

Augmenting Instructable Computing with Planning Technology

Clayton T. Morrison

University of Arizona
Department of Computer Science
1040 E. 4th Street
Tucson, Arizona 85721

Daniel Bryce

Utah State University
Department of Computer Science
Old Main 414
Logan, Utah 84322

Ian R. Fasel

Antons Rebguns
University of Arizona
Department of Computer Science
1040 E. 4th Street
Tucson, Arizona 85721

Abstract

Advances in human-instructable computing are contributing to a new breed of computer systems that can be taught by natural instruction rather than requiring direct programming. The current approach in the MABLE “electronic student” emphasizes the interface that maps different modes of instruction to machine learning algorithms that can learn the concepts and task knowledge being taught. While the interface provides more natural interaction with the system, there are still many constraints put on how the teacher teaches, in particular in what the teacher can assume about MABLE’s ability to compose previously learned concepts. We present a method for automatically translating MABLE’s learned task knowledge into a STRIPS planning domain, and planner-generated plans back into MABLE’s knowledge representation. In this way, existing planning technology is used to augment MABLE’s problem solving ability. This allows us to relax the requirement that the teacher explicitly teach *every* composite procedure and also provides a role for planning to contribute directly to learning in a more capable student.

Introduction

Human-instructable computing aims to build computational systems that can be taught through natural instruction rather than requiring programming by specially-trained engineers. We are currently working in the DARPA Bootstrapped Learning Project to build a human-instructable “electronic student” called MABLE, the Modular Architecture for Bootstrapped Learning Experiments (Mailler et al. 2009). MABLE consists of a set of learning *strategies* designed to interpret different instruction methods naturally used by humans. These instruction methods include giving declarative definitions and descriptions, providing examples and demonstrations, and giving feedback based on student actions. MABLE interacts with a teacher that provides instruction by using these methods to teach concepts that build on one another – *bootstrap* – to more complex concepts and skills. These concepts, in turn, are represented in MABLE’s knowledge representation language, *Interlingua*.

However, there are still a number of unnatural requirements placed on the teacher of MABLE. In the current system, lessons teach self-contained concepts, such as the actions corresponding to steps in a procedure. The lessons, however, tend to be specific to what is involved in executing the action and do not always include information about the action’s effects and preconditions. For MABLE to understand how to compose its previously learned actions into a composite procedure, the teacher *must* give an explicit later lesson that wraps the actions as steps in an HTN-like procedure.

We have developed a learning strategy called *Learning by Noticing* (LbN) whose job is to identify patterns in teaching and concept use that might be otherwise implicit – i.e., not directly included in the teacher’s utterances or part of the target concept being taught in the lesson. By observing the lessons that teach atomic steps in a procedure, LbN constructs action models that fill in action effects and in some cases preconditions of actions. These action models provide the raw materials for defining planning operators.

To make full use of these nascent planning operators, we have developed a process that translates these action models into the PDDL planning language (McDermott and the AIPS’98 Planning Competition Committee 1998) so that compound procedures using the component actions as steps can be identified through planning. We target planning for the STRIPS domain and use an implementation of the Graphplan planning algorithm as our planner (Blum and Furst 1997). When planning is successful, the plan produced by the planner is translated back into Interlingua and incorporated into MABLE’s knowledge base. In this way, LbN action model construction combined with planning technology *relaxes* the need for the teacher to necessarily teach every compound procedure through an additional explicit lesson.

For the purposes of the ICKEPS competition, our method is best viewed as a service translating concepts taught in MABLE’s Interlingua (IL) knowledge representation to PDDL and PDDL-expressed plans back to IL. A key contribution of our method is that it is not merely translating IL to and from PDDL but also performs inference to fill in the action models for the learned Interlingua task knowledge. From a broader perspective, although we are translating from IL to PDDL, the vision is for translating from human interaction to formal concept representation and pro-

cess languages like PDDL, in a human-instructable computing framework. We believe our tool demonstrates the benefits of both action model learning as well as the benefits of light weight planning for instructable computing, and the potential for instructable computing to provide a natural human interface for teaching planning knowledge.

The following sections present a brief overview of the MABLE architecture and its learning environment, followed by a description of IL, the language underlying MABLE's knowledge representation. We then present how LbN constructs action models, followed by a discussion of our method for translating from IL to PDDL, and planner-generated plans back into IL for MABLE's use. We conclude with a discussion of future directions for exploiting the synergy between instructable computing and planning.

