Interactive Bootstrapped Learning for End-User Programming

Michael Freed
freed@ai.sri.com
SRI International, Inc.

Daniel Bryce
daniel.bryce@usu.edu
Utah State University

Jiaying Shen and Ciaran O'Reilly
{shen,oreilly}@ai.sri.com
SRI International, Inc.

Abstract

End-user programming raises the possibility that the people who know best what a software system should do will be able to customize, remedy original programming defects and adapt systems as requirements change. As computing increasingly enters the home and workplace, the need for such tools is high, but state of practice approaches offer very limited capability. We describe the Interactive Bootstrapped Learning (iBL) system which allows users to modify code by interactive teaching similar to human instruction. It builds on an earlier system focused on exploring how machine learning can be used to compensate for limited instructional content. iBL provides an end-to-end solution in which user-iBL dialog gradually refines a hypothesis about what transformation to a target code base will best achieve user intent. The approach integrates elements of many AI technologies including machine learning, dialog management, AI planning and automated model construction.

Introduction

Complex operating environments such as homes or workplaces present especially difficult software engineering challenges. Requirements are often imperfectly understood, so initial design is usually faulty. Operating needs can change, so initially correct design decisions might become invalid. Development processes are typically rigorous and time-consuming, so modifying software can be very costly.

Allowing end-users to extend and modify software that they use on a daily basis offers a potential solution. However, state of practice end-user programming (EUP) methods typically provide very limited control. For example, macro recorders in spreadsheet software and rule frameworks used in many email clients allow users to automate some otherwise manual behaviors, but do not allow users to modify existing software behavior. Other approaches make it relatively easy for users to write structured code (e.g. visual programming (Shu, 1999) and some natural programming (Myers, Pane, and Ko, 2004) and automated programming (Pecheur, Andrews, and Nitto, 2010) methods). But these require formal expression of user intent (desired system behavior) that does not scale well to complex applications.

The idea of Programming By Instruction (PBI) is to give users ways to express intended software behavior similar to those they would use to teach a person. MABLE (Mailler et al., 2009) is a first-generation PBI architecture that supports three forms of instruction. Teaching by Telling involves direct expression of instructional content, but may omit a great deal of information that MABLE must retrieve or infer from background knowledge. Teaching by Example (demonstration or indicated instances) is useful when telling would be onerous or involves hard-to-express concepts. Teaching by Feedback is useful when the system can generate approximately correct behavior.

As described by Mailler et al. (2009), MABLE was designed to explore the use of current-generation machine learning methods to address a key problem in PBI: specifying concepts, corresponding to code-level procedures and conditions, from limited instructional content - filling in omissions and generalizing from examples and feedback. MABLE was evaluated by outside researchers (as required by DARPA who funded the work) and found to perform well in learning a broad range of concepts. However, the machine learning focused evaluation process required designing MABLE not to interact with a human user, but to receive instructional content (lecture style) from an artificial teacher.

Interactivity is particularly important for PBI systems because, without it, the end-user needs to guess what information to provide. That means either providing all information that could be useful without regard to what the system can infer or else knowing the inner workings of the system well enough to select just the right content. Either of these undermines the effectiveness of PBI. In contrast, PBI incorporating question-asking capabilities can focus user effort on delivering only needed content.

The Interactive Bootstrapped Learning (iBL) system described in this paper is an interactive PBI system that builds on the machine learning based approach developed for MABLE. An iBL user modifies target performance software (PS) through instructional dialog used to generate a Code Transformation (CT), a set of point changes to the PS that together achieve user intent. CTs range from simple changes (e.g. substituting a new value for one argument of a specified procedure) to changes to diverse code structures spread across multiple procedures. Dialog between iBL and user is an interactive search for the correct CT with each step of dialog translating into one or more constraints in CT-space (the space of possible code transformations). The core function of iBL is to interact with the user to manage this search, a process that is summarized in this paper following a description of the iBL system. We describe the iBL system with an example taken from a military domain, but note that iBL is a generic system whose underpinnings have been motivated by and demonstrated in many diverse domains, including space shuttle diagnosis, vehicle routing, and Robocup soccer. These domains exhibit many of the same challenges faced by everyday end-users and stress the need for simple and intuitive programming tools.

