MABLE: A Framework for Learning from Natural Instruction

Roger Mailler
University of Tulsa
Tulsa, Oklahoma 74104
mailler@utulsa.edu

Daniel Bryce
Utah State University
Logan, Utah 84322
daniel.bryce@usu.edu

Jiaying Shen,
Ciaran O'Reilly
SRI International
Menlo Park, California 94025
{shen,oreilly}@ai.sri.com

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Design, Experimentation

Keywords

Learning, MABLE, Architecture

ABSTRACT

The Modular Architecture for Bootstrapped Learning Experiments (MABLE) is a system that is being developed to allow humans to teach computers in the most natural manner possible: by using combinations of descriptions, demonstrations, and feedback. MABLE is a highly modular, well-engineered, and extendable system that provides generalized services, such as control, knowledge representation, and execution management. MABLE works by accepting instruction from a teacher and forms concrete learning tasks that are fed to state-of-the-art machine learning algorithms. To make the learning tractable, specialized heuristics, in the form of learning strategies, are used to derive bias from the instruction. The output of the learning is then incorporated into the system's background knowledge to be used in performing tasks or as the basis for simplifying the process of learning difficult concepts.

Although still in development, MABLE has already demonstrated the ability to learn four different types of knowledge (definitions, rules, functions, and procedures) from three different modes of student/teacher interaction on two separate, qualitatively different domains. MABLE presents a unique opportunity for machine learning researchers to easily plug in and test algorithms in the context of instructible computing. In the near future, MABLE will be freely available as an open source project.

1. INTRODUCTION

The computer is probably the most flexible and powerful tool ever devised by humans. Yet in spite of these properties, computers still retain one fundamental limitation: the cost, difficulty, and time needed to develop new software. In stark contrast, humans are able to learn complex tasks from, what most programmers would consider to be, horrible instruction. Imagine for a moment a parent teaching their child to write the letter "p". Most would **describe** it as "Draw a straight line and then put a loop at the top". Then the parent would **demonstrate** the action of drawing the "p" while explaining the actions and finally, would allow the child to demonstrate mastery of the action while giving **feedback** on their performance. Assuming the child can draw a reasonably good line and loop, they will master this task quickly. We aim to design a digital student that can learn from similar forms of natural instruction.

The Modular Architecture for Bootstrapped Learning Experiments (MABLE), which is being constructed as part of DARPA's Bootstrapped Learning program, can be thought of as a step toward *human-instructible computing*. The primary goal is to create a system (a digital student) that learns from human instruction in all of its various forms and in the face of incompleteness and imprecision. This makes MABLE different from existing systems that are designed to learn a single type of knowledge (e.g., a process) from a single form of instruction (e.g., a demonstration). MABLE must be able to create and modify hypotheses (bootstrap) that may have been learned by a myriad of different learning techniques.

Achieving human-instructible computing entails a large number of conceptual challenges, some of which will be addressed in this paper. For example, how can we leverage all of the work that has been done in machine learning over the past 50 years? How can the system control the learning process? How can you form a common representation that can represent everything MABLE has learned and provide sufficient meta-information about the use of that knowledge? How can we manage the interaction between the teacher and the system to avoid the natural language understanding problem? How can we resolve ambiguities that result from incomplete and inaccurate instruction or as a result of misunderstanding?

A secondary goal of the program is to provide the MABLE system (and supporting architecture) to the AI community as a research platform. We expect that researchers will benefit from having a plug-in architecture with which they can test algorithms for learning from natural instruction, and more-traditional algorithms for machine learning. Therefore, every attempt as been made to make MABLE as modular, extendable, and well-constructed as possible. MABLE will be freely available as an open-source project released under the terms of the BSD license [10].

In the following sections of this paper, we give an overview of the bootstrapped learning program and the current version of MABLE. We then describe the framework that is used to test and develop MABLE and an example of instructible computing in Blocksworld. Finally, we will discuss the open issues within our system, which leads to our future directions.

2. BOOTSTRAPPED LEARNING

As mentioned in the introduction, MABLE is being developed as

Cite as: MABLE: A Framework for Learning from Natural Instruction, Roger Mailler, Daniel Bryce, Jiaying Shen, Ciaran O'Reilly, *Proc. of 8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009),* Decker, Sichman, Sierra and Castelfranchi (eds.), May, 10–15, 2009, Budapest, Hungary, pp. 393–400
Copyright © 2009, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org), All rights reserved.

part of DARPA's Bootstrapped Learning (BL) Program. The goal of this program is develop a system that can learn from a teacher using natural instruction to revise, refine, or increase its capabilities by bootstrapping on its existing knowledge. This knowledge takes many forms including new rules, functions, and processes and can be used to either perform tasks in a simulator or the real world.

To help facilitate the learning, the teacher is assumed to be able to break down complex learning tasks into rungs that form laddered curriculum. Each rung is used by the student to create, refine, or replace existing knowledge in its memory. You can imagine this process as taking some set of knowledge K_t and combining it with one rung of instruction I_t to yield some new or, hopefully, improved knowledge K_{t+1} or

$$K_t \cup I_t \mapsto K_{t+1}$$

Although this formula is a vast simplification of bootstrapping, one can immediately begin to form a set of issues that needs to be answered in order for bootstrapping to work. The first issue that the astute reader will recognize is that K needs to be in a standardized form because it is recursively defined. Within the MABLE system, knowledge is represented in a language called *Interlingua* (IL) [4], discussed in Section 2.1.

