An XML-based Forward-Compatible Framework for Planning System Extensions and Domain Problem Specification

Eric Cesar E. Vidal, Jr. and Alexander Nareyek

NUS Games Lab
Interactive and Digital Media Institute, National University of Singapore
21 Heng Mui Keng Terrace, Level 2, Singapore 119613
ericvidal@nus.edu.sg, elean@nus.edu.sg

Abstract

Real-world planning problems, e.g., planning for virtual characters in computer games, typically come with a set of very specific domain constraints that may require specialized processing, like symbolic path planning, numerical attributes, etc. These specific application requirements make it necessary for planning systems to have an extensible design. We present a framework for a planning system that recognizes *planning extensions* (such as new data types or structures, sensing/acting functionality, and others). The framework is designed to be forward-compatible, exposing an XML-based domain language that allows current and future problems that use such planning extensions to be properly specified.

Introduction

In artificial intelligence, *planning* is a problem where, given a set of goals and possible actions, the necessary actions are to be determined (along with their proper temporal arrangement) to attain the given goals. A logistics problem, for example, can define a set of possible actions, such as "move a vehicle V from location A to location B" or "load/unload package P in vehicle V", and a set of goals such as "deliver n packages, labeled $P_{1..n}$ from locations $A_{1..n}$ to locations $B_{1..n}$". A solution is called a *plan*; in this case, the plan would involve multiple "move" and "load/unload" actions on a correctly-ordered, non-conflicting schedule. A plan is considered *valid* when it reaches the goal state without inconsistencies such as violations of action preconditions (e.g., moving a package requires the package to be loaded first) or forbidden overlapping of actions (e.g., a package cannot be unloaded while the vehicle is moving). A *planning system* or *planner* is a program that automatically creates such plans.

Many planning problems are simple enough such that a general (i.e., *domain-independent*) planning system is not needed, e.g., path planning in most computer games is usually implemented as a simple A* search. On the other hand, more complex problems (e.g., planning a dynamically-generated story for a computer game) can benefit from the solving capabilities of a general planning system. In order to do this, the properties of the problem must be formalized into a *domain definition*, using a specification language that can be understood by a planner.

To solve real-world problems, however, a general planning system usually needs to be extended to handle specific application requirements. A computer game, for example, will require extensions such as *online planning* (i.e., feeding the planner-selected actions into the game, and then sensing in real time the current state of the game world, updating the plan accordingly), *numerical resources* such as player health or money, and *specialized solving heuristics* to let the planner more efficiently handle specific sub-problems like symbolic path planning (where symbols are mapped to actual positions in the world, for faster planner reasoning about connectivity and distances compared to regular path planning). A planner written without such extensibility in mind will invariably need continuous re-design to handle these and future extensions. A better solution, from a software engineering point of view, is to adhere to a framework that readily integrates such extensions, making a planner *forward-compatible* with current and future planning problems.

Many such extensions may expose new planning constructs—for example, online planning introduces the concept of *actuators* and *sensors* into the domain ontology—thus making it necessary for the extensible planner architecture to tie in seamlessly with its domain specification language.

Background

In this section, we introduce the issues that accompany the design of an extensible planning architecture, and issues related to the domain specification of planning extensions.

This work was supported by the Singapore National Research Foundation Interactive Digital Media R&D Program under research grant NRF2007IDM-IDM002-051.

Monolithic versus General-Search-based Planning

We first discuss existing planning approaches to establish the context of our planner extensibility problem. Different planning approaches vary in the degree they can be extended.

Systems such as STRIPS (Fikes and Nilsson 1971) and Graphplan (Blum and Furst 1997) are *monolithic* systems. Although these systems are extensible to a certain degree, these systems use relatively rigid planning frameworks that are often optimized to exploit a particular problem representation and are not specifically designed with extensions in mind. Thus, the possibility of extending such systems ranges from impractical to impossible.

Planners that map to *general search frameworks* like propositional satisfiability (SAT), integer linear programming (ILP) or constraint programming (CP) can usually handle planning extensions much more easily, although they are often not as expressive as monolithic approaches for specific domains. SAT-based systems, such as Blackbox (Kautz and Selman 1998) and SatPlan-2006 (Kautz, Selman, and Hoffman 2006), can handle Boolean propositions, which somewhat limits the types of problems that can be expressed. ILP-based planners, such as LPSAT (Wolfman and Weld 1999), take into account numerical resources but are restricted to linear inequality constraints. CP-based planners, such as CPLAN (van Beek and Chen 1999) and the EXCALIBUR agent's planning system (Nareyek 2001), can theoretically handle more general constraints. See Nareyek et al. (2005) for a more detailed discussion.

