated.

For instance, given the described demonstration about the book being put on the shelf, the above described hypothesis (a) would replace $\mathit{Process}(\mathit{Item})$ by $\mathit{Put}(\mathit{Item}, \mathit{Shelf})$, which is one possible option of realizing the procedures, choosing the $\mathit{Stow}(x)$ branch, not doing any *goto*'s, and then choosing *Shelf* for v). Hypothesis (b), on the other hand, would replace it with

$$
\begin{aligned}
& [\, \mathit{cond}(\mathit{Book}(\mathit{Item}))?; \\
& \textbf{if } \mathit{Book}(\mathit{Item}) \textbf{ then } \mathit{Put}(\mathit{Item}, \mathit{Shelf}) \textbf{ else } nil; \\
& ((\pi c)[\mathit{cond}(c)?; \textbf{if } c \textbf{ then } \mathit{Stow}(x) \textbf{ else } nil])^*]
\end{aligned}
$$

which is obtained by choosing the second branch in the non-deterministic choice of the $\mathit{Process}(x)$ procedure.

We are now ready to define the problem of programming by demonstration and what counts as a solution. We assume

the target procedure to be deterministic. However, in the absence of sufficient training data, we may not be able to specify all the details of this target procedure and hence still end up with a non-deterministic procedure, after considering a number of examples. This is reflected in the following definition.

Definition 1. 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 $S_i \sqsubset S_i'$. A *solution* to this problem is any 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 $(S_i, S_i') \in \{(S_1, S_1'), \ldots, (S_n, S_n')\}$:

$$\mathcal{D} \models Do(\delta', S_i, S_i') \wedge (\forall s).Do(\delta', S_i, s) \supset s = S_i'$$

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, in the definition, δ is a (usually non-deterministic) Golog program describing a space of possible procedures, and δ' is another Golog program that is a specialization of δ and in particular ensures that the program behaves 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 made during the simulation. Note that even though we assume the target procedure 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. Nevertheless, Golog's semantics readily provides the means for executing such non-deterministic programs, if necessary.

3.1 Resolving uncertainty in Golog programs

Given a starting situation S and a program δ, the originally intended use of Golog was to create a constructive proof for the query $\mathcal{D} \models \exists s'.Do(\delta, S, s')$, 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.

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 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

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

Figure 1: Axioms for refining programs.

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 1, 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 execution. 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.

There are a few things to note: First, note that 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 (De Giacomo, Lespérance, and Levesque 2000), and we assume programs are reified analogously. This also allows us to define the semantics in terms of a predicate rather than macro-expansion. Third, we do not consider recursive procedures. Also, the reification of programs allows us to assume that procedures are defined in the background knowledge using the relation $\mathit{Proc}(P(\vec{x}), \delta)$, where P denotes the procedure name, $\vec{x}$ the formal arguments, and δ the body, which may mention elements from $\vec{x}$. Finally, the following property follows from analogy to the original semantics.

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

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

3.2 Updating version spaces

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, in order to refine a Golog program δ using a given demonstration (S, S') we constructively prove the query:

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

Note that there may be several such 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 a proof into a search problem. We will make such an implementation available on our web site. It can be used to obtain all possible refined programs δ'.

As an example of how this might look in practice, consider our example program P_1 from the previous section, a situation S where there is only one book on the floor, and the action sequence $\mathit{PickNextItem}, \mathit{Put}(\mathit{Item}, \mathit{Shelf})$. Then one can verify using above definitions that the following holds:

$$\mathcal{D} \models Do'(P_1, S, do([\mathit{PickNextItem}, \mathit{Put}(\mathit{Item}, \mathit{Shelf})], S), P_2)$$

for P_2 being like P_1 but with $\mathit{Process}(Item)$ replaced according to any one of the two previously described hypotheses (a) or (b).

The resulting programs can be further refined through demonstration or manual editing. Regarding the former, one can show that this leads to solutions of PbD problems[1].

Theorem 1. Let $M = \langle \delta, \{(S_1, S_1'), \ldots, (S_n, S_n')\}\rangle$ 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.

[1] All proofs will be provided in the final version of this paper or a technical report.

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., (Bacchus, Halpern, and Levesque 1999; Bienvenu, Fritz, and McIlraith 2006; Grosskreutz and Lakemeyer 2000).