MABLE and the Learning Environment

MABLE is currently being developed within a larger interaction framework that includes a simulated Environment and Teacher. All three components interact with each other on a message *Timeline*.

The *Environment* simulates a perceivable world, keeping track of its state and any changes produced by actions from the Teacher or MABLE. The Environment posts perceptual update messages representing the current world state to the Timeline. A variety of simulators are currently available for use as the Environment. These include a version of the classic "blocks world", the 2-dimensional robocup soccer simulator (Kitano et al. 1997), a simulation of an unmanned aerial vehicle in a 3-dimensional world, a 2-dimensional tactics-level wargame simulation, and a simulation of the system control for the International Space Station. For exposition, we take the bulk of our examples from the more familiar blocks world simulator.

A set of structured *curricula* have been authored by humans to teach various concepts in each of the different simulator domains. Some concepts depend on others, so the curricula are decomposed into sets of lessons that depend on one another. Each lesson aims to teach one concept. These lessons form "rungs" in a partially-ordered curriculum "ladder", with some lessons making use of the concepts taught in earlier lesson rungs. Lessons themselves also have structure. They begin with a set of *teaching epochs*, in which the Teacher teaches a concept according to one of the natural instruction methods. These are followed by a *testing epoch*, in which the Teacher sends messages including an imperative for MABLE to answer questions or perform actions. A grade is assessed according to MABLE's performance in a test.

The simulated *Teacher* itself executes teaching and testing epochs as a script, with some allowance for interaction based on MABLE's possible response messages. The scripts manage initializing the Environment state and the generation of Teacher messages. Teacher messages include utterances and imperatives, as well as action messages that induce changes in the Environment world state.

MABLE's task is to observe the messages that come across the Timeline and use them to build a model of the current state of teaching (and testing), using the messages from the Teacher and percept updates from the Environment to

learn. The Mable architecture itself consists of a set of modules that work together to incrementally learn concepts from the Timeline messages. A set of learning *strategies* provide component interfaces to the different methods the teacher uses for teaching concepts. For example, the *ProcedureByTelling* strategy interprets declarative teacher utterances to construct the component steps in a procedure; on the other hand, the *ConditionByExample* strategy identifies how the teacher is providing examples of a rule. In all cases, learning strategies extract, interpret and repackage data from the Teacher and Environment messages to construct concepts. Learned concepts are represented in IL and stored in MABLE's knowledge repository. When appropriate, strategies invoke learning *services*, dedicated machine learning algorithms that can help with concept learning. For example, (i) a predicate learning service is driven by inductive logic programming, (ii) regression algorithms can learn numeric functions, and (iii) reinforcement learning is used to learn policies by feedback. All module activities are coordinated by a *control* module, which has the job of identifying learning targets, ensuring the appropriate learning strategies and services are invoked, and ensures that progress is being made toward learning the target concept in order to perform well in the testing epoch. Finally, MABLE's *execution engine* is used to execute IL and monitor procedure execution, for example when a learning strategy requests to evaluate an IL expression, or in order to answer an imperative messages sent by the Teacher. We will return to the execution engine in the next section.

Interlingua

The *Interlingua* (IL) language has been designed to accommodate a broad spectrum of duties (Oblinger 2008). IL provides the building blocks necessary to construct the variety of concepts MABLE will need to represent as background knowledge and as the result of learning. These include rules, a type hierarchy of classes, functions, and procedures. IL is also expressive enough to define other languages as extensions within IL. In particular, the *Interaction Language* (ITL) is a specialized IL extension that is used to represent all messages that appear on the Timeline between MABLE, the Teacher and the Environment. Here we present the components of IL and ITL that we need as background to explain how models of actions are constructed and how we will translate the models and problem instances into PDDL.

Core IL consists of three component languages. The first of these is the *syntax language* and is used to define types, using the **is** construct, and properties, using the **arg** construct. For example, the expression

```
is Dog Animal;
```

defines a type called *Dog* that is a sub-type of *Animal*. Properties associate values with instances of a type (called the property's *host* type). They are named and also impose a constraint on the type of values that may be bound to them. For example, the expression

```
arg Dog age Integer;
```

defines a property of a *Dog* called *age*, and only *Integers* can be associated with a *Dog*'s *age*. All IL types that have

properties associated with them are called *composites*; `Dog` is therefore a composite. IL also defines a set of *atomic* types that have no properties associated with them (and therefore no property-based composite structure); IL provides familiar atomic types such as Numbers, Integers, Floats, Booleans, Strings and Symbols. All property values can also be bound to a special atomic `Null` value, irrespective of the type constraints placed on the property value; this is a feature of IL that we will return to, below.