A fully-extensible planning architecture should be able to handle flexible planning problem constructions such as the general search frameworks described above (including future refinements to these frameworks), while retaining the domain-specific expressiveness found in monolithic systems.

PDDL and Planner Extensibility

There are many available planning systems, often using very different internal representations of planning domains. The Planning Domain Definition Language (McDermott et al. 1998) was conceived to enable standardized comparisons and competitions between planning engines. PDDL solves a critical problem by exposing an *extensible language* to introduce new features to a planning system's model—by default, it recognizes STRIPS-style actions, but it also recognizes feature extensions such as conditional effects, hierarchical actions, durative actions and numerical reasoning (Fox and Long 2003), and as of version 3.0, preferences and soft constraints intended for CP planners (Gerevini and Long 2005). The `requirements` tag of PDDL invokes these extensions, which, in turn, change parts of the language's definition.

However, since PDDL is designed as a common language intended for academic planning competitions, it has distinct disadvantages in real-world applications. PDDL was conceived during a time when monolithic planners with STRIPS-like constructions were the norm, and the extensions were added stepwise as new planning paradigms were introduced. Consequently, these extension constructs, including but not limited to the simplified treatment of resource properties, durative actions, nonlinear numerical projections and unknown information, have been subject to criticism (Boddy 2003). Additionally, there is no direct way to expose sensing and actuating interfaces to the outside world, which is a requirement for specifying online planning problems. While a planner can add new or improved constructs in its private implementation of PDDL, this would result in the proliferation of non-standard extensions that are incompatible across planning systems. It may be possible to standardize certain extensions (PDDL versions 2.1 and above are indeed targeted towards providing standard extensions); however, as PDDL's intent is to provide a common interface that is not necessarily efficient nor sufficiently expressive (for example, continuous numerical effects in PDDL are assumed to be linear, making non-linear continuous functions hard to express), strict adherence to the language will impose an artificial restriction on a planning system's capabilities and will limit extensibility.

Furthermore, PDDL's `requirements`–based extensibility is *not a solution to support real-world applications*. As mentioned earlier in the Introduction, real-world applications using a planning system need to extend that planning system according to their special requirements by providing their own custom modules (e.g., new data types, new heuristics, or custom sensors and actuators). Ideally, external users (application developers or even third-party vendors) should be able to add new constructs to the planning problem definition without the domain modeler needing to recompile the planning system or its problem definition parser. This functionality is inherently absent from PDDL as it was intended to be an academic tool, with little consideration for a professional or industrial environment.

This paper proposes a solution to these problems by presenting a general planning system framework based on the Extensible Markup Language (XML), allowing a simple, modular way to extend the planning system and its model. The goal is to create a *pluggable system of planning extensions that neatly tie into the representation language*. Efficiency is not the main focus (although a clean problem representation that directly corresponds to the planning system's internal structure will naturally be more efficient than a poorly-fitting PDDL representation); rather, a planning system implementing our framework will be "future-proof", with a vast potential for new planning extensions to extend the capabilities of planning beyond what is currently being explored in academic circles.

The next section introduces an example scenario where extensions are needed, followed by a discussion of the framework itself, and the extension possibilities it allows.

An Example Scenario for Extensible Planning

The Crackpot planning system will be used throughout this paper as an example to show how our proposed framework can be implemented by a typical planning system. This section contains a brief introduction to Crackpot, along with a sample problem to be tackled by this planner.

The Example Planner

Crackpot[1] is the successor of the EXCALIBUR agent's planning system (Nareyek 2001). As such, it uses the same principle of local search based on iterative repair to make and improve plans—a plan with inconsistencies or *costs* (e.g., unmet goals, mutually-exclusive actions that overlap, unmet preconditions for an existing action, etc.) is iteratively improved by using one of several *repair* heuristics (e.g., add a new action, move an action's start/end times, etc.). Crackpot is intended to be an *online* planner, where agents other than itself might change the state of the world as time passes, and actions can only be added to the plan at positions at or beyond the current time.