MABLE: Learning By Instruction

iBL builds on MABLE (Mailler et al., 2009), a domain-independent PBI system that supports several natural instruction methods and outputs learned concepts corresponding to code-level procedures, functions and conditions. It has been tested in diverse domains including RoboCup Soccer, Armored Task Force military doctrine and planning, International Space Station fault diagnosis, Unmanned Aerial Vehicle (UAV) command and control and a hidden domain unknown to the researchers who developed MABLE. The latter domain was recently used by outside evaluators to ensure domain-independence and to allow comparison of MABLE to human learning performance.

The evaluation tested how well MABLE could learn lessons by each of the supported instruction methods. For example, in the UAV domain, the PS might be instructed to capture video imagery of a suspicious target near a power plant using the Teaching by Telling method, then taught what a suspicious target is by Example and what constitutes nearness by Feedback. The project objectives required achieving 75% of human performance on a set of 6 top-level lessons, each of which depended on learning a tiered set of building block lessons. That translated to a required score of 68% (i.e., roughly equal to 100% correct on 4 of the 6 lessons). Reflecting a project assumption that PBI would resemble human instruction (Cronbach and Snow, 1977) in that the best instruction method would depend on what was to be learned, MABLE's score was based on whichever instruction method worked best.

MABLE scored 100% on each of the six final problems. This strong result indicated a high level of maturity in discovering how to use machine learning techniques to learn certain classes of PBI lessons. It also highlighted limitations on what sorts of problems could be effectively addressed without relaxing the non-interactivity assumption. For example, MABLE might have trouble learning by Example due to too few negative examples, but would be unable to ask the user to provide some.