The instruction I may come in a different language than K. For example, a human may speak in English, although, their brain represents information as something very different (electrical signals, chemicals, neural connections). Within the BL program, we use a second language called the *Interaction Language* to represent communication actions, including speech, gestures, and other percepts. The interaction language will be discussed in section 2.2.

Interaction Language describes the basic syntax of communication between the teacher, student, and world, but it says nothing about the semantics or protocol of the interaction. Section 2.3 describes the basic form of the curriculum and how each rung in the ladder is taught to the student through the use of Natural Instruction Method (NIM) contracts.

2.1 Interlingua

IL is a specialized language that was custom built to support the BL program. It is a general, domain-independent language that is robust enough to represent a tremendously diverse set of knowledge, yet limited enough to be learnable. Interlingua is not a simple language though. In fact, it was designed to support additional learnable extensions as the BL system evolves. Interlingua has a fixed upper ontology that is extensible to include other embedded languages.

Currently, Interlingua has three primary embedded languages. The first of these is the *syntax language*. Being one of the most powerful languages in IL, the syntax language permits the creation of new legal syntax and is used in a recursive manner to define the upper ontology and IL itself. The syntax language is composed of the keywords **is**, **arg**, and a simplifying macro called **defSyntax**.

The **is** keyword is used to define new types within the IL language. For example, one could say

```
 is Dog Animal;
```

which defines a new type called Dog that is a sub-type of the Animal type. Unlike many languages, IL allows for multiple inheritance, with the explicit order of super-types disambiguating properties inherited from parents in the type hierarchy.

The keyword **arg**, defines properties of types. For example, an integer property of a Dog called age is specified by:

```
arg Dog age Integer;
```

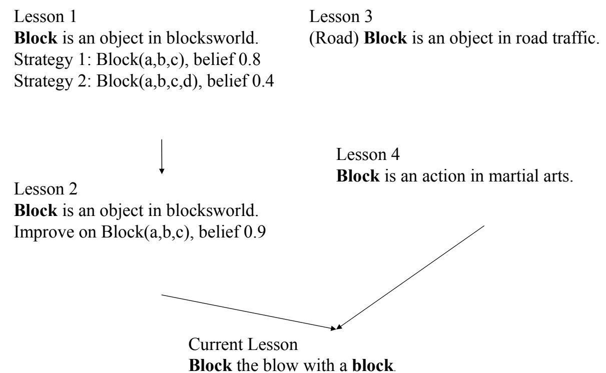

Figure 1: Knowledge ambiguity example

The **defSyntax** macro is a short-hand method for creating new types. Instead of writing the two statements above, we can specify the Dog type as follows:

```
defSyntax Dog extends Animal (Integer age);
```

The second of the IL languages is the *basic encoding language*. This is the language of the objects and properties of objects that exist in real or simulated worlds. For example, the following is a legal instance of an object in the basic encoding language of IL:

```
Dog(name=Rover, color=black, age=3);
```

The third core language of IL is the *code body language*. As the name implies, this language specifies computable procedures or functions. Currently, the code body language has support for evaluating functions, running recursive procedures that are formed with combinations of the control statements "if" and "while", and for evaluating predicates defined in first-order predicate logic.

The classic "Hello World" application is given below as a sample

```
defSyntax HelloWorld extends Function;
defCode HelloWorld FunctionEngine
 Functionbody(Return(returnValue="Hello World!"));
```

To support the executable form of IL, the current implementation of the language comes with an interlingua virtual machine (ILVM). As the name implies, the ILVM is capable of taking well formed IL and executing it either against the world or just natively. Like the IL language itself, the ILVM is capable of supporting new code body language extensions by allowing programmers to add additional defCode engines to it. As the example above shows, this extensibility is facilitated by having the authors of defCodes specify the name of the interpreter (e.g., the FunctionEngine) that is able to understand the specific syntax and semantics of the code body.

2.2 Interaction Language

The interaction language is a specialized language, which is constructed using IL, that is used within BL to represent **perceptions** coming into MABLE, **utterances** that are made by MABLE or the teacher, and **imperatives** that MABLE or teacher must perform. In other words, interaction language is the language that the student uses to interact with its environment.

The imperatives, percepts, and utterances that are exchanged between the student and its world form a *timeline*. The timeline, which forms the central communication mechanism between the student and its world, is simply a time-stamped record of *messages*. These messages can be retrieved by the teacher, the student, or the world.

