An Extensible Action Architecture for Planning in Complex Domains

Amit Kumar and Alexander Nareyek
Department of Electrical and Computer Engineering
National University of Singapore
4 Engineering Drive 3, Singapore 117576
amit_kumar@nus.edu.sg, alex@ai-center.com

Abstract
Planning in complex domains, including temporal and resource planning, often requires an expressive and flexible model of actions. For example, in a rover domain, the energy of the rover, a bottleneck resource, is consumed at a high rate while performing a drive action and in proportion to the distance traversed. Based on the current energy levels, the planning system can adapt the distance traversed due to the drive actions in order to best achieve its goals, i.e., if more energy is left, drive actions can be prolonged. To handle such domains, we consider an expressive model of actions containing variables (to denote, e.g., the energy consumed and the distance traversed within a drive action), and relations between the variables (e.g., the energy consumed is proportional to the distance traversed). We call these relations Action Component Relations (ACRs). ACRs may also be relations involving structures such as the conditions, effects, actuator tasks, etc., of the action, and other complex values (e.g., an ACR containing a set of temporal relations between the actuator tasks in a job-shop-scheduling domain). For efficient search, ACRs are optionally accompanied with pluggable domain-specific heuristics. In our extensible architecture, a programmer can easily add new kinds of ACRs. Finally, we present the use of ACRs in a story-planning domain for computer games. While our action specification model is generally applicable, our planning architecture is specific to utilize local-search-based planning.

Introduction
We consider complex-world planning, including temporal and resource planning. The typical model of actions, such as in PDDL, consists of conditions and effects. In PDDL (Fox and Long 2003; Hoffmann and Edelkamp 2005), conditions and effects are at the “start”, “end” or “over all” of the action. Many applications require a more expressive model of actions (e.g., see (Boddy 2003; E. Smith 2003; Frank, Golden, and Jonsson 2003)). In job-shop scheduling, for example, there may be multiple conditions and effects within a single complex action, along with temporal relations between them. For a non-temporal example, in a rover domain, the energy of the rover, an insufficient resource, is consumed at a high rate to perform a drive action. Consider the relation: “The energy consumed in a drive action is in proportion to the distance traversed.” Based on the current energy levels, the planning system can adapt the distance traversed in the future drive actions in order to best achieve its goals.

In this paper, we extend the expressiveness and flexibility of the typical model of actions to include variables, and relations, such as the above, between the various components of the action such as the variables, preconditions, effects, actuator tasks, etc. We call these relations as Action Component Relations (ACRs). An action may have zero or more ACRs. ACRs are useful not only to express more complex action models such as in job-shop scheduling above, but especially useful in dynamic re-planning to adapt existing actions within the plan. Some ACRs are required to be satisfied for the action to be feasible, while others do not affect the feasibility of the action and are relevant to plan optimization.

In previous works (Nareyek 2001a), an extensible planner architecture was constructed in order to be able to add new kinds of values (such as numerical, sets, 3D location, etc) for attributes of objects, enabling reuse of the common components of the planner. In this complementary work, we construct an extensible planner architecture into which new kinds of ACRs can be added into the planner.

There have been multiple other approaches to extending the model of actions in planning domains, such as in Zeno (Penberthy and Weld 1994), IxTet (Laborie and Ghallab 1995), CNT (Verfaillie, Pralet, and Lemaître 2010b) and CAIP (Frank and Jónsson 2003) frameworks. In terms of expressiveness of planning domains, ACRs are as powerful as general constraints, which are supported by CNT and CAIP (IxTet and Zeno support a restricted family of constraints). However, our approach uses higher-level constructs such as composite value types (e.g., 3D location), conditions, contributions, etc., and ACRs between such higher-level constructs. While general constraint techniques can be used to process ACRs during planning, the usage of
high-level constructs allow encoding of domain-specific efficient solution heuristics for ACRs, e.g., heuristics that cater to specific constrained resources. High-level constructs also allow modeling of problems in a concise and natural way. For a discussion on high-level versus low-level constructs in planning, see (Nareyek 2001b). Therefore, for purposes of efficiency, in our architecture, the pluggable ACRs are optionally accompanied with custom solution heuristics.

Local-search methods have been effectively used for solving complex planning problems, including dynamic changes and anytime computations (e.g., see (Rabideau et al. 1999; Nareyek 2001a)). While our action domain model is generally applicable, our solution architecture is specific to utilize local-search-based planning.

In the paper, we also present the use of ACRs in a story-planning domain for computer games.

Running Example Throughout the paper, we consider an extension of the logistics domain (Bacchus 2001). The duration of the execution of a Drive(truck: Truck, src: Depot, dest: Depot) action is a function of the src and dest parameters of the action. To reduce the duration of a Drive action during planning, for example, it is possible to replace the current destination by a closer destination.

Preliminaries

To be able to describe the structure of actions and relations between them, we first present a brief description of the model of the world recognized by the planner. This world model is based on the Excalibur architecture (Nareyek 2001a) with name changes and the new concept of Read-In.

Basics of the World Model

An object-based notation provides an intuitive way to model the world. An object type describes a family of objects of a certain kind, such as the Truck type. An object instance is an instance of a particular object type, such as truck1. Each object type (object instance) has a set of attribute types (attribute instances, respectively) each of which describes a particular property of the object. For example, Truck.location is an attribute type and truck1.location is an attribute instance. An attribute instance may store the projected value resultant from the actions in the plan. For example, in Figure 1, the projected value of truck1.location at 5pm is depot2. See Figure 2 for relationships between the various concepts.