iBL System Overview

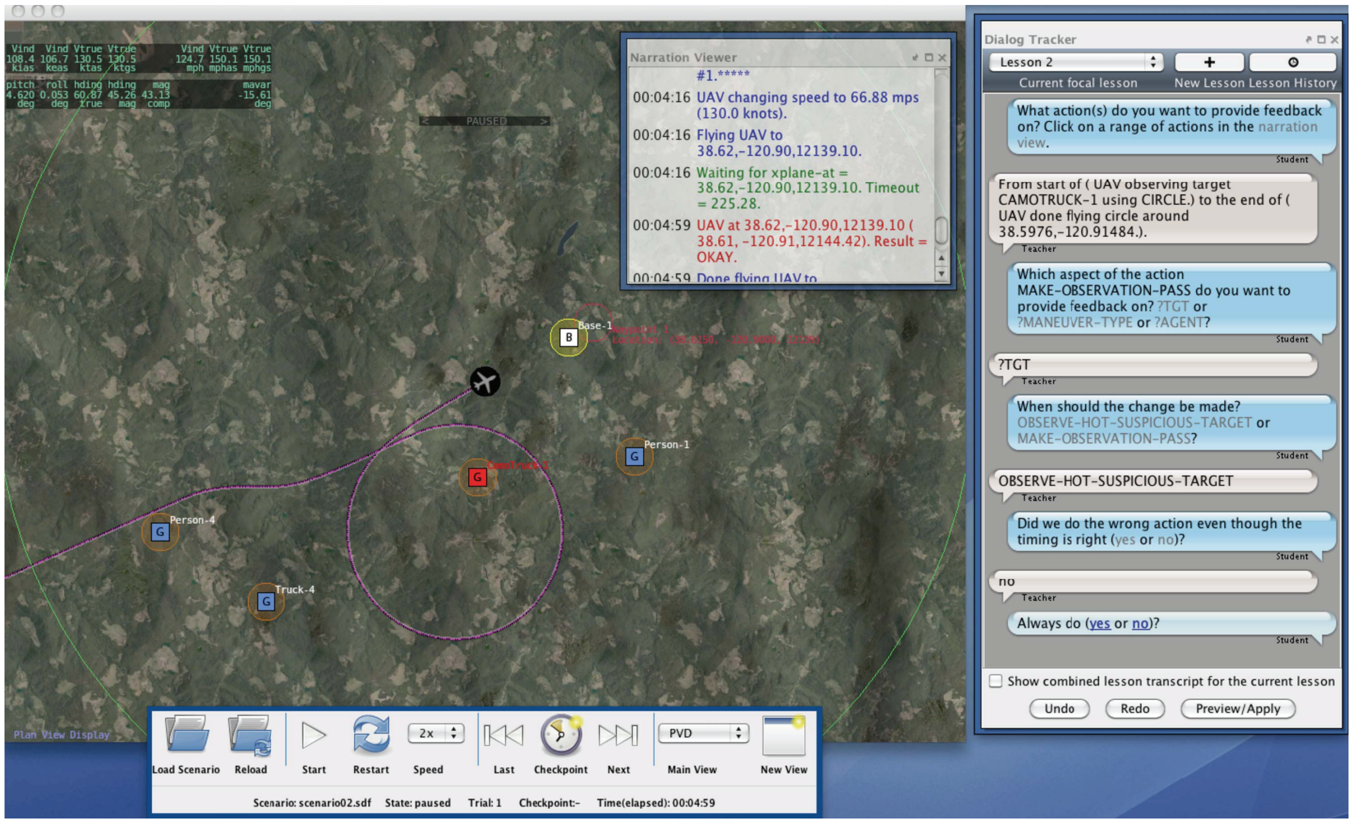
MABLE was designed to support an investigation of how machine learning algorithms can be adapted to distill concepts from sparse but well-chosen and well-structured instructional inputs. But it is not a complete, usable PBI system. In particular, it includes no capability for interacting with a human user, no model of how to conduct such an interaction to advance learning goals and a very simplified model of the PS to be modified by instruction. iBL was developed to fill these gaps and provide a complete, interactive PBI solution.

We assume that iBL users are trained in how to use the system and that they are experts in the operational domain of the PS they wish to modify. The iBL use process starts with some goal for altering system behavior or enhancing system performance. In a UAV Intelligence, Surveillance and Reconnaissance (ISR) domain for example, such goals are likely to originate with end users of the ISR information products (e.g. analysts, decision-makers) whose needs are not being met. The iBL user would then carry out the following steps:

1. Load or create a simulation scenario that produces goal-relevant conditions.
2. Run the simulation until these conditions occur, then pause. This sets a context for teaching.
3. Initiate a new lesson and engage in dialog to specify the lesson goal (user intent).
4. Resume the simulation, pausing to add additional instruction (for instance, to point out instructive examples) as needed.
5. When iBL's interpretation of the lesson goal is accurate, mark the lesson complete. The PS is then modified.
6. Run simulation to see if the PS now behaves as desired.
7. Once validated within iBL workbench, export the code change for appropriate vetting.

Figure 1 depicts iBL in use during a UAV-ISR lesson. From the user perspective, iBL is very much like a video editing tool that allows the user to play, pause, rewind, edit, create, and verify different system behaviors (as supported by the toolbar at the bottom of the figure). The user initiates a lesson and conducts instructional dialog using the Lesson UI widget (right of figure). The first step in such a dialog is for the user to specify (from a menu) how they wish to convey the lesson goal: by telling, example or feedback. For example, if they select by Feedback, the initial stages of dialog will be to specify what should have happened in the just-completed simulation run instead of what actually happened. Subsequent dialog generally follows a question/answer format in which iBL prompts the user to completely specify intent. In addition, iBL will generate questions aimed at selecting between alternative ways of implementing intent (i.e., as a code transformation) that have different effects. Users answer questions in a variety of ways including selecting from a menu of choices, typing a value into a text field, gesturing on the simulation main view to select an object or indicate an example and providing qualitative feedback through specialized UI widgets. Dialog continues until iBL has all the information it needs.