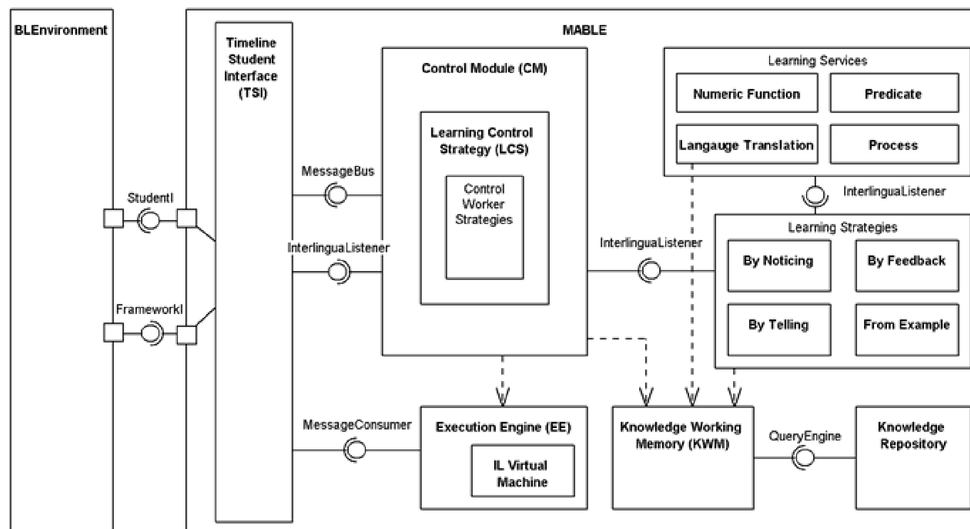

Figure 2: MABLE Framework

The interaction language is made up of three primary interaction modalities: imperatives, perceptions, and utterances. The first of these, imperatives, are requests to perform procedures, actions, or logical/functional computations. Actions are the set of things that MABLE or teacher can directly do in the environment. For instance, a primitive action might be "kick", which is not decomposed to "lift foot, bring foot back...".

Perception messages are associated with objects that the student can perceive in the world. For example, if MABLE is shown a brown table with one blue block on it, it will receive the following perception message:

```
Perception(name=Percept-123,
 gainedPercepts={Table(color=brown, name=table1),
                 Block(color=blue, name=block1,
                       support=table1))},
 lostPercepts=null);
```

The last, and probably most complex, of the interaction modalities is the utterance language. The utterance language, like the other interaction modalities, is a sub-language of interlingua and as such is a structured language. This is not to imply that the language is unambiguous. In fact, interaction language is purposely constructed to introduce limited ambiguity in that the teacher may use terms in ambiguous way like "use the block to block the car" (see Figure 1) or may use them in an incomplete manner, as in using Houston to refer to the Houston Astros. In addition, it is not generally assumed that the teacher knows the symbols that the student uses to represent its knowledge. For instance, the student may understand the word "Dog", but the teacher refers to a dog as a "Canine". This will be discussed in further detail is Section 3.2.

2.3 Curriculum

As mentioned previously in this paper, the goal of MABLE is to learn new concepts from the teacher. To help the student grasp difficult concepts, it is often necessary to break the concept down into simpler components that can be built or bootstrapped upon in the future. Each of these simpler components forms a lesson in a curriculum. For example, the curriculum may have the goal of teaching the student to build a stack of blocks. This can be broken down into teaching the student to recognize blocks, lift blocks, reason about the "supports" relationship, etc. Lessons can therefore be thought of the combination of a *target concept* and a natural instruction method (NIM) for conveying that information.

Each lesson requires that MABLE have some prerequisite knowledge. For example, you cannot stack a block if you have no idea how to pick one up. This begs the question about whether the student can learn anything, because when it first starts, it has no knowledge. In BL, this problem is resolved by *injecting* the student with IL that provides basic information about its world and its capabilities on startup.

Target concepts can take many forms, but in BL these are generally, either a new rule, definition, function, or procedure. Because of the laddered nature of the curriculum, the lessons also usually include an examination at the end of them. This helps both the teacher and student determine if a proper level of mastery has been achieved before moving onto more difficult concepts.

Each lesson also has an associated NIM. NIMs can be thought of as a contract or protocol that is used by the teacher and student when conveying information intended to teach the particular type of target concept. For example, one NIM contract is the **demonstration** of a **procedure**. Currently, there are eight NIM contracts that cover telling and examples of predicates, functions, and procedures as well as feedback using score and explanation.

3. MABLE

The MABLE framework (Figure 2) embodies an integrated approach that provides the functional capabilities needed for bootstrap learning. It consists of a highly modular software framework that can be downloaded, installed, and executed with minimal effort. In this section, we describe MABLE's overall architecture and top-level modules: controller, execution engine, knowledge working memory and repository, learning strategies and services.

Section 3.1 describes features of the software framework that tie together MABLE's functional modules and make MABLE highly extensible. Section 3.2 describes the Knowledge Repository (KR), Working Memory (WM), and Query Engine (QE) that store and retrieve IL objects. Section 3.3 describes the learning strategies and learning services that encapsulate alternative algorithms for learning from natural instruction and standard machine learning, respectively. Section 3.4 describes the Execution Engine (EE) module that supports IL program linking, interpretation, and execution. Section 3.5 discusses the centralized Control Module (CM) that co-

ordinates all system behavior, much like an operating system's kernel. In Sections 4 and 5, we describe how MABLE fits into a larger environment with a teacher and simulator for instructible computing and describe how MABLE's modules interact in an example curriculum.

3.1 Framework Overview

At its simplest, the MABLE framework can be thought of as a collection of inter-operating modules that collaborate to tackle the bootstrap learning problem. A primary objective of the MABLE architecture is for learning strategies and services to be easily added or removed to configure alternative bootstrapped learning experiments; this requires considerable foresight into the software framework design.

To support multiple interacting modules, MABLE (which is implemented in Java) is built on the Spring Framework [11], allowing it to take advantage of several pre-existing and proven technologies. Spring greatly simplifies module life-cycle management and messaging. In particular, at its core, the Spring Framework supports the *inversion of control containers* and *dependency injection* design patterns, which decouple the modules and make them reactive. Support for these patterns, simplifies the modularization of MABLE's components by having each module programmed to explicit interfaces. As such, the system can be configured in different ways. Another useful feature of the underlying Spring framework is the support for transparent (i.e., no re-compilation) distribution of the modules across a network; thus, computationally costly learning strategies and services can be run on dedicated hardware.