Like many other typed object-oriented programming languages (e.g., Java), properties associated with inherited types are also inherited and associated with the new type. A new type may have new properties associated with it, which are then added to the list of properties inherited, or a property of a new type may *restrict* inherited properties by giving them new, compatible type restrictions. In the case of restriction, the new type restriction must be a subtype of the type restriction of the inherited property.

Unlike many languages, however, IL also allows for multiple inheritance. That is, a type *A* might inherit from types *B* and *C*. More formally, the *is* type assertion is reflexive and transitive, but not symmetric, and combined with the possibility of multiple inheritance, this means the overall type hierarchy structure is that of a directed acyclic graph (DAG). Figure 1 shows a portion of the type hierarchy for the blocks world domain. Here, the `Object` type inherits from both the `Composite` and `Percept` types. Multiple inheritance poses a critical challenge to translating IL concepts into typical planning domain representations such as PDDL; we will address this in our translation, below.

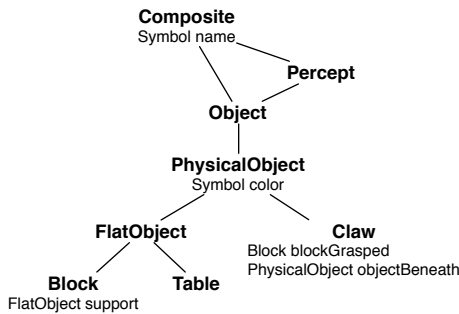


Figure 1: Portion of the blocks world type hierarchy; property definitions are displayed below their host type, with the property value type constraint followed by the property name.

For convenience, IL defines a “macro” called `defSyntax` that combines type inheritance and property associations of a type in one expression. For example,

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

defines our `Dog` with its `age` property, and also includes the property `color`, which is a `Symbol`; a `defSyntax` may include multiple extensions for multiple inheritance.

The second core IL language, the *instance encoding language*, is used to express ground instances and their property

value bindings, as they exist in real or simulated worlds at a particular time. For example

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

describes a `Dog` named `Rover` that is age 3 and is black in color.

Finally, the third core language of IL is the *code body language*. This language specifies functions and procedures that can be executed. An expression defining a code body includes specification of a code body *interpreter* which handles executing the code body. IL provides several code interpreters, including one for evaluating functions, another for running recursive procedures that are formed with combinations of control statements such as “if” and “while”, and an interpreter for evaluating predicates defined in first-order predicate logic.

The following example defines a function to add 1 to an input integer:

```
defSyntax Add1 extends Function
  (Integer arg);
defCode Add1 FunctionEngine
  FunctionBody(Return(Plus(arg,1)));
```

The `defSyntax` defines a new type of `Function` called `Add1`, and it has the property `arg` that is an `Integer`. The corresponding `defCode` specifies that the body of the code will be interpreted and executed by the `FunctionEngine`. The `FunctionEngine` knows how to execute the `Plus` function, which takes two argument; the reference to `arg` is interpreted as whatever value is bound to the `arg` property in an instance of the `Add1` type. In this sense, properties and their values are treated by the code interpreters as arguments for the code body. The return value of `Plus` is then returned as the value of executing `Add1`.

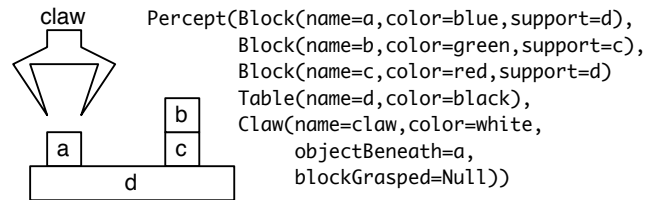


Figure 2: Example blocks world state and its IL `Percept` representation.

Interaction Language

We finish this summary of IL by presenting components of the Interaction Language (ITL) we will use in our discussion below. ITL is a specialized language built within IL that defines the composite terms used for interaction between the Teacher, Environment and MABLE. `Percept` expressions are generated by the Environment simulator and describe the current state of the environment. `Percepts` wrap ground IL instance expressions that describe the state of the world. Figure 2 depicts an example blocks world state and the corresponding `Percept` message describing it.¹

¹In order to make our examples of IL instance expressions more concise, we will often drop properties/value pairs that would other-

Observations of the execution of actions that affect the Environment simulator are presented in a different form. First, in addition to the specialized executable language terms (such as `Plus`), MABLE is also provided with a set of primitive simulator commands. These are defined in the same way as other executables (with `defSyntax` and `defCode`), but their execution interpreter specifies the Environment simulator and they are interpreted as direct commands to the simulator. While the `defSyntax` tells MABLE what arguments the action takes and their general type constraints, it does not say how the actions are to be used, i.e., what values should be bound to arguments; these bindings are taught by the Teacher or acquired by observation (by the LbN learning service).