Many different kinds of code-level changes can result from an iBL lesson. These include, for example, adding steps to procedures, modifying step parameters, changing the conditions under which procedure steps are invoked and



Narration Viewer

#1.*****

00:04:16 UAV changing speed to 66.88 mps (130.0 knots).

00:04:16 Flying UAV to 38.62,–120.90,12139.10.

00:04:16 Waiting for xplane-at = 38.62,–120.90,12139.10. Timeout = 225.28.

00:04:59 UAV at 38.62,–120.90,12139.10 (38.61, –120.91,12144.42). Result = OKAY.

00:04:59 Done flying UAV to

Student: What action(s) do you want to provide feedback on? Click on a range of actions in the narration view.

Teacher: From start of (UAV observing target CAMOTRUCK–1 using CIRCLE.) to the end of (UAV done flying circle around 38.5976,–120.91484.).

Student: Which aspect of the action MAKE–OBSERVATION–PASS do you want to provide feedback on? ?TGT or ?MANEUVER–TYPE or ?AGENT?

Teacher: ?TGT

Student: When should the change be made? OBSERVE–HOT–SUSPICIOUS–TARGET or MAKE–OBSERVATION–PASS?

Teacher: OBSERVE–HOT–SUSPICIOUS–TARGET

Student: Did we do the wrong action even though the timing is right (yes or no)?

Teacher: no

Student: Always do (yes or no)?

Figure 1: iBL Workbench.

adding event handlers. One lesson may incorporate a second, subordinate lesson. For example, a lesson that adds a procedure step to make a UAV descend prior to observing a suspicious target might be followed by another lesson defining the suspicious target condition. Similarly, a later lesson might be used to refine a lesson (e.g., narrowing the definition of suspicious target).

iBL Technical Approach

The instructional process takes place in two main phases: intention elicitation and code transformation.

Intention Elicitation

The goal of intention elicitation is to acquire a description from the user of how the system should behave and represent this as a formal, machine-readable software requirement grounded in a declarative domain model. Elicited requirements are represented using an enhanced temporal plan network notation based on Constraint-based Attribute Interval Planning (CAIP) (Frank and Jónsson, 2003).

CAIP representations consist of State Variables (SVs) and intervals on SV timelines called tokens. Each token is defined by a start-time, end-time and value. For example, given SV-1 representing the altitude of the UAV, there might be a token (SV=SV-1, start = 1:07.30, end = 1:08.00, value = 2000) representing a specific 30 second interval where the aircraft should be at 2000 ft. Often, token attributes will be defined as functions of other token attributes (e.g., where the start of token-2 is 10 seconds after the end time of token-1). We depict CAIP software requirements as shown in Figure 2 with line segments showing functional dependencies between tokens.

Each token represents an external condition (e.g., that altitude is at a certain value or within a certain range) or internal condition (e.g., that the performance system is carrying out a certain activity such as navigating to a target). A CAIP structure as a whole can be interpreted as a software requirement of the general form: under applicability conditions (C1) the software will act to produce conditions (C2) hold which will lead to a cascade of conditions (C3) resulting in meeting outcome requirements defined by conditions (C4).

To support this kind of interpretation, we extend the standard CAIP notation in three ways. First, tokens are labeled to be of type *enforce* (shown in blue), meaning it must hold for the requirement to be met as needed for C2–C4, or of type *detect* (red) meaning that the condition must be observable as needed for C1. Second, tokens are *direct* (bold border) if the condition can be directly controlled or observed. Third, tokens are *anchors* if they define either the intended scope (applicability) or end goal of the requirement.