In addition to the Spring Framework, MABLE uses several other Open Source projects, and all third party libraries or contributed source used by MABLE are BSD license [10] compatible or better. MABLE, once matured to a sufficient level, will itself be publicly released as an open source project.

3.2 Knowledge Storage and Retrieval

One of the key goals of the MABLE framework is to bootstrap learn new concepts in terms of existing knowledge, either by interpreting, extending, chaining, modifying, or refuting. This style of learning requires both a short term and a long term memory. The long term memory stores and supports queries over knowledge obtained from other curricula and lessons, while the short term memory focusses on knowledge relevant to the current target concept. In MABLE, the Knowledge Repository (KR) is the long term memory and the Working Memory (WM) is the short term memory, which is a cache of the KR. The Query Engine (QE), as the gateway to the WM and KR, populates and maintains the KR as well as processes WM storage and retrieval requests.

Knowledge Repository: The KR stores multiple information types, including the IL ontology, concepts learned by MABLE, and meta-knowledge about concepts (called provenance). Storing the IL ontology in the KR enables recreation of the language at each startup and the grounding of each concept that was, and will be, learned. However, the KR's primary function is to store all knowledge ever learned, including instances (domain objects), declarative knowledge (predicates, functions and rules) and procedural knowledge (a series of steps with branches and loops). The last type of information, provenance, is meta information about the knowledge learned, including which learning strategy learned this piece of knowledge in which lesson with which instructions and prior knowledge. Provenance also records versioning information and distinguishes competing hypotheses of the same concept. With provenance information, the student is able to reconstruct the context of learning at a later time, and revise or extend the existing knowledge if necessary.

Query Engine: There are multiple queries that strategies can make of the KR via the QE. Queries supported by the QE can vary significantly in detail from the name of the knowledge to some property. For example, MABLE may hear the assertion "This is a dog" and sees a few examples of dogs. Before trying to learn the concept "dog", a learning strategy may query the KR to see if MABLE already knows what a "dog" is, and whether its existing concept of "dog" needs to be modified to conform to the examples provided. The first query is of the form "Do I know anything about dog". The KR returns all definitions of concepts named "dog" and caches the concepts within the WM. However, if no concept named "dog" is known, MABLE may know "dog" by another name, such as "canine". In this situation, MABLE can query attributes of objects used in the instruction. For example, the query may be "Do I know an animal that has four legs and a tail and barks?".

Ambiguity: IL permits ambiguity by allowing alternative concepts to share the same name. Viewed as a feature of IL, shared names refer to competing hypotheses of the same concept and mimic the flexibility of human language. One important function of the QE is to identify and possibly resolve ambiguity to its best ability. When the ambiguity cannot be resolved, the QE returns the possible matches to the query with more information (to help strategies with disambiguation).

Consider an example of four types of ambiguity, illustrated in Figure 1. In this example, MABLE has previously learned in three lessons different concepts named "block". The types of ambiguity are:

- Type Ambiguity: When "block" is a procedure (action/verb) and "block" is a predicate (object/noun), the QE returns both or filters by a specified type.
- Contextual Ambiguity: When "block" in the Blocksworld is different from (road) "block" in the traffic context, the QE can filter the context in the query.
- Competing Hypotheses: A belief weight is attached to each hypothesis and the QE can filter by a belief weight threshold.
- Revisions: Revisions of the same concept, by default, should shadow the previous versions. The QE returns the most recent version by default and, when specifically asked, returns older versions.

The current system fully supports revisions and partially supports scenario type ambiguity. Fully supporting all types of ambiguity resolution is planned for the near future.

Working Memory: The WM has three main functions: i) storing the current lesson context, ii) caching KR queries, and iii) acting as a shared medium for knowledge interchange. The lesson context includes teacher instructions, percepts, and test grades. The WM caches queries and thus focuses the attention of MABLE on only the part of the knowledge base that is thought to be relevant to the current learning session. The most important function of the WM is to share knowledge between learning strategies and services, and is closely related to blackboard systems [3]; Learning strategies output intermediate learned concepts and/or hypotheses they deem important or useful for the other collaborating strategies. Together with the knowledge retrieved from the KR, this collection of new forming knowledge is the focus of attention used to bootstrap new concepts.

3.3 Learning Strategies and Services

The learning strategies and services are the most modular components of MABLE. Learning services capture well-studied black-box machine learning algorithms, such as those addressing induc-

tive logic programming (ILP), reinforcement learning, and regression. These services are modular so that researchers can easily perform bootstrapped learning experiments with their algorithms in MABLE.

Learning strategies are in some respects novel to bootstrapped learning. That is, learning strategies are meant to understand a particular natural instruction method, such as learning predicates by example. Learning strategies interpret the teacher's instruction and apply bias in invocations of various learning services. For example, a strategy for learning predicates by example, may attempt to learn a predicate by invoking an ILP service multiple times by selecting different features each time. Ambiguity in the instruction (such as the "block" example from the QE discussion) may require exploration of multiple alternatives to narrow down the set of features.

3.4 Execution Engine

The EE is responsible for executing, controlling, and monitoring procedures that are specified in IL. Internally it contains an IL Virtual Machine, and has several additional features that simplify interacting with the teacher and world, these include:

Interacting with the World: During learning and testing MABLE can be asked to perform a task in the world, which can comprise multiple actions. Each domain has a set of primitive actions; in Blocksworld these would be grasp a block, raise, lower, release, and move the claw. These atomic action requests are detected by the EE during the run of an IL program and are translated into world actions. Once an action is executed, the EE will wait for an observation from the world before proceeding. In addition, all teacher instructions received by MABLE during the execution of a program are consumed by the EE. This allows the EE to annotate execution traces with instructor feedback. On completion of a program, the EE can provide a trace of actions and responses received to be analyzed further.