Crackpot, as with most planners, separates the notion of a general *domain* from a specific *problem* of the domain, allowing modelers to create separate specifications of each. Crackpot internally models a domain/problem using an object-oriented design amenable to implementation in C++. Figure 1 depicts the relationships between Crackpot's domain specification constructs using a UML (OMG 2010) class diagram.

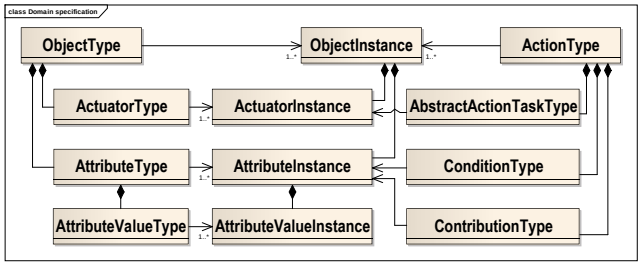

Figure 1. Crackpot's domain specification class structure. (This specification is a work-in-progress.)

In general, a distinction is made between the *type* and *instance* of a particular construct (e.g., ObjectType vs. ObjectInstance): the abstract types (e.g., the "person" type) appear in the domain specification, while the grounded instances (e.g., a "person" named "Joe") appear in the problem specification. (It is the planning system's task to create action instances and their components; hence the related classes are not shown in the above diagram.)

A domain *object* (e.g., a person) contains state resource variables called *attributes* (e.g., walking speed) whose projections over time can consist of one or more *attribute values* (e.g., 0.0 m/s at the start, 1.0 m/s at the end). An object also contains action resource variables called *actuators* (e.g., legs).

An *action* (e.g., walking from one place to another) is made up of *object parameters* (technically, "parameter object instances") specifying which objects are related to the action (e.g., which person is doing the walking), *conditions* on the object parameters' attributes that must be met for the action to execute (e.g., the person must be at the start location), *contributions* of the action to the attributes (e.g., the person ends up in the target location), and *action tasks* indicating how actuators are used throughout an action (e.g., a person uses his legs to walk).

Note that this design roughly maps to the EXCALIBUR planning system's model. In particular, actuators, attributes, and conditions/contributions directly map to action resource constraints, state resource constraints, and task constraints. (Nareyek 1998)

Because of efficiency reasons, Crackpot is *not* a completely-modular system, taking a middle ground between monolithic and fully-modular planning systems. Its design currently assumes a fixed flow of the planner's execution cycle (find a repairable cost in the plan, repair the cost, repeat). Third-party extensibility of the planner itself is currently restricted to introducing specialized attribute value types (e.g., a SymbolicLocation type extended from the provided Symbolic type to allow for path planning, or perhaps a collection-oriented Set type).

However, the next version of Crackpot that is currently being worked on will increase the expressiveness of domains, by introducing a *control flow actuator* (Nareyek 2003) to allow changes in the planning execution cycle (e.g., temporarily focusing on specific plan repairs), a *cost management system* to allow domain-influenced selection of the repair heuristic by specifying modifiers for each available cost (e.g., to cause the planner to prefer adding certain actions over others), *action-component relations* that specify additional constraints between an action's object parameters and other action-related parameters (e.g., to set the duration of the action according to the value of a resource), and *read-ins* that take in attribute values and feed them into action-component relations. These extension possibilities must be taken into account when designing a representation language for Crackpot.

A Sample Problem with Extension Possibilities

Consider a very simple planning problem: A person is currently in his living room, and he is hungry. Given the following scenario, what must he do to satiate his hunger?

- There is an apple in the kitchen. Conveniently, he can travel from one place to another by walking.
- The way to the kitchen is separated by a closed door. The only way to overcome this formidable obstacle is to open it with his hands.

To model the *domain* of this problem, a modeler may use Crackpot's constructs in the following manner. (The

[1] Crackpot is a work-in-progress planner available at the following URL: http://sourceforge.net/projects/crackpot

problem specification is omitted to save space, but the initial state and goals may be inferred from the above description.)