When the user begins a lesson, a single token is created to seed the requirement specification process. For example, starting a lesson of type Feedback-About-Outcome creates a token with enforce=true and anchor=true but all other at-

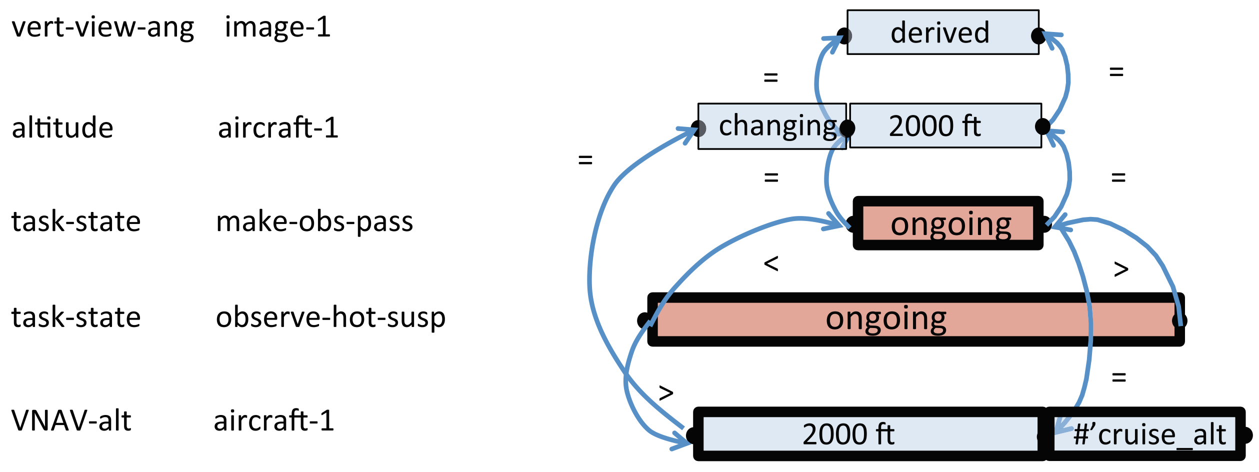


Figure 2: CAIP Representation of Instructor Intention.

tributes are unknown. In other words, the initial state is a partial requirement stating that some outcome ought to be enforced, but nothing yet is specified about what, when or how. Knowledge acquisition goals (KAGs) are generated to fill in the missing information. iBL first tries to satisfy the goal by looking in a domain model describing background knowledge. KAGs that cannot be satisfied this way (most of them) require information from the user. A dialog management component selects which unsatisfied KAG to ask about next and selects a method for interacting with the user. These methods (called Dialog Episode Operators since they are meant to be composable into dialog plans) specify how to pose the question and which UI widget must be invoked to let users express an answer.

KAGs drive the requirement elicitation process. There are 4 sets of rules for generating elicitation KAGs. The first set ensures token completeness (i.e., that the start time, end time, state variable and value of each token is completely specified). The second ensures intention completeness: that every token is labeled for enforce vs detect, directness and anchoring, that all indirect tokens are causally linked to direct ones, and that users have a chance to exercise certain forms of discretion (e.g. whether to specify additional restriction conditions on the applicability of the intention). The third set is for generalization. Users may answer questions by referring to specific simulator states and examples, often creating ambiguity about how to generalize the information. These KAGs either query the user or invoke machine learning algorithms to resolve such ambiguities. The fourth set is for model completeness. In particular, the domain model may not include condition types that the user needs to reference (e.g., the definition of a suspicious target) or causal operators needed to enforce an indirect token (e.g., that changing altitude changes viewing angle on a target). Dialog continues until all KAGs are achieved, indicating that the requirement is completely specified.

The following UAV-ISR scenario illustrates intention elicitation dialog leading to creation of the formal requirement shown in Figure 2.