Call Backs: Learning strategies that invoke the EE typically provide completely self-contained IL programs, but in some cases a call back to the invoking strategy can be useful. Online learning algorithms may require procedures that generate data (such as state and action sequences); call backs can i) feed runtime data to invoking strategies and ii) allow strategies to dynamically adapt the call back sub-procedure based on the runtime data. For example, a strategy can associate a call back with an sub-procedure to learn the cases where the sub-procedure is invoked and supply the correct variation on the sub-procedure for the execution engine to perform (perhaps integrating a reinforcement learning algorithm implemented within a learning strategy).

Standard Libraries and Extended Interpreters: The EE includes a standard set of library routines that can simplify the construction of more complex procedures during learning. In addition, extended interpreters allow specialized problem solvers to be used to execute different classes of IL programs. This avoids the problem of trying to use a single general purpose problem solver to handle all situations. Instead, specialized solvers can be developed and registered with the Execution Engine in order to extend the capabilities of IL and MABLE in a modular manner.

Dynamic and Static Linking: During the normal flow of events, while learning and being tested, MABLE receives imperatives from the teacher, asking it to answer a question directly or to perform a task. In these scenarios, MABLE must figure out what of its current knowledge is relevant to best answer the imperative. The response given by MABLE to the teacher is generated by constructing an IL program and then executing it in the EE. MABLE may have learned multiple candidate solutions, but the EE has no knowledge of these different solutions nor how to choose between them. In these cases, the EE delegates dependency resolution to a dynamic linker. Different algorithms may be used to select the relevant set of IL objects necessary to construct a complete IL program.

3.5 Control

The Control Module (CM) manages MABLE with a Learning Control Strategy (LCS). The LCS is a policy responsible for determining what MABLE is doing at any given moment in time; whether selecting a lesson, answering an imperative, processing instruction, or employing a learning strategy to learn a concept. Each such control action is classified as either a learning strategy or a worker strategy. Learning strategies perform the majority of learning, taking teacher instruction and previously learned concepts to bootstrap learn a new concept. Worker strategies embody other actions not used for learning, such as lesson selection, imperative answering, and instruction processing.

Like any policy, the LCS requires a state space to which it maps actions (strategies). The state space is a view of MABLE's working memory called a template. The template is a structured representation of the data present in working memory. For example, the template view of the working memory categorizes i) concepts and teacher instruction into labeled objects, actions, procedures, predicates, and functions, and ii) information about the state of the curriculum, such as outstanding imperatives, lesson selection options, the lesson's natural instruction method, and a history of actions.

The template assists the LCS to determine which strategies are applicable at any given time. Each strategy declares a pattern (precondition) with respect to templates. Learning strategy patterns state constraints, such as the natural instruction method type, the existence of labeled data, the existence of reward functions, etc. Worker strategy patterns also declare constraints over working memory, such as the existence of an outstanding imperative and a suitable concept for answering the imperative.

With templates and patterns to determine the applicable strategies, the LCS must select among the applicable strategies which to execute next. There are several issues that the LCS must consider when selecting strategies: i) is the target concept known, ii) has the target concept been learned *well*, iii) is the learned concept of high quality, and iv) which instructions or concepts should can be used to learn? Each of these issues are the subject of ongoing research on the LCS. The primary goal of the LCS is to create high-quality concepts that can then be incorporated into the corpus of knowledge that MABLE uses to bootstrap.

The current LCS is based on the following logic. Each applicable learning strategy can create new concepts that other strategies may use; as long as applicable strategies are changing the working memory by adding learned concepts, then each learning strategy must be invoked again, in turn. This LCS is based on an extreme notion of convergence in learning, that is, learning has converged when no changes to concepts occur. Future developments in the LCS will use a more general notion of convergence, where repeated learning strategy invocation may continually improve the quality of learned concepts, up to some desired level of accuracy.

4. BL FRAMEWORK

Our goal is for instructible computers to be immersed in a variety of environments, such as offices, homes, and laboratories, but MABLE requires a test framework to support high throughput experiments during development. The BL framework (depicted in Figure 3) supports the basic interactions with a teacher and world that a digital student like MABLE requires. The "BLEnvironment"

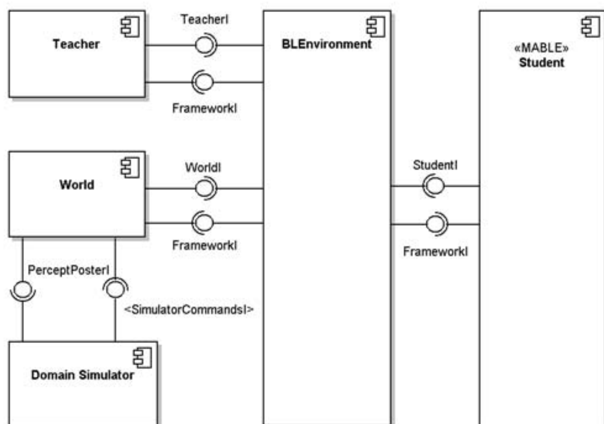

Figure 3: The Bootstrapped Learning Framework.