ObjectType: **Person**
AttributeType: **Hungry** { *true, false* }
AttributeType: **Location** { *livingroom, door, kitchen* }
ActuatorType: **Legs**
ActuatorType: **Hands**
ObjectType: **Apple**
AttributeType: **Existing** { *true, false* }
ObjectType: **Door**
AttributeType: **Open** { *true, false* }
ActionType: **EatApple**
Parameters: { p : Person, a : Apple }
Conditions: { p.Hungry = *true*, p.Location = *kitchen*, a.Existing = *true* }
Contributions: { p.Hungry = *false*, a.Existing = *false* }
ActionTask: *uses* p.Hands
ActionType: **WalkFromLivingRoomToDoor**
Parameter: { p : Person }
Condition: { p.Location = *livingroom* }
Contribution: { p.Location = *door* }
ActionTask: *uses* p.Legs
ActionType: **WalkFromDoorToKitchen**
Parameters: { p : Person, d : Door }
Conditions: { p.Location = *door*, d.Open = *true* }
Contribution: { p.Location = *kitchen* }
ActionTask: *uses* p.Legs
ActionType: **OpenDoor**
Parameters: { p : Person, d : Door }
Conditions: { d.Open = *false*, p.Location = *door* }
Contribution: { d.Open = *true* }
ActionTask: *uses* p.Hands

The above representation is adequate for a planning domain with relatively simplistic assumptions. True enough, it is also possible to create a PDDL description out of this domain with predicates and actions (each with parameters, preconditions and effects), all but with a slight loss of fidelity to the modeler's intent; for example, the concept of actuators are lost in the translation. (The PDDL version is not shown here, again due to space constraints.) However, suppose that this is part of a more sophisticated computer game world, where a non-player character (NPC) agent has a relatively simple behavioral AI such as that described above (in order to tell a simple story, for example). Some problems with the above domain representation immediately become clear:

1. Game worlds, more often than not, have a running game clock, so actions don't occur instantaneously but are executed over certain durations.
2. Agent attributes such as hunger (or generally, health) in computer games are, more often than not, modeled as numerical resources that rise and fall over time, not just Boolean values as assumed here.
3. It is inadequate to specify game world locations as plain symbols. Without information about each location's actual Cartesian coordinates and its connectivity with other locations, this representation does not scale well to a real path-planning problem (as the current form requires many "walk" actions to be defined between each connected location).
4. The door's actual state may change irrespective of the agent's interactions—if a player closed the door immediately after our simplistic agent opened it, the agent will suddenly not be able to pass the "real" door in the game (although the agent thinks it has), nor would it know that it needs to re-open the door, unless the planner is notified of the change.

All these problems stem from a lack of expressiveness in the domain. What we need in this case are mechanisms to specify action durations, numerical attributes, symbolic path-planning, and some form of sensing functionality. These features will require extensions in the planning system. Perhaps just as importantly, these extensions must be properly exposed in the corresponding representation language. Note that if we had used a PDDL representation, we will be able to solve the first two representation problems (as PDDL 2.1 and above already support durative actions and numerical attributes), but we cannot solve the last two without extending PDDL's language specification.

Key Guidelines of the Proposed Framework

Having introduced our example planning system and problem, this section now presents the key guidelines of our planning extension/representation framework, explained via examples using the Crackpot system.

Correspondence between Language and Planning System Elements

The concept of an extensible planning language, introduced by PDDL, is quite essential for our proposed framework. However, PDDL is not able to handle nuances unique to a specific planning system, limiting its real-world use. In Crackpot, for example, it is non-trivial (although possible) to map PDDL predicates to ObjectTypes and AttributeTypes; worse, there is no direct PDDL analogue for ActuatorTypes.

This problem can be alleviated by designing the language around the planner, not the other way around. More succinctly, *form follows function*; this idea has been pointed out in critiques of PDDL (Boddy 2003). It must be noted that PDDL's "one-size-fits-all" representation stemmed from the need to provide common language elements across planning systems. Since our main focus is to extend planning systems into real-world applications such as games, with little to no use for inter-planner compatibility, we extend the basic idea into this

philosophy: *Develop a language that closely corresponds to the target planning system's internal representation.*

For example, since Crackpot recognizes ObjectTypes and AttributeTypes as first- and second-class constructs, respectively, they should be represented as-is in the language with their relative hierarchy unchanged (as opposed to representing their relationship as a predicate in PDDL). This has the advantage of easier extensibility system-wise, because new classes of constructs can be introduced to a domain language using the same class hierarchy of the planning system; for example, it is now trivial to add ActuatorTypes to the new language.