Example: Initially, the user hears complaints from intelligence analysts receiving UAV surveillance data that video imagery collected during circular observation passes is often taken at too oblique an angle to be useful. The user recreates this situation in simulation, pauses when a good context for teaching has been reached (i.e., when the UAV is currently observing a target at a bad angle) and begins a new lesson.

iBL: What do you want to give feedback about?
User: vertical-view-angle of camera-image-1 *The user gestures to object in system object list. iBL generates an enforce/anchor token and a KAG for ways to affect vertical-view-angle and finds altitude or target-distance in background knowledge.*
iBL: Should a better vertical view angle be achieved by changing altitude or target-distance?
User: altitude. *The user specifies "how" to modify behavior. iBL creates an enforce token for a new value of altitude, and a KAG for the temporal interval over which the value holds.*
iBL: When should altitude have been different to achieve a more desirable vertical-view-angle? *The user specifies "when" to modify behavior as constraints on start and end times of the new altitude token.*
User: From the start to the end of make-observation-pass. *The user gestures to interval in narration view. iBL creates a direct detect token for make-observation-pass to anchor future constraints on it.*

[. . .]

iBL: In what conditions should altitude be adjusted to achieve a better vertical view angle? *Several options are listed.*
User: (a) When in observe-hot-suspicious-target procedure. *The user sets the condition for modification. iBL creates a function describing the condition and annotates altitude token.*

[. . .]

iBL: I know of one way to control altitude which is set-vnav-altitude-restriction(altitude, location). Is it OK to use this or should we define an alternative method? *iBL uses the domain model to determine that the VNAV-alt setting affects altitude and uses the code model to determine that VNAV-alt values are directly enforceable. iBL allows ad-hoc creation of new behaviors.*
User: OK to use this. *iBL creates a direct enforce token for set-vnav-altitude-restriction(altitude, location) and constrains it to begin before the altitude token.*

[. . .]

iBL: What altitude should be achieved before starting make-observation pass when performing the activity observe-hot-suspicious-target? You can tell altitude directly or give feedback and examples.
User: Tell.
iBL: Should the same altitude be selected in all conditions or should it vary depending on the situation?
User: same. 2000 ft.

[. . .]

Code Transformation

The second phase of instruction is to specify a code transformation (CT) to the performance system code base that is intended to achieve the user's intent. This is treated an AI planning problem. Inputs include a planning goal, the software requirement specified during the intention elicitation phase, and an initial world state in the form of a model of the performance software to be modified. The code model is generated by static analysis and includes submodels for data flow, control flow and intended behavior (assertions). iBL includes a set of Code Transformation Templates (CTTs) that serve as planning operators. Each CTT includes preconditions (properties of the unmodified code base that determine if the CTT applies), a goal condition (that must match the elicited software requirement) and post-conditions (a set of point modifications to the code base which together achieve the goal).

Figure 3 shows an example of code in the RAPs language (Firby, 1987) transformed to meet the requirement represented in Figure 2. The original code specified a procedure for observing a hot suspicious target that consisted of four steps: laterally navigating to a position near the target, powering up the camera, making a circular observation pass around the target and then powering down the camera. Applying a code transform to achieve a better view angle resulted in changes (highlighted) for inserting a new altitude change step prior to be observation pass and then changing the altitude back after the observation pass is complete. CTTs are specialized for a particular programming language, but represent idiomatic code changes that apply in many languages. For instance, the idea of modifying code to make a temporary change to some condition, perform an action and then restore the condition to its prior value is common in all procedural languages.

Though a general approach would create a code transform that combines multiple CTTs, just as AI plans typically sequence multiple operators, our current approach is limited to code transform derived from a single CTT. In this simplified approach, the problem of CT generation has two parts: selecting a CTT and specifying its parameters.