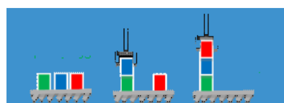

Figure 4: Blocksworld

Figure 5: Robocup 3 vs 2

links the "Student" (MABLE), the "Teacher", and the "World" by organizing learning sessions and collecting performance metrics. As previously mentioned, the teacher (either a software agent or human) can instruct the student via messages that describe gestures, utterances, and imperatives. Both the teacher and the student can observe and affect the world by performing actions. The world i) provides an abstraction from a simulator or actual environment, and ii) maintains a description of the world state. Currently, the BL Framework supports two separate simulators; the classic Blocksworld domain simulator (Figure 4) and the Robocup soccer simulator [6] (Figure 5).

5. LEARNING WITH MABLE

MABLE supports bootstrapped learning from natural instruction through tightly coupled interaction of its various modules, described in Section 3, and interaction with a teacher and world via the framework described in Section 4. This section details how the components interact through an illustrative example taken from Blocksworld. The example curriculum is as follows, learn from examples a predicate "IsAStack", learn by being told a procedure "MakeStack" that works in the case when all blocks are on the table, and learn from feedback a procedure refinement of "MakeStack" that works when the blocks are already in unintended stacks (e.g., the Sussman anomaly [12]). Each concept builds upon a prior concept: a MakeStack procedure uses IsAStack as its goal, and the MakeStack refinement uses the prior MakeStack as a special case. Each concept is taught over a series of learning and testing lessons.

Background Knowledge: Prior to the curriculum, MABLE is provided a minimal amount of background knowledge represented in IL. This knowledge includes IL concepts such as i) the predicate "On(Block, Block)" and the procedure "MoveOnto(Block, Block)", ii) atomic actions such as "Grasp", "Lift", and "Release", and iii) syntactic descriptions of *some* concepts to be learned such as the fact that "IsAStack" has three Block parameters named "top", "middle", and "bottom". This background knowledge serves as the foundation upon which bootstrapped concepts are built. These concepts are stored in the KR from the start of the curriculum.

Learning Lessons: The curriculum is taught by the teacher first supplying lesson options to MABLE, and then MABLE selecting a lesson and the teacher sending a stream of lesson-specific messages to MABLE. All messages are first handled by MABLE's Teacher/Student Interface (TSI); messages are then either redirected to the message bus, if MABLE is not currently executing an EE program, or sent to the EE otherwise (because the messages are relevant to the EE program). Meanwhile, the CM is executing its LCS; when new messages arrive on the message bus, the LCS policy is to invoke a worker strategy to analyze each message. This worker strategy populates the working memory template. Based on the template, the LCS determines the next strategy to invoke. In Blocksworld, the first message is a list of lessons, either an "IsAStack" learning lesson or an "IsAStack" testing lesson. A lesson selection worker strategy is invoked in this case to pick learning lessons, when available, and otherwise pick testing lessons. Picking a lesson corresponds to creating an outgoing message and sending it to the TSI for delivery to the teacher via the timeline. Similar to the other incoming messages, the lesson messages for "IsAStack" end up in the working memory template; the messages are marked as labeled examples of the form: "(IsAStack (top= Block(name =A, support=B), middle=Block(name=B, support=C), Block(name=C, support=T)), true)".

Learning Strategies: At this point, the LCS determines that two learning strategies are applicable: learning by noticing and predicates by example. Learning by noticing is invoked first and notices a symmetry between the blocks and their support with the On predicate. Learning by noticing adds to the working memory a predicate "On(Block1, Block2) := Block1.support=Block2". The CM invokes the predicates by example learning strategy next. The strategy uses both the learned symmetry and the labeled examples to generate two interpretations of a target predicate concept: "IsAStack(top, middle, bottom) := On(top, middle), On(middle, bottom)" and "IsAStack(top, middle, bottom) := top.support=middle, middle.support=bottom". In this case, the strategy determines that the first predicate concept is more succinct and adds it to the working memory. The CM notices that the working memory has changed and that both the learning by noticing and predicates by example are still applicable. The CM invokes both again, but neither update the working memory. At this point, the LCS states that because i) no further messages have been received, ii) there are no outstanding imperatives to answer, and iii) the applicable learning strategies have not generated new concepts, that it should run a worker strategy to signify that MABLE is done with the current lesson and ready for the next.

Testing Lessons: The teacher sends a message to MABLE to indi-

cate the available lessons, now only a testing lesson for "IsAStack" is available. Similar to before, MABLE selects the lesson and is given a stream of messages. The first message is an observation of the world state and the second is an imperative message. The worker strategy for analyzing instruction adds the current world state and the outstanding imperative to the template. Next, the LCS determines that only the learning by noticing strategy is applicable; it is invoked but adds no new knowledge to the knowledge working memory. The LCS then determines that the worker strategy for answering imperatives is applicable and invokes it. The answer imperative strategy first determines whether it has the knowledge to answer the imperative by querying the QE. Determining if the appropriate knowledge exists is non-trivial in most cases because the teacher does not necessarily tell the student the name and parameters for a target concept taught in learning lessons. In this case, the teacher did identify the target concept by name and the predicates by example strategy used the same symbolic name for its learned concept. The answer imperative strategy finds a best matching concept by inspecting symbolic names, parameter signatures, and even temporal relevance. The answer imperative strategy picks the concept learned by the predicates by example strategy and creates an execution engine program with the "IsAStack" predicate definition, and EE dynamically links definitions for the "On" predicate. The EE accepts the program, evaluates the call to "IsAStack", and returns a boolean truth value. The answer imperative strategy takes this return value and sends it as an answer message to the teacher. The teacher grades the answer and sends a grade to the student. In this case the grade is 100% and the student sends a message to the teacher indicating they are done with the lesson.