MABLE observes the Teacher executing an action when a Timeline message contains a special ITL term, like the following:

```
TeacherAction(action=Grasp());
```

This indicates that the `Grasp` action has been executed. Only primitive simulator actions will show up in these messages. In order for the Teacher to specify that a non-primitive action is being executed (e.g., one that the Teacher is currently teaching MABLE), an utterance like the following is used:

```
Utter(utterance=WatchMe(DoWith(
  MoveOnto(Block(name=a),
            Block(name=b))));
```

Finally, ITL *imperatives* are used to direct an agent to execute some command or code body. The Teacher and MABLE can both send messages with imperatives to the Environment in order to execute simulator commands. When an imperative is directed from the Teacher to MABLE, it is a request for MABLE to execute a code body. In most cases the imperative request directly names the action the Teacher expects MABLE to have learned a code body for and to execute. However, the Teacher can also use the special imperative `MakeSo` to specify a world state the Teacher would like MABLE to achieve.

The Limits of Execution

We can now state precisely what MABLE is missing that the LbN learning service partially provides and that our planning language translation service completes. MABLE's execution engine executes IL code bodies either when prompted by a learning strategy or service, or when the Teacher requests MABLE to do so. In general, these calls specify, in the instance encoding language, the name of the form to be executed along with any values bound to properties (in this case, treated as arguments). The execution engine does not itself know how to select which action to execute without this information.² The procedure interpretation language

wise be present in the expression. For example, the contents of the `Percept` in Figure 2 would also include the `Percept` instance name, and the list of objects would be enclosed as a list bound to the `perceptsGained` property).

²Internally, MABLE's knowledge repository keeps track of multiple hypothesized learned code definitions, and a process guided by control selects the "best" current hypothesis.

does include control flow constructs and recursive procedure and function calling, but this machinery only works once the procedure has been selected for execution. The execution engine itself does not have a mechanism for determining that one or more executable code bodies could be used to achieve some specified state or result, such as that specified in a `MakeSo` imperative. In the next section, we describe how the LbN learning strategy learns action models that can answer Teacher `MakeSo` imperatives.

Learning by Noticing

In general, MABLE's learning strategies respond to and do their work based on specific classes of messages and percepts from the Teacher and world. A notable exception is the *Learning by Noticing* (LbN) learning strategy. LbN's primary job is to analyze incoming messages and mine them for patterns that may indicate information relevant to learning that is not otherwise explicitly expressed in teacher utterances or world percepts. When LbN identifies such patterns, it makes them explicit by posting the findings in special `Relevant` messages and as hypothesized new concepts, both made available to the other learning strategies in MABLE.

LbN cannot look for all patterns, so it relies on a set of heuristics to decide what kinds of patterns to search for. One of LbN's core heuristics is to look for changes in the world state that result from observed actions. This requires building a transition model of world states. For the purpose of building models of actions, this transition model is based on `Percept` messages describing world state changes. (Other heuristics also use Teacher utterances and imperatives.)

LbN treats the sequence of `Percept` messages that come over the Timeline as a time series of structured objects. To construct pattern identifiers, LbN decomposes this set of structured objects into a set of atomic events over which random variables are defined. For our presentation here, we focus on representing instances of properties and their value binding. Each atomic event is defined as a pattern consisting of four pieces of information: the type of the host composite (the composite the property is associated with), the composite's unique identifier (its name), the property name, and the value currently bound to the property.³ If the property value is itself another composite object (as opposed to an atomic value), then the value is represented by the name of the composite. For example, the following instance

```
Block(name=block-47,
      support=Table(name=table-3)),
```

is represented as the property-binding event

```
[Block, block-47, support] <= table-3
```

These events are also used to build generalized patterns, for example by "wild-carding" the name of the object instance:

```
[Block, *, support] <= table-3
```

³We ignore here two other classes of random variables: representing how many instances of a type exist at a given time, and the values of derived relations resulting from other functions and predicates, such as *Near* and *Distance*.

This pattern matches any instance of a `Block` whose `support` property is bound to table-3. These event patterns are used to define random variables whose values can be tracked over time as well as used to define probabilities.