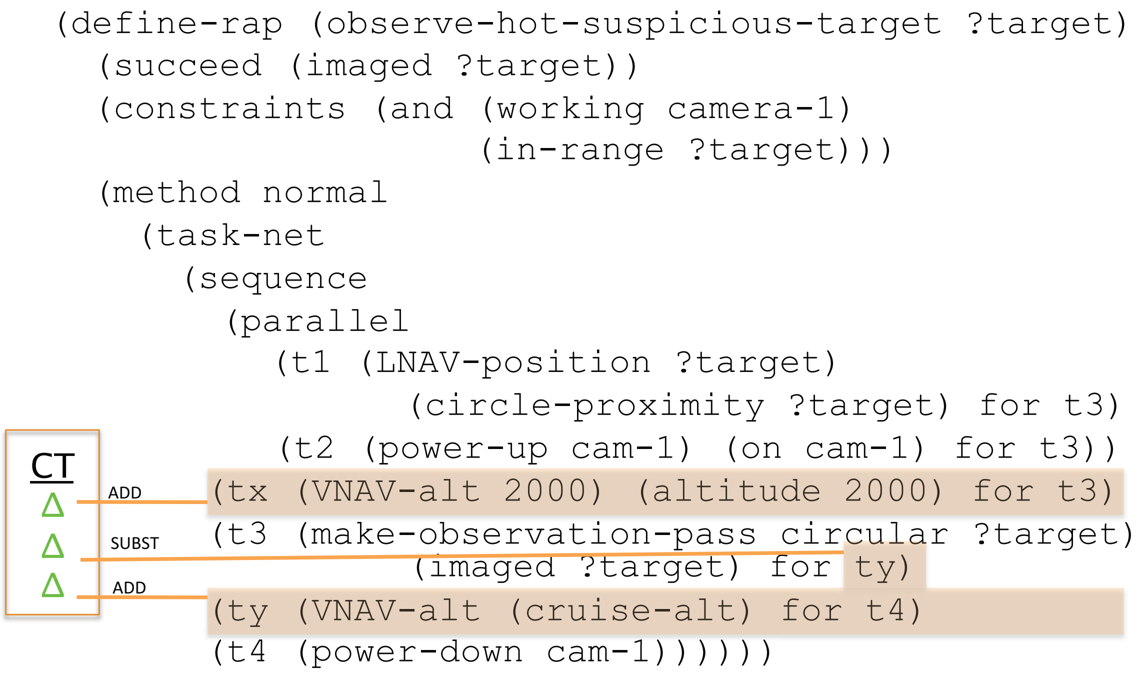
CTT selection requires matching the CTT goal to the user-specified intent and checking whether the CTT's preconditions hold. For instance, the CTT described above for temporarily modifying a variable requires matching that variable to a directly enforced condition in the user intent specification. Preconditions define what must be true of the unmodified code base for a particular change to have the desired effect. For example, a CTT that makes changes by inserting steps before and after an existing step will only work if that step is within a sequential portion of a procedure. If it is in a segment of parallel code, a different CTT is needed.

CTT specification is mainly a side-effect of CTT-selection. For instance, the CTT in our example requires specifying which RAPs procedure to modify and which step within the procedure should execute when the condition (altitude in this example) is temporarily changed. These values are bound during model queries used to test preconditions. One exception is when there is genuine ambiguity about how to integrate new behavior with correct existing behavior. When unknowns of this sort are identified by iBL, the system generates a KAG to acquire the needed information from the user. For example, iBL may note that in the simulated scenario, altitude was set earlier to a value defined by the function cruise altitude and need to know if exact prior value should be restored or whether altitude should be restored to the current value of this function.