Planners conforming to the said philosophy will, of course, not be able to read each other's languages. Also, in the worst case, future planning problems and systems might require restructuring of the ontology: For example, Crackpot improves over EXCALIBUR (Nareyek 2001) by requiring resources to be grouped into objects for more expression possibilities (e.g., attributes can be references to objects), at the expense of incompatibility. However, language translation tools exist, such as proposed by Clark (1999), that allow wide-scale restructuring of a language, removing unnecessary data or even adding missing data, making it possible to import problems between planners.

Distributed Parsing of the Planning Language

A system that may be extended by external modules needs to have some form of *registration system*, which registers the cases when a planner needs to dispatch tasks to an external module rather than its internal constructs; for example, calling the constructor of an externally-created attribute instead of the system's built-in attributes.

The *Observer design pattern* (Gamma et al. 1995) is used as the basis of the registration system. This pattern is developed mainly for distributing events to *observers* or *listeners*, and it works well with our scenario—this allows modules to independently handle their own constructs. The Observer pattern effectively *distributes* the parsing of the representation language to the specific modules that are interested in smaller parts of the language.

For example, a SymbolicLocationAttribute external module can register as a listener on the same parts of the planning system that other attributes (SymbolicAttribute, NumericAttribute, etc.) also listen into. This way, whenever the language parser encounters the use of an attribute, a general "event" is fired, and the registered listener (in this case, SymbolicLocationAttribute) does the actual task, e.g., construction of the attribute, production of value instances, and managing of relations such as equality, comparison, etc. that are valid for the attribute.

XML as a Language Base

Theoretically, this planning framework can use a PDDL-like syntax as the language base. However, a much better option exists, in the form of the Extensible Markup Language or XML (Bray, Paoli, and Sperberg-McQueen 1998). Using XML as the language base has several advantages over maintaining a separate language:

- Since XML is widely considered as a standard, it enjoys vast third-party library support. Extension-aware planners will invariably have languages that change frequently. Existing XML libraries already allow users to change an XML-based language without needing to recompile the parser itself, which is perfect for a rapidly-evolving language.
- The XML SAX API (Megginson 2004) allows exactly the kind of distributed parsing that we need. SAX is a lightweight, event-driven API where each well-formed XML element (or "tag") fires an event; the target system's internal parser looks at an incoming XML "event" and distributes the event to the appropriate listener. The parser only needs to maintain a lookup table to find out which listener should be activated for which XML element.
- Using XSLT (Clark 1999), it is possible to do automatic translation between different languages (as recommended earlier). In fact, it is possible to transform the language into a version of PDDL with XML-style tokens, on which a simple token substitution can be performed to obtain pure PDDL.

XML is only used for planning system features where very high performance is not required. Generally, planning domains and problems are only loaded at the start of the planning process, so performance is not normally an issue, especially when taking into account the benefits that XML provides in terms of flexibility.

Example Implementation: Crackpot SLAP

A new domain specification language, dubbed "Scalable Language for Action Planning" or SLAP, is developed specifically for Crackpot. Two XML document schemas are created: The `<domain>` schema handles the formal definition of a domain (i.e., all *xxx*Type constructs), while the `<problem>` schema specifies a problem instance of that domain (i.e., all *xxx*Instance constructs).