The final representation of random variables resembles a discrete-time sensor readout, where the “sensors” are random variables determined by the event patterns that exist in the world at each time they are sampled.

For building models of the effects of actions, LbN keeps track of how the random variables change when actions are executed. A minor complication here is that there may be several `Percept` message updates about the world state between each action. To handle this, LbN divides the `Timeline` of `Percept` messages into chunks based on the boundaries of lessons and the time ticks at which actions are executed: the first chunk consists of the `percepts` between the start of the lesson and the first action; the second chunk between that first action and the second, and so on, with the effects of the last action being the world state updates between the last action execution and end of the lesson. Although multiple `Percept` updates may happen within the chunk, LbN only pays attention to the state of the world by the end of the chunk.⁴ In this work, all primitive actions are atomic and their effects are assumed to occur within the chunk directly after the action. An action a taken at time t is denoted a_t . The world state just prior to the action is represented by the world state changes of the previous chunk of `Percept` updates up to time t , and is denoted s_t . The world state after executing a_t is s_{t+1} and is represented by the world state at the end of the `Percept` update chunk after t , but before action a_{t+1} (or the end of the lesson).

The random variables defined for s_t and their changes from s_t to s_{t+1} provide for a variety of models of how actions effect changes in the world. LbN makes use of several heuristics to identify whether actions may be relevant to some aspect of the unfolding world state. However, in order to meet the demands of constructing action models that conform to the semantics of STRIPS planning operators, LbN uses the following two-stage construction process.

First, LbN keeps track of the values of each property-binding event before and after an action is taken and then looks for cases where taking an action *consistently* causes the properties to change from one value to another. LbN collects statistics for such transitions, and when the probability of *change*, $P(X_{t+1} \neq u | a_t, X_t = u)$, exceeds a threshold, the random variable involved in the transition is treated as a candidate effect of the action. These candidate random variables are collected for each action.

In the second phase, LbN then analyzes for each action the set of random variables identified as candidate effects. The random variables denote properties that are believed to be consistently affected by the execution of the action.

⁴Because a number of `Percept` updates might have occurred since the start of the chunk, this means LbN may be ignoring important information about the dynamics of world changes between any two actions. This is a topic of ongoing research; in the work we present here, this information loss has not affected LbN’s overall ability to construct action models, but we expect it may in the future.

For these candidate effects to be considered as a model of a STRIPS operator, the properties themselves must be “hosted” by (i.e., belong to) a composite instance that satisfies one of the following *identifiability* criteria:

1. The host instance is always unique in every situation in which the action is taken (e.g., there is always one and only one `Claw` in the blocks world simulation). LbN identifies this condition by counting the number of instances of a type it observes; if there is always one and only one, then the host instance of the property satisfies this criterion.
2. The host instance is the value of one of the arguments to the action.
3. The host instance appears as one of the values bound to another property in the set of candidate effects.

The purpose of this test is to ensure we can always identify the relevant properties that are changing as a result of the action, even if their host composite changes from one action execution to another (as in criteria 2 and 3).

If this test is satisfied for each candidate effects property, then LbN can *lift* the representation of the candidate effects, replacing the (non-Null) ground values in the specific action execution instances with variables (leaving Null values in place); host composites are also consistently replaced with variables. The *same* variables are used within the set of candidate effects properties when the following condition holds: Across all of the instances of the action being taken, the (non-Null) value of candidate effect property **A** in state s_t always appears bound in state s_{t+1} to candidate effect property **B**. All other ground values are assigned different random variables.

As an example, suppose LbN identifies that the `Release` action affects the random variables representing the property bindings for the `support` property of a `Block` and the `blockGrasped` and `objectBeneath` properties of the `Claw`, so that for a particular instance, the changes to the properties are as follows:

```
Block(name=a) support :
  Null -> Table(name=d)
Claw blockGrasped :
  Block(name=a) -> Null
Claw objectBeneath :
  Table(name=d) -> Block(name=a)
```

Each of the properties satisfies one of the *identifiability* criteria. *If* this same pattern of properties and value changes consistently occurs for all instances of the `Release` action, then the following lifted representation of the action model is produced:

```
?a support :
  Null -> ?b
?c blockGrasped :
  ?a -> Null
?c objectBeneath :
  ?b -> ?a
```

The variables are also typed. Variables that replace host composites of properties are given the type of the host composite, and variables replacing property values are given the

type of the property’s value type constraint. In the above example, this means that `?a` is of type `Block` and `c` is of type `Claw`, according to the type hierarchy and property assignments in Figure 1. `?b` poses a conflict because as the outcome of the change in the `support` property, it is constrained to be a `Block`, but as the prior value of `objectBeneath`, it is a `PhysicalObject`. As long as the IL type definitions are consistent, then one type will always be a subtype of another. In such cases, we always choose the more restrictive type, so `?b` is constrained to be of type `Block`.