The next available lessons are for "MakeStack" learning and testing. The learning lesson is chosen and several messages telling the steps of "MakeStack" are received. First is a message "CurrentGoal(IsAStack(top, middle, bottom))", referring to the previously learned concept. Second is a message "First(MoveOnto(middle, bottom))", followed by "Finally(MoveOnto(top, middle))". In the same manner MABLE analyzes the messages and adds them as labeled (the labels are CurrentGoal, First, and Finally) concepts to the template. The LCS determines that the learning by noticing and the procedures by telling strategies are applicable and invokes each in turn. Learning by noticing does not generate any new concepts, but the procedures by telling strategy generates "Procedure-145(top, middle, bottom)" with steps "MoveOnto(middle, bottom), MoveOnto(top, middle)" and a provenance annotation stating that the goal of the procedure is "IsAStack(top, middle, bottom)". The CM tries additional strategies, but none contribute new concepts. MABLE finishes the lesson and starts the next, the testing lesson. MABLE is given a state and the imperative "MakeSo(IsAStack(A, B, C))". The answer imperative strategy has a built in understanding of "MakeSo" and uses the QE to find all procedures; within the set of procedures it looks for those relevant to "IsAStack". It finds that Procedure-145 has a provenance annotation for "IsAStack" as its goal. The strategy creates an EE program that the EE links and evaluates.

Execution: To create a fully interpretable EE program, the program needs extra information, such as the definition for "MoveOnto", which is a procedure consisting of other concepts "Grasp", "Lift", "Move", "Lower", and "Release". The answer imperative strategy makes use of the EE linker to traverse the top-level concepts and find concept dependencies. Unlike the evaluation of a predicate like "IsAStack", evaluating a procedure involves performing world actions. For example, Procedure-145 involves evaluating "MoveOnto" twice with different parameters. Each "MoveOnto" evaluation grounds out to atomic actions, such as "Grasp". These atomic actions are communicated to the TSI by the EE and from there to the world. The result of evaluating an ground action is a state observation, which the EE caches within an execution trace. Upon EE completion the answer imperative strategy sends a message to the teacher indicating it is done. The teacher grades the student by observing the world state and sending a grade.

The final concept is a refinement of the "MakeStack" procedure taught through feedback. The teacher starts the learning lesson by sending a current state observation where C is on A and B is on the table, a message "CurrentGoal(IsAStack(A, B, C))", a gesture to the support of C indicating its relevance, and a message "MakeSo(IsAStack(A, B, C))". The answer imperative strategy has already answered this imperative in the last lesson and does so again, but, unfortunately, the grade returned is zero. The initial state had C on A, making the sub-procedure "MoveOnto(A, B)" fail. MABLE starts the next learning lesson and tries the feedback by score strategy instead. The strategy creates a state and action space from the previously seen state observations and actions (including the learned "Procedure-145"), a reward function from the "CurrentGoal" and the gesture to block C's support, and it invokes the reinforcement learning service. This learning service interacts with the EE to execute actions and maintain state. Eventually the learning service returns to the strategy a procedure called "Procedure-575(top, middle, bottom)". MABLE completes the lesson and takes a testing lesson. MABLE uses the "Procedure-575" to pass the test. MABLE then passes the curriculum because it learned the final concept.

6. RELATED WORK

The Cyc project [8] relates very closely to MABLE in that both are motivated by a need to represent and reason with large and possibly ambiguous knowledge bases. Cyc is aimed at performing the same type common sense reasoning that a human might, whereas MABLE is not quite as broad, performing only in specific domains. The primary difference between MABLE and Cyc is MABLE's emphasis on learning from natural instruction and Cyc's emphasis on reasoning with knowledge (whether learned or hand-coded).

The CALO project [9] seeks to design agents that learn about humans' daily activities and provide assistance to complete repetitive tasks. MABLE addresses a somewhat broader problem in that it can learn to perform tasks autonomously or collaboratively and it learns actively rather than passively from an instructor. While MABLE could, in practice, continually improve performance over time, the emphasis is on having distinct learning and performance periods where passing a curriculum implies mastery.

Soar [7] is a very general cognitive architecture that has been used to learn from instruction [5]. Soar is well-known for its EBL-like universal learning method, called "chunking". Soar has used chunking to associated learned aspects of instruction that are later used for reasoning [5]. Huffman and Laird [5] studied tutorial instruction where the learning agent is given an abstract task and elicits the instructor as needed to learn requisite knowledge. The primary difference between MABLE and Huffman and Laird's work is the underlying architecture; MABLE can cope with uncertainty and ambiguity in the world, instruction, and its own knowledge, where Soar currently does not.

Prodigy [2], unlike Soar and much like MABLE, employs multiple learning methods. Prodigy is mainly aimed at solving planning problems and uses learning to speed-up plan search. Prodigy does support a type of apprentice learning, where the instructor uses a graphical interface to edit domain models and partial plans,

much like a mixed-initiative planner. While many of the target concepts learned by MABLE are procedural, MABLE also supports learning of other domain knowledge. The primary difference is that MABLE learns from natural instruction, whereas Prodigy learns from more-direct instructor manipulation of internal data structures.