3.3 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 may, however, be interesting to consider the input from other learners. For instance, for a certain domain a particular learning algorithm may be known to provide better generalization than the version space approach underlying our framework. To also enable the combination of PbI with such external algorithms, and hence loosen our dependence on the specific approach for learning from examples assumed in the previous section, we here consider the problem of *intersecting* version spaces. The problem can be stated as follows: Given two version spaces for the same target procedure, 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. 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' with $S \sqsubset S'$: $\mathcal{D} \models Do(\delta', S, S')$ iff $\mathcal{D} \models Do(\delta_1, S, S')$ and $\mathcal{D} \models Do(\delta_2, S, S')$.

To understand the intuitive use of this, consider two programs, both describing some partial knowledge of the same target procedure of going to another room x:

$$[\mathit{leaveCurrentRoom}; ((\pi v)v)^*]$$
$$[((\pi z)z)^*; \mathit{enterRoom}(x)]$$

One possible, and in fact the most general, intersection of these is $[\mathit{leaveCurrentRoom}; ((\pi v)v)^*; \mathit{enterRoom}(x)]$.

We conjecture that the problem of (constructively) proving the existence of an intersection is undecidable for general Golog programs and do not consider it any further.[2]

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

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

Figure 2: The set of axioms Σ_{sync}.

Instead 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 can define the set of axioms shown in Figure 2, 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. This representation is very compact, as it uses the non-deterministic constructs of Golog to represent the space of procedures still considered to be a candidate for the target procedure. 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 $Z = \{z_1, \ldots, z_n\}$, $\mathit{nondet}(Z) \stackrel{\text{def}}{=} (z_1 \mid \cdots \mid z_n)$. Our restrictions on non-deterministic iteration ensure that this set always turns out to be finite[3]. This set of axioms again can be implemented and reasoned about efficiently in Prolog as in contained in our implementation.

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. $\Sigma_{\mathit{sync}} \models \mathit{sync}(\delta_1, \delta_2, \delta')$.

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.

4 Example

To demonstrate how the presented framework can be used to flexibly combine PbD and PbI we consider an extended example. We assume that the correct, intended target procedure is as follows (or equivalent to):

$$
\begin{aligned}
P_1 = \textbf{while } &(\exists x).\mathit{OnFloor}(x) \textbf{ do } \big[\, \mathit{PickNextItem};\\
&\textbf{if } \mathit{Toy}(\mathit{Item}) \textbf{ then } \mathit{Put}(\mathit{Item}, \mathit{Box})\\
&\textbf{else if } \mathit{Clothing}(\mathit{Item}) \textbf{ then}\\
&\quad [\mathit{goto}(\mathit{Closet}); \mathit{Put}(\mathit{Item}, \mathit{Hanger})]\\
&\textbf{else if } \mathit{Book}(\mathit{Item}) \textbf{ then } \mathit{Put}(\mathit{Item}, \mathit{Shelf})\\
&\textbf{else } [\mathit{goto}(\mathit{Bin}); \mathit{Put}(\mathit{Item}, \mathit{Bin})]\,\big]
\end{aligned}
$$

Following the approach (i) of Section 1.1 the user may first provide the rough structure of the program,

$$
\begin{aligned}
P_2 \quad = \quad &\textbf{while } (\exists x).\mathit{OnFloor}(x) \textbf{ do}\\
&\qquad [\mathit{PickNextItem}; \mathit{Process}(x)]
\end{aligned}
$$

where $\mathit{Process}(x)$ is the procedure defined in Section 3.2. To further limit the scope of conditions, she may define as part of the background theory $\mathit{cond}(c) \equiv (\exists f, o).\mathit{Fluent}(F) \wedge c = f(o)$ and ensure that fluents only hold for finitely many objects. Given this, she can proceed by demonstrating what to do for individual items, by showing situations where only one item is lying on the floor. Consider a situation S_1 where only one shirt is on the floor, and the demonstration $S_1' = \mathit{do}([\mathit{PickNextItem}; \mathit{goto}(\mathit{Closet}); \mathit{Put}(\mathit{Item}, \mathit{Hanger})], S_1)$. Using our Prolog implementation we can prove that $\mathcal{D} \models (\exists \delta').\mathit{Do}'(P_1, S_1, S_1', \delta')$ holds for a (finite) number of programs δ', all of which are like P_1 but replace the $\mathit{Process}(x)$ by different sub-programs: one program replaces it by $[\mathit{PickNextItem}; \mathit{goto}(\mathit{Closet}); \mathit{Put}(\mathit{Item}, \mathit{Hanger})]$, all others replace it by