Identifying the effects of `MoveClaw` provides another illustration. In this case, `MoveClaw` takes a single argument, constrained to be a `PhysicalObject`, which according to the type hierarchy in Figure 1 includes `Blocks`, `Tables` and even the `Claw` itself. `LbN`, however, has already identified that the argument always ends up being the value bound to the `Claw`’s `objectBeneath` property after the action

```
MoveClaw arg1 = Block(name=a)
Claw objectBeneath :
  Table(name=d) -> Block(name=a)
```

Again, the property satisfies one of the identifiability criteria. In this case, `Block(name=a)` is replaced in two places by the same variable, but the `Table(name=d)` gets a unique variable:

```
MoveClaw arg1 = ?a
?b objectBeneath :
  ?c -> ?a
```

Again, the variables are typed according to the roles they play in the property definition. And as with variables `?b` of `Release`, the conflict between possible type constraints for variable `?a` is chosen to be the more restrictive `Block`.

`LbN` can identify and construct these models only when effects are consistently produced by an action, and only when enough data are presented in the form of action executions with observed prior and outcome world states. But when the data is available and `LbN` does identify these transitions, then `LbN` can provide lifted action models that can be used for STRIPS planning.

A PDDL Translation Service for Interlingua

After teaching a set of actions through a series of lessons, the Teacher may request that `MABLEMakeSo` some world state. IL provides a variety of terms that can be used to describe a world state, but here we focus on two: `SameAs` and `Of`. `SameAs` is a predicate that asserts that two arguments are the same. `Of` also takes two arguments, where the first gives the symbol name of a property, and the second gives the name of a composite instance. In this way, `Of` is used to refer to the value of a property of a composite instance. The following example combines these terms:

```
MakeSo(SameAs(Of(support, a), b))
```

This imperative requests that `MABLE` transform the current state (which for the following example we will assume is that depicted in Figure 2) into the state where the support of `a`, which happens to be a `Block`, is `b`, which also happens to be a `Block`.

This triggers `MABLE`’s control module to execute the IL-to-PDDL translation service. The first task this service engages is to create the domain model, translating from IL concepts and action models into a PDDL domain model. The translation service executes the following steps:

STEP 1: Identify action models and associated types and properties. We do not need to translate the entire `MABLE` knowledge repository; nor do we need to translate all of the elements currently observed within the world state. Instead, what we translate will be driven by the current set of action models provided by `LbN`. For this working example, we assume that in addition to the action models defined for `Release` and `MoveClaw`, `LbN` has also formed a model for `Grasp`:

```
?a support :
  ?b -> Null
?c blockGrasped :
  Null -> ?b
?c objectBeneath :
  ?b -> ?a
```

The translation service takes these definitions and collects all of the different types and properties referenced. Only these types and properties need to be translated to the PDDL domain definition, and only current world objects that are members of these types need to be translated to the PDDL problem definition. Any other objects in the world and any other types in the `MABLE` knowledge base are irrelevant – they play no role in the current action effects model, so won’t help in planning.

STEP 2: Translate IL types into PDDL. The IL multiple inheritance type system is not directly compatible with most standard planning domain representations, such as PDDL, because they are restricted to single inheritance. There is an active strand of research that is looking at various methods for translating multiple inheritance into single inheritance systems (Dao et al. 2004; Crespo, Marquès, and Rodriguez 2002). However, for our purposes here, rather than using the PDDL `:types` features, instead we treat types as a property of objects in the PDDL domain, and therefore translate types as PDDL domain predicates. In order to ensure there are no accidental name clashes in the translation from existing IL type and object names, the following naming convention is used: each IL type is translated to a PDDL domain predicate by appending `_t-` to the IL type name to create the predicate name. For example, the IL type `Block` is translated as: `(_t-Block ?t)` (The variable name doesn’t matter here).

In the PDDL problem domain, types must then be asserted for each object, in the `:init` clause. Each object is not just an instance of its base type, but also every ancestor type the base type inherits from. A predicate assertion is added for each ancestor. Thus, a `Block` object named `b` will have the following list of type assertions: `(_t-Block b)`, `(_t-FlatObject b)`, `(_t-PhysicalObject b)`, `(_t-Object b)`, `(_t-Percept b)`, and `(_t-Composite b)`.