The example domain presented earlier roughly translates to this form in SLAP, which corresponds with how Crackpot models the domain internally (for illustrative purposes only; details are left out due to space constraints):

```
<domain name="Apple Domain">
  <!-- definition for the location type -->
  <attribute_value_type name="LocType"
                        data_type="symbolic">
    <value name="livingroom" />
    <value name="door" />
    <value name="kitchen" />
  </attribute_value_type>

  <!-- object definitions -->
  <object_type name="Person">
    <attribute_type name="Hungry"
                    attribute_value_type="boolean" />
    <attribute_type name="Location"
                    attribute_value_type="LocType" />
    <actuator_type name="Legs" capacity="1" />
    <actuator_type name="Hands" capacity="1" />
  </object_type>
```

```
  <object_type name="Apple">
    <attribute_type name="Existing"
                    attribute_value_type="boolean" />
  </object_type>
  <object_type name="Door">
    <attribute_type name="Open"
                    attribute_value_type="boolean" />
  </object_type>

  <!-- action definitions -->
  <action_type name="EatApple">
    <parameter name="p" object_type="Person" />
    <parameter name="a" object_type="Apple" />
    <condition_type parameter="p"
                    attribute_type="Hungry"
                    relation="equals" value="true" />
    <condition_type parameter="p"
                    attribute_type="Location"
                    relation="equals" value="kitchen" />
    <condition_type parameter="a"
                    attribute_type="Existing"
                    relation="equals" value="true" />
    <contribution_type parameter="p"
                       attribute_type="Hungry"
                       value="false" />
    <contribution_type parameter="a"
                       attribute_type="Existing"
                       value="false" />
    <action_task_type parameter="p"
                      actuator_type="Hands" />
  </action_type>
  <action_type name="WalkFromLivingRoomToDoor">
    <parameter name="p" object_type="Person" />
    <condition_type parameter="p"
                    attribute_type="Location"
                    relation="equals"
                    value="livingroom" />
    <contribution_type parameter="p"
                       attribute_type="Location"
                       value="door" />
    <action_task_type parameter="p"
                      actuator_type="Legs" />
  </action_type>
  <action_type name="WalkFromDoorToKitchen">
    <parameter name="p" object_type="Person" />
    <parameter name="d" object_type="Door" />
    <condition_type parameter="p"
                    attribute_type="Location"
                    relation="equals" value="door" />
    <condition_type parameter="d"
                    attribute_type="Open"
                    relation="equals" value="true" />
    <contribution_type parameter="p"
                       attribute_type="Location"
                       value="kitchen" />
    <action_task_type parameter="p"
                      actuator_type="Legs" />
  </action_type>
  <action_type name="OpenDoor">
    <parameter name="p" object_type="Person" />
    <parameter name="d" object_type="Door" />
    <condition_type parameter="p"
                    attribute_type="Location"
                    relation="equals" value="door" />
    <condition_type parameter="d"
                    attribute_type="Open"
                    relation="equals" value="false" />
    <contribution_type parameter="d"
                       attribute_type="Open"
                       value="true" />
    <action_task_type parameter="p"
                      actuator_type="Hands" />
  </action_type>
</domain>
```

To implement the extensible framework itself, there were minimal changes to Crackpot's class structure. See Figure 2 for an overview. The Xerces-C++ parser (Apache Xerces Project 2010) was used for XML parsing, wrapping the library in the class XMLFactory using the Façade pattern (Gamma et al. 1995).

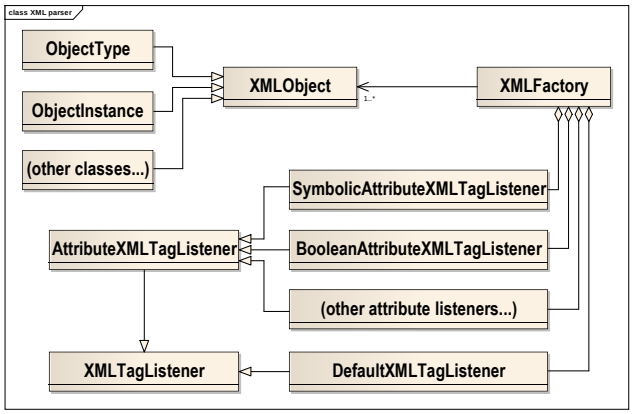

Figure 2. Crackpot's XML parser class structure.

For a custom module to register with the extensible framework, the interface XMLTagListener is provided. Since Crackpot also allows for third-party-supplied attributes, AttributeXMLTagListener is provided to further expand the XMLTagListener interface with helper methods that are relevant for attribute modules.

Registered listeners are stored in a lookup table on the tags that they listen to, e.g., all AttributeXMLTagListeners listen to the `<attribute_value_type>` tag. To resolve simple conflicts between multiple extension modules that listen to the same XML tags, Crackpot utilizes a *last-registered-first-called* rule, where the last listener to register is the first listener to be invoked by the parser. This rule makes sense because custom modules typically register with the system *after* the base modules. Future versions of the system may incorporate more sophisticated conflict resolution (more on this in the next section), but as it stands, the current system already allows for many interesting extension possibilities.

Possibilities for Planner Extensions

The advantages of our framework become apparent once the example problem is extended with new functionality.