More recently, Allen et. al [1] describe PLOW, a task learning agent. PLOW learns from natural instruction, that is spoken natural language, and observations of key strokes and mouse clicks how to perform web browser tasks. PLOW shares many attributes of CALO, but focusses on natural language as an instruction medium.

7. DISCUSSION AND FUTURE WORK

MABLE has been tested on curriculum from both the Blocksworld and Robocup soccer domains. In Blocksworld, the system was able to bootstrap through three lessons in order to learn to make a stack of three blocks. These three lessons included the lessons described in Section 5. In Robocup, MABLE was able to learn how to control a keeper in the 3 versus 2 keepaway sub-game by bootstrapping on four lessons. These included learning what "out of bounds" means from examples, learning how to catch a ball by being told, learning how to compute distance from examples, and finally learning a correct passing strategy by getting feedback from the teacher. We consider these to be great successes in demonstrating the flexibility of the system because these two domains were completed with zero re-programming or configuration changes made between curriculum.

However, we learned a great deal from these experiences concerning the short-comings of the system. Most of these involve missing shared services or additions to services that already exist in the system. Below is a list of some of the additions that need to be made to the system:

Semantic ontology: Although IL is able to represent nearly any type of knowledge that can be taught or learned, it is missing extensions that explicitly support reasoning about the application of this knowledge and its relationship to other knowledge in the system. We intend to create an semantic ontology to support explicit reasoning about the practical usefulness of student knowledge.

Execution engine belief maintenance: The EE currently provides the mechanics necessary to support simple state updates on properties of objects that are in the environment. It does not, however, automatically update lifted or inferred state. We plan to add in a belief maintenance system that will allow components in the system to register predicates and functions that will be updated when the primitive state is altered.

Generalized probabilistic reasoning: MABLE can currently handle simple ambiguity, but lacks the ability to do anything beyond deterministic inference. Although some of the learning services are able to generate a set of weighted theories that cover a given input, the rest of the system must choose the highest weighted one and consider that as fact.

Enhanced control: The current LCS assumes that all information is relevant to a target concept. Directing the focus of attention is an important aspect of controlling learning that will be addressed more aggressively by focussing attention on small portions of working memory. Also, templates and patterns describe the inputs to learning strategies that, when coupled with a target concept, allow a form of back-chaining to determine which inputs to the learning strategy are required to learn the target concept. In the future, missing learning strategy inputs can be supplied through additional means-ends reasoning to identify strategies that could learn the required information, or ask the teacher.

Mixed NIMs: It is our plan to allow for the mixing of natural instruction. For example, we would like the teacher to be able to talk through a demonstration, which is actually a mixing of telling and demonstration. The systems should be flexible enough to handle these types of student/teacher interaction.

Additional domains: Although it is a great success that MABLE works in two domains, we need to test on additional ones to demonstrate and validate the flexibility and generality of the system. Currently plans are to test MABLE on three additional domains.

8. CONCLUSION

MABLE represents a significant step toward human-instructible computing. There are a myriad of challenges that have been faced in taking this step and in this paper we have elucidated many of them and given our first approximations at their solutions. Currently this project is just finishing its first of a three year effort, and we expect that many of our early solutions will need to be improved to be useful for generalized learning from natural instruction. MABLE's framework provides a solid foundation from which to adapt because it was designed with modularity and extendibility in mind. In the near future, we hope to release the first version to the AI community at large and welcome suggestions and improvements from all who wish to join into this growing area of research.

Acknowledgements: This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA/IPTO) 'MABLE: Modular Architecture for Bootstrapped Learning Experiments', issued by DARPA/CMO under contract #HR0011-07-C-0060.

9. REFERENCES

[1] J. F. Allen, N. Chambers, G. Ferguson, L. Galescu, H. Jung, M. D. Swift, and W. Taysom. Plow: A collaborative task learning agent. In *AAAI*, pages 1514–1519. AAAI Press, 2007.

[2] J. G. Carbonell, O. Etzioni, Y. Gil, R. Joseph, C. A. Knoblock, S. Minton, and M. M. Veloso. Prodigy: An integrated architecture for planning and learning. *SIGART Bulletin*, 2(4):51–55, 1991.

[3] D. Corkill. Blackboard Systems. *AI Expert*, 6(9), January 1991.

[4] Daniel Oblinger. Bootstrapped Learning – External Materials. http://www.sainc.com/bl-extmat/, October 2008.

[5] S. B. Huffman and J. E. Laird. Flexibly instructable agents. *Journal of Artificial Intelligence Research*, 3:271–324, 1995.

[6] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. Robocup: The robot world cup initiative. In *AGENTS '97: Proceedings of the first international conference on Autonomous agents*, pages 340–347, New York, NY, USA, 1997. ACM.

[7] J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artif. Intell.*, 33(1):1–64, 1987.

[8] D. B. Lenat, R. V. Guha, K. Pittman, D. Pratt, and M. Shepherd. Cyc: Toward programs with common sense. *Cm. ACM*, 33(8):30–49, 1990.

[9] D. Morley and K. Myers. The spark agent framework. In *Proc. of the Third Int. Joint Conf. on Autonomous Agents and Multi Agent Systems (AAMAS-04)*, pages 712–719, New York, NY, July 2004.

[10] Open Source Initiative. Open Source Initiative OSI - The BSD License:Licensing. http://www.opensource.org/licenses/bsd-license.php, October 2008.

[11] SpringSource. Spring Framework. http://springframework.org/, October 2008.

[12] G. Sussman. *A Computer Model of Skill Acquisition*. Elsevier Science Inc. New York, NY, USA, 1975.