Conclusion

iBL differs from prior PBI approaches in its overall user interaction design, support for multiple interleaved instructional methods, use of ML techniques to compensate for limited instructional content, use of a constraint-based hypothesis space to allow composition ("bootstrapping") of incrementally-specified lessons and ability to modify extant code rather than add additional code elements. However, aspects of it are closely related to many previous PBI systems, especially those for task learning (Myers et al., 2007; Gervasio, Yeh, and Myers, 2011) which also use machine learning to compensate for limited content. Another closely related approach is the work of Fritz and Gil (2011), also an outgrowth of MABLE research, which combines learning by demonstration and by telling. An important difference from iBL is that hypotheses are in the form of executable code rather than CTs and user intent representations. This reduces the amount of interaction needed to complete a lesson but does not easily support more complex changes that need to be defined incrementally.

The central element of the iBL approach is use of a constraint-based representation for the hypothesis space that allows automatic identification of further knowledge requirements. These can be used to drive dialog and thereby gain the advantages of interactive PBI. A variety of AI technologies are integrated to support this process including



```
(define-rap (observe-hot-suspicious-target ?target)
  (succeed (imaged ?target))
  (constraints (and (working camera-1)
                    (in-range ?target)))
  (method normal
    (task-net
      (sequence
        (parallel
          (t1 (LNAV-position ?target)
                (circle-proximity ?target) for t3)
          (t2 (power-up cam-1) (on cam-1) for t3))
        (tx (VNAV-alt 2000) (altitude 2000) for t3)
        (t3 (make-observation-pass circular ?target)
                (imaged ?target) for ty)
        (ty (VNAV-alt (cruise-alt) for t4)
        (t4 (power-down cam-1))))))
```

Figure 3: Code Transformation.

elements of dialog management, constraint-based reasoning, automatic model generation, AI planning, and machine learning. Limitations of the system, or alternately opportunities to improve it, are primarily limitations on how well these elements have been employed. For example, the AI planning notation used to represent user intent is not very good for expressing concepts such as repetition and conditional execution. This limits how well iBL can be used to modify many code-level control structures. Other limits stem from having a limited library of CTTs and DEOs to conduct dialog and effect code transformations. These limitations will be addressed by ongoing iBL development.

Acknowledgements: This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-07-C-0060.

References

Cronbach, L. J., and Snow, R. E. 1977. *Aptitudes and instructional methods : a handbook for research on interactions / by Lee J. Cronbach and Richard E. Snow*. Irvington Publishers : distributed by Halsted Press, New York.

Firby, R. J. 1987. An investigation into reactive planning in complex domains. In *Proceedings of the sixth National conference on Artificial intelligence - Volume 1*, AAAI'87, 202–206. AAAI Press.

Frank, J., and Jónsson, A. 2003. Constraint-based attribute and interval planning. *Constraints* 8:339–364.

Fritz, C., and Gil, Y. 2011. A formal framework for combining natural instruction and demonstration for end-user programming. In *Proceedings of the 2011 International Conference on Intelligent User Interfaces (IUI), February 13-16, 2011, Palo Alto, CA, USA*.

Gervasio, M.; Yeh, E.; and Myers, K. 2011. Learning to ask the right questions to help a learner learn. In *Proceedings of the IUI'11*.

Mailler, R.; Bryce, D.; Shen, J.; and O'Reilly, C. 2009. Mable: A framework for learning from natural instruction. In Sierra, C.; Castelfranchi, C.; Decker, K. S.; and Sichman, J. S., eds., *AAMAS (1)*, 393–400. IFAAMAS.

Myers, K. L.; Berry, P.; Blythe, J.; Conley, K.; Gervasio, M. T.; McGuinness, D. L.; Morley, D. N.; Pfeffer, A.; Pollack, M. E.; and Tambe, M. 2007. An intelligent personal assistant for task and time management. *AI Magazine* 28(2):47–61.

Myers, B. A.; Pane, J. F.; and Ko, A. 2004. Natural programming languages and environments. *Commun. ACM* 47:47–52.

Pecheur, C.; Andrews, J.; and Nitto, E. D., eds. 2010. *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. ACM.

Shu, N. C. 1999. Visual programming: perspectives and approaches. *IBM Syst. J.* 38:199–221.