STEP 3: Translate IL properties into PDDL. IL Properties naturally translate to PDDL predicates in the `:predicates` domain definition. For example, the prop-

erty support associated with `Block` is translated as (again using a special naming convention): `(_p-support ?b ?f)`. However, this definition does not itself put constraints on the two variables. Type checking will now be moved into the action precondition clause, and it is up to the translation process to keep track of the appropriate type constraints. For example, if the support predicate is asserted in an action precondition as `(_p-support ?b ?f)`, then the translation process must also include the type constraint assertions on `?b` and `?f`: `(_t-Block ?b)`, `(_t-FlatObject ?f)`.

All properties can be bound to the special `Null` value. Rather than reify `Null` as a special object, instead we treat it as a predicate, one for each property. For example, the condition of the support property being set to `Null` is translated as: `(_isNull-support ?s)`.

STEP 4: Translate Action Models into PDDL. The basic building blocks for translating IL action models to PDDL have already been defined. The lifted action models constructed by LbN specify the properties that change. Each component property change model specifies four things: the property that changes, the host composite the property is associated with, and the value of property before the action (at s_t) and after (s_{t+1}). For example, the support property in the `Release` action model has been lifted so that the host composite is represented by variable `?a`, which is constrained to be of type `Block`, and in s_t it is `Null`, and then in s_{t+1} becomes set to the value of variable `?b`, which is constrained to be type `Block`. To represent this change, the prior value of support is asserted in the `:precondition` clause of `Release` as an instance of the `_p-support` predicate with its associated action-model-assigned value. If the value assignment was a variable, that would be asserted along with the host composite variable. But in this case, the support is `Null`, so the `_isNull-support` predicate is asserted: `(_isNull-support ?a)`. For the effect outcome, the resulting value of support at s_{t+1} is asserted as `(_p-support ?a ?b)` in the `:effect` clause for `Release`. We also need to negate the `isNull` predicate: `(not (_isNull-support ?a))`. Once these precondition and effect predicate assertions are made, we assert the type constraint predicates for any variables mentioned in the precondition clause, in this case for `?a` and `?b`. Finally, any variables we have mentioned so far are added to the `:parameters` clause. We repeat the above translation for each of the property effects components of `Release`, in this case for `blockGrasped` and `objectBeneath`. The final translated action definitions is as follows:

```
(:action Release
:parameters (?a ?b ?c)
:precondition
  (and (_t-Block ?a)
        (_t-Block ?b)
        (_t-Block ?c)
        (_isNull-support ?a)
        (_p-blockGrasped ?c ?a)
        (_p-objectBeneath ?c ?b))
:effects
  (and (not (_isNull-support ?a))
```

```
(_p-support ?a ?b)
(not (_p-blockGrasped ?c ?a))
(_isNull-blockGrasped ?c)
(not (_p-objectBeneath ?c ?b))
(_p-objectBeneath ?c ?a) ))
```

The above steps are repeated for each action model provided by LbN. This, along with the prior steps, completes the domain definition.

STEP 5: Complete Problem Definition. The final step in the translation process is to complete the problem definition. First, all of the objects that are of the types identified in *STEP 1* are added as to the `:objects` clause; objects are added according to the name. For the world state represented in Figure 2, this would include: `a`, `b`, `c`, `d` and `claw`. As described in *STEP 2*, ground type predicates are asserted for all of the types these objects inherit.

Finally, the `MakeSo` request is translated as described at the beginning of the section and asserted as the ground formula in the `:goal` clause.

This completes the the translation to PDDL. The PDDL form is then provided to a planner (in our case, an implementation of Graphplan), and a plan is produced. The plan produced by the planner consists of a sequence of ground actions. For the example we have constructed throughout this section, the plan is:

```
((MoveClaw C Claw Table))
((Grasp C Claw Table))
((MoveClaw A Claw Table))
((Release Claw C A))
```

The translation schemes between the IL and PDDL versions of the actions make the translation back to IL simple, producing:

```
MoveClaw(C)
Grasp()
MoveClaw(A)
Release()
```

This ground plan may solve the particular problem instance defined in the problem. However, we generalize this solution using the same lifting technique we used above.