Introduction of New Features

In our architecture, it is possible to add a new XML element for each new feature added to the system. For example, in Crackpot, a timing module[2] can be introduced to the system to handle action durations, which registers with our framework by listening to a new XML element, `<timing>`. This new element can be placed as a child under `<action_type>`. This allows many ways of implementing durative actions, such as a fixed duration:

[2] Crackpot currently implements durations in a different way; action-component relations will handle durative actions even more generally.

```
<action_type name="WalkFromLivingRoomToDoor">
  <timing duration="10" />
  ...
```

Or a condition- or contribution-related sub-duration (the overall duration is computed from all sub-durations):

```
<contribution_type parameter="p"
                   attribute_type="Hungry" value="false">
  <timing effect_time="30" />
  ...
```

Using this framework, any planning system can decide how to model durations without being tied to a particular representation like that of PDDL, where durative actions needed a completely new construct `(:durative-action)` to support a single way of specifying durations.

Custom modules to add sensors to the outside world (in order to support online planning) are likewise easy to add in. An application can create hooks to attributes by adding a tag under the `<attribute_type>` tag:

```
<object_type name="Door">
  <attribute_type name="Open"
                  attribute_value_type="boolean">
    <sensor_stream id="doorState" resolution="5" />
  </attribute_type>
</object_type>
```

In this example, the `<sensor_stream>` element is provided by a custom module, and specifies that a refresh of the door state is triggered every 5 time units. The actual sensor values may be transmitted to the planner via low-level means (i.e., not XML, for higher performance).

Overloading of Existing Features

It is also possible to extend the behavior of an XML element via *element overloading*, i.e., letting multiple modules listen-in on the same XML element. For example, custom modules to support new attribute value types like NumericRange and SymbolicLocation can provide listeners to the `<attribute_value_type>` tag, overloading its use when it encounters a `data_type` string that corresponds to what this module handles. They can make their own XML tags further down the hierarchy:

```
<attribute_value_type name="HungerType"
                      data_type="numeric_range">
  <range begin="0" end="100" />
</attribute_value_type>
<attribute_value_type name="LocType"
                      data_type="symbolic_location">
  <value name="livingroom">
    <coordinates x="0.0" y="0.0" />
    <connection to="door" />
  </value>
  <value name="door">
    <coordinates x="0.0" y="10.0" />
    <connection to="kitchen" />
    <connection to="livingroom" />
  </value>
  <value name="kitchen">
    <coordinates x="10.0" y="10.0" />
    <connection to="door" />
  </value>
</attribute_value_type>
```

These modules can then also override the `<condition>` and `<contribution>` tags to specify their own relations and operations:

```
<!-- a more natural model of hunger satiation -->
<condition_type parameter="p"
                attribute_type="Hunger"
                relation="greater_than" value="50">
  <timing check_time="0" />
</condition_type>
<contribution_type parameter="p"
                   attribute_type="Hunger"
                   operation="linear_decr" value="25">
  <timing effect_time="20" duration="30" />
</contribution_type>
```

These XML elements are handled directly by their respective modules, giving these modules the freedom to specify an entirely new XML hierarchy for their own data; for example, custom Set or Matrix attributes may include sizable amounts of formatted numeric data (potentially with the base functionality inherited from NumericRange).

Feature overloading may introduce problems when conflicting modules listen-in on the same XML elements (necessitating *conflict resolution*, mentioned in the previous section), but these issues are not unlike those encountered with OOP languages like C++; in future implementations, these problems may be solved using the same software engineering principles commonly used in these languages (such as disallowing multiple inheritance, adding support for public/private visibility, and so on).

Planner-Specific Exposure of the Solution Process

So far the preceding extensions simply modify existing planning constructs to support better expressiveness of a problem domain. However, *internal planner extensions* can also expose or even introduce changes to the solution process itself. Such extensions are not meant to be written by third-parties but by internal developers of the planning system. For example, Crackpot's forthcoming cost management system, an improved version of what is found in EXCALIBUR (Nareyek 2001), will allow cost modifiers to influence the selection of repair heuristics in a specific domain. These costs are specified in the form of *domain hints*. First, *cost collections* are specified by the domain:

```
<cost_collection name="satisfaction">
  <cost_type name="goal" cost_mapping="3x" />
  <cost_type name="aux" cost_mapping="2x" />
</cost_collection>
<cost_collection name="optimization">
  <cost_type name="optional" cost_mapping="default" />
</cost_collection>
```

Then, cost types may be registered for the different *cost centers* in the domain, i.e., attributes, actuators and (forthcoming) action-component relations:

```
<attribute_type name="Hungry"
                attribute_value_type="boolean">
  <cost_registration cost_name="unsatisfied"
                     cost_type="goal" />
</attribute_type>
<attribute_type name="Location"
                attribute_value_type="LocType">
```

```
  <cost_registration cost_name="distance"
                     cost_type="optional" />
</attribute_type>
<actuator_type name="Legs" capacity="1">
  <cost_registration cost_name="usage_overlap"
                     cost_type="aux" />
</actuator_type>
```

This allows for an expressive model of plan preferences that more closely mirrors Crackpot's internal planning architecture (which is based on cost repair via local search) than that of strong and soft constraints in PDDL 3.0 (Gerevini and Long 2005).