$$
\begin{aligned}
&\big[\textbf{if } F(O)\\
&\quad \textbf{then } [\mathit{PickNextItem}; \mathit{goto}(\mathit{Closet}); \mathit{Put}(\mathit{Item}, \mathit{Hanger})]\\
&\quad \textbf{else } nil;\\
&((\pi c)[\mathit{cond}(c)?; \textbf{if } c \textbf{ then } \mathit{Stow}(x) \textbf{ else } nil])^*\big]
\end{aligned}
$$

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

for some fluent F and some object O, such that $\mathcal{D} \models F(O)[do(\textit{PickNextItem}, S)]$. That is, any fluent literal that is true in the demonstration after picking up the item is a candidate for the condition. Many of those will be found inconsistent as soon as a second demonstration, regarding a different type of item, is considered. Any such program can be readily executed, randomly picking a place to store any item not covered by the hypothesized condition (if the chosen condition at all regards the current item).

5 Discussion

We have described a formal framework that implements a version space algebra approach to PbD as an extension of the logic programming language Golog. This enables the integration of programming by demonstration with explicit programming by instruction and further allows for the symbolic intersection of version spaces. Our use of Golog also facilitates extending PbD in other ways. For instance, one might use planning in order to resolve some of the nondeterminism remaining in a procedure hypothesis if it needs to be executed before a deterministic target procedure has been learned. Also learning from experience is facilitated.

Finally, also Blythe et al. (2007) aim to integrate PbD and PbI, though, focusing on a particular domain. Unfortunately, the exposition does not provide sufficient technical detail to understand the extend to which the goal of integration has been achieved in that work in generality and to compare the described system to our framework.

Other possible directions for future work include: the study of other algorithms for searching the space of hypotheses in different ways, e.g., guided by some specification of most likely choices in the program, or based on heuristics; considering the feasibility of PbD under incomplete knowledge of the state, or in the face of uncertain action effects; investigating the effects of the availability of detailed background knowledge about the domain and its actions; and considering environments with continuous state and action domains and possibly durative actions. In all of these areas it might be possible to readily build on existing work to extend the scope of PbD accordingly.

Acknowledgements

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

References

Bacchus, F.; Halpern, J. Y.; and Levesque, H. J. 1999. Reasoning about noisy sensors and effectors in the situation calculus. *Artifical Intelligence* 111(1-2):171–208.

Bienvenu, M.; Fritz, C.; and McIlraith, S. A. 2006. Planning with qualitative temporal preferences. In *Proc. KR*, 134–144.

Blythe, J.; Kapoor, D.; Knoblock, C.; Lerman, K.; and Minton, S. 2007. Learning information-gathering procedures by combined demonstration and instruction. In *Proc. of the AAAI 2007 Workshop on Learning Planning Knowledge*.

Chen, J.-H., and Weld, D. S. 2008. Recovering from errors during programming by demonstration. In *Proc. IUI*, 159–168.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1–2):109–169.

Grosskreutz, H., and Lakemeyer, G. 2000. Turning high-level plans into robot programs in uncertain domains. In *Proc. ECAI*, 548–552.

Lau, T. A.; Wolfman, S. A.; Domingos, P.; and Weld, D. S. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53(1-2):111–156.

Levesque, H. J.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1-3):59–83.

Lieberman, H., ed. 2001. *Your wish is my command: programming by example*. San Francisco, CA, USA: Morgan Kaufmann.

Mateus, P.; Pacheco, A.; Pinto, J.; Sernadas, A.; and Sernadas, C. 2001. Probabilistic situation calculus. *Annals of Mathematics and Artificial Intelligence* 32(1-4):393–431.

Oblinger, D.; Castelli, V.; and Bergman, L. D. 2006. Augmentation-based learning: combining observations and user edits for programming-by-demonstration. In *Proc. IUI*, 202–209.

Prabaker, M.; Bergman, L. D.; and Castelli, V. 2006. An evaluation of using programming by demonstration and guided walkthrough techniques for authoring and utilizing documentation. In *Proc. CHI*, 241–250.

Reiter, R. 2001. *Knowledge in Action*. Cambridge, MA, USA: MIT Press.