This new capability also opens the possibility for the Teacher to make use of MABLE's planning ability to teach compound actions whose defcode procedure consists of the steps in the plan. To do this, the Teacher can provides an explicit name and syntax for the action to be learned, along with arguments corresponding to the arguments the teacher expects the action would take to achieve the `MakeSo` request. For example:

```
DoWith(MoveOnto(a,b))
```

Combined with the lifted solution plan, the next composite procedure learned by MABLE is as follows:

```
MoveOnto(?x, ?y)
Do(InSequence(
  MoveClaw(?x),
  Grasp(),
  MoveClaw(?y),
  Release() ));
```

Of course, this learned procedure is not necessarily general enough. For example, there are a number of preconditions not represented here:

1. The Claw may already be above the block to be moved, so the initial move isn't needed.
2. The Claw may currently grasp an object, so need to first release the currently held block, and do so *not* above the block that is to be moved.
3. The Claw may already have the target block grasped.

These conditions still need to be learned (either through trial and error or further instruction)

Conclusion

Our ability to successfully create a plan, execute it and have it successfully achieve the target goal given the current state, is based entirely on how *complete* and *accurate* our current action models are.

There are (at least) two places where our action models may fail us: (1) An *incomplete model for planning*: our action models may not be complete in the sense that they may not provide enough information for the planner to identify a sequence of actions that will transform the world state to the goal state; and (2) an *incomplete or incorrect model of the world*: our action models may also be inaccurate with respect to the world: we may think an action will bring about some change in the world when in fact it either doesn't, or the effect is conditional on other world states being true or other actions taken, or our actions may have other effects we haven't represented. Also, successfully constructing a plan and achieving the world state does not mean our action models are correct, as pointed out at the end of the last section.

All of these are possibilities and we won't know without trying to form a plan and executing it. In this sense, forming a planning model for a given problem description in IL, attempting to build the plan, and then attempting to execute the plan, are all components of an experiment. Failure at any step of the process from translation, to planning, to execution can be very informative: (1) Failure during planning may provide information about deficits in the completeness of action models, and could lead to questions and other tests or explorations to fill out our action models. This is a topic for future work, likely including looking a plan-generation traces and analyzing the partial plan graph; and (2) Successfully generating a plan but then failing during execution provides information about where things might go wrong. Here, we will want to look at the execution trace and analyze where the world appears to have diverged from the expected plan execution model.

The translation framework we have presented here (in detail!) still leaves quite a number of questions unanswered and many directions for improvement. The following are directions for future work:

1. Augment the planner to analyze a partial plan graph or plan trace in cases of planning failure.
2. Handle conditional plans: take different actions depend on results of a test

3. Handle planning with objects that may be "created" and "destroyed" by actions. For example, the action of "cutting" a piece of paper in half in a sense "destroys" the original paper and now produces two separate objects. This is a deep issue in planning domain knowledge engineering, but one that will have to be addressed in full human-instructable computing in the real world.
4. Handle numeric values; MABLE learns numeric functions and many of the domains involve numeric property values that are affected by actions.
5. Finally, handling IL lists. They will likely be handled similar to how other composite objects are manipulated, but a special set of list manipulation actions need to be defined and appropriately represented.

Acknowledgments

This work was supported in part by contract HR0011-07-C-0060 with the Defense Advanced Research Projects Agency (DARPA) as part of the DARPA Bootstrapped Learning Program.

References

- Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.
- Crespo, Y.; Marquès, J.; and Rodriguez, J. 2002. On the translation of multiple inheritance hierarchies into single inheritance hierarchies. In Black; Ernst; Grogono; and Sakkinen., eds., *Proceedings of th Inheritance Workshop at ECOOP 2002*, 30–37.
- Dao, M.; Huchard, M.; Libourel, T.; Pons, A.; and Villerd, J. 2004. Proposals for multiple to single inheritance transformation. In *Proceedings of MASPEGHI'04: 3rd Workshop on Managing SPECIALIZATION/Generalization Hierarchies*.
- Kitano, H.; Asada, M.; Kuniyoshi, Y.; Noda, I.; and Osawa, E. 1997. Robocup: The robot world cup initiative. In *Proceedings of the first international conference on autonomous agents*, 340–347. New York, NY: ACM.
- Mailler, R.; Bryce, D.; Shen, J.; and Oreilly, C. 2009. MABLE: A modular architecture for bootstrapped learning. In *The Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS09)*.
- McDermott, D., and the AIPS'98 Planning Competition Committee. 1998. PDDL - the planning domain definition language. Technical report, Yale University, Available at: www.cs.yale.edu/homes/dvm.
- Oblinger, D. 2008. Bootstrapped learning – external materials. Technical report, <http://www.sainc.com/bl-extmat/>.