Conclusion

Our proposed framework solves two important problems: how to make a planning system support a level of extensibility to facilitate its use for real-world problems, and how to model and support an evolving domain representation language that allows problems to take advantage of such planner extensibility. The framework uses three key guidelines: *maintaining correspondence* between domain ontology and the planner's internal architecture, *distributing language parsing* to external modules through the use of the Observer design pattern, and *using XML as a language base* to facilitate language design, parsing and translation to other languages.

Possible future work include the development of more sophisticated forms of conflict resolution between planning extension modules, a common XSLT stylesheet library to allow translation of domain problems between planning systems (or to/from PDDL), and extensions to the Crackpot planning system itself, such as the aforementioned cost manager, control flow actuator, action-component relations, and a complete sensing/acting system to fully support real-world online planning.

References

Apache Xerces Project. 2010. Xerces-C++ XML Parser, Project documentation, available at http://xerces.apache.org/xerces-c, Apache Software Foundation.

Blum, A., and Furst, M. 1997. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence* 90(1-2): 281-300.

Boddy, M. 2003. Imperfect Match: PDDL 2.1 and Real Applications. *Journal of Artificial Intelligence Research* 20: 123-137.

Bray, T.; Paoli, J.; and Sperberg-McQueen, C. M. 1998. Extensible Markup Language (XML) 1.0. Technical Report, W3C recommendation, W3C.

Clark, J. 1999. XSL Transformations (XSLT) Version 1.0. Technical Report, W3C recommendation, W3C.

Fikes, R. E., and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 5(2): 189-208.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20: 61-124.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA; Menlo Park, CA: Addison-Wesley Publishing Co.

Gerevini, A., and Long, D. 2005. Plan Constraints and Preferences in PDDL3, Technical Report, R. T. 2005-08-47, Università degli Studi di Brescia, Dipartimento di Elettronica per l'Automazione.

Kautz, H., and Selman, B. 1998. BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving. In *Working Notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98,* 58-60. Pittsburgh, PA.

Kautz, H.; Selman, B.; and Hoffman, J. 2006. SatPlan: Planning as Satisfiability. In *Abstracts of the 5th International Planning Competition,* available at http://www.cs.rochester.edu/~kautz/satplan/index.htm.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL--The Planning Domain Definition Language--Version 1.2, Technical Report, CVC TR-98-003, Yale Center for Computational Vision and Control.

Megginson, D. 2004. SAX - Simple API for XML, Project documentation, available at http://www.saxproject.org/, Megginson Technologies, Ltd.

Nareyek, A. 1998. A Planning Model for Agents in Dynamic and Uncertain Real-Time Environments. In *Proceedings of the Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments at the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98),* 7-14. Menlo Park, California: AAAI Press.

Nareyek, A. 2001. *Constraint-Based Agents: An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds (LNAI 2062).* Springer.

Nareyek, A. 2003. Planning to Plan - Integrating Control Flow. In *Proceedings of the International Workshop on Heuristics (IWH'02),* 79-84.

Nareyek, A.; Fourer, R.; Freuder, E. C.; Giunchiglia, E.; Goldman, R. P.; Kautz, H.; Rintanen, J.; and Tate, A. 2005. Constraints and AI Planning. *IEEE Intelligent Systems* 20(2): 62-72.

OMG. 2010. Unified Modeling Language (UML), Version 2.2, Formal specification, available at http://www.omg.org/spec/UML/2.2/, Object Management Group.

van Beek, P., and Chen, X. 1999. CPlan: A Constraint Programming Approach to Planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99),* 585-590.

Wolfman, S. A., and Weld, D. S. 1999. The LPSAT system and its Application to Resource Planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence,* 310-316. Stockholm, Sweden.