Throughout the paper, we use terms like “XXX type” and “XXX instance”. Their definitions belong to the domain model and problem model respectively as commonly used in PDDL. While “XXX type”’s are used to represent various sorts of structural invariants, “XXX instance”’s are used to represent an instance that follows the invariant specified by the type.

Figure 1: Projection of truck1.location and other attributes, and truck1.engine actuator for a Drive from depot1 to depot2. The Drive action has a condition that the truck can drive only within a city. In the plan, this condition is checked at 2pm. The action has an actuator task on truck1.engine actuator between 2-4pm and a contribution on truck1.location at 4pm onwards.

Considering Resources An actuator is a device used to execute a task within an action. A simple actuator can execute just one actuator task at a time and is busy during this execution. An object type (object instance) has a set of actuator types (actuator instances, respectively) each of which represents an actuator and is projected (see Figures 1 and 2).

Figure 2: UML Class Diagram of Concepts Related to the Planning Model

The Action Model

For purposes of clarity, we first provide an informal description of some constituents of an action. Each action type (action instance) has a reference time variable (value, respectively). This reference time is used
in the specification of the constituents of the action. A notation \( t^+ \) is used to denote the time \( t \text{+} \text{reftime} \) where \text{reftime} \ is the reference time of the action.

A contribution replaces the PDDL notion of an effect of an action. A contribution of an action is a change affected by the action to an attribute within a certain interval of time. For example, an action to pour water into a bucket increases the water level at rate } +3 \text{ during the interval the pour is carried out. Two concurrent pour-water actions result in an increased rate } +6 \text{ of water filling. Thus, the notion of contribution allows our model to accommodate concurrent and synergistic changes to an attribute produced by multiple actions. Sometimes a contribution may not be applicable at the current projected value of an attribute. For example, a numerical attribute may have a restricted range between 0 and 100, and therefore a contribution of } +4 \text{ cannot be added to the current projected value of 98. A valid plan should not have any such impossible state transitions.}

Our notion of condition, in a sense generalizes, but also differs, from the PDDL 2.2 (Hoffmann and Edelkamp 2005) notion of a condition of an action. While a PDDL-condition is at the “start”, “end” or “overall” of an action, our condition may last over any domain-specified time duration relative to the reference time of the action. A difference from PDDL-condition is that while PDDL-condition may be a test involving multiple variables, our condition is a test on a single attribute that needs to hold at a particular time interval for the action to be feasible. For example, a Drive action may have a condition truck.location \( ?= \text{src} \) on the truck.location attribute of truck just before the execution of the action starts. A relation between multiple attributes is an ACR, which we describe later. ¹

Reading Values from Attributes Actions often need to use values from the plan, e.g., a Drive action needs to test that the destination is in the same city as the value of the location of the truck at the start time of the action. Using a read-in, an action can receive a projected value from an attribute instance at a particular time point in the plan relative to the reference time of the action, and put that value to an assigned variable. Thus, in a well-formed plan, the value of the assigned variable is equal to the read-in projected value.

In the Drive action, instead of a PDDL-condition truck.location \( \approx \text{src} \), the value of src can be received by a read-in at the start time of the action. That is, in the model, replace the parameter src by a new internal variable src and the PDDL-condition by a new read-in: Depot src = readIn(truck.location, 0+). The number of free parameters of an action, and therefore the search space to find a feasible action, is reduced by the use of read-ins. A read-in is not a sensory input but is received from within the plan. Read-ins are similar to unary functions in Functional Strips (Geffner 2001).

Goals Goals often can be specified as tests on attribute instances. For example, a goal may be pkg1.location = depot1 at time 6pm on the attribute location of a package pkg1. Other kinds of goals may be added utilizing our extensible architecture.

\[
\text{Drive}(\text{truck}: \text{Truck}, \text{dest}: \text{Depot}) \{ \\
\text{Variable} \quad \text{dur}: \text{Integer}, \\
\text{Variable} \quad \text{src}: \text{Depot}, \\
\text{Variable} \quad \text{srcCity}: \text{City}, \\
\text{ReadInType} \quad \{ \text{src} = \text{readIn(truck.location at time} \ 0+) \}; \\
\text{ReadInType} \quad \{ \text{srcCity} = \text{readIn(src.inCity at time} \ 0+) \}; \\
\text{ConditionType} \quad \{ \text{dest.inCity} \approx \text{srcCity during} \ [0+,1+]) \}; \\
\text{ACRTyp}e \quad \text{src} \{ \\
\quad \text{// ACR has a reference to an} \\
\quad \text{// external dynamicTable.} \\
\quad \text{dur} = \text{dynamicTable(src, dest, reftime)}; \\
\} \\
\text{ContributionType} \quad \text{ctrl} \{ \\
\quad \text{truck.location} \quad \text{dest} \quad \text{during}\ [\text{dur}+(\text{dur+1})]; \\
\} \\
\text{ActuatorTaskType} \quad \text{actrl} \{ \\
\quad \text{drive-task on} \quad \text{truck.engine} \quad \text{during}\ [0+,\text{dur+}]; \\
\} \\
\}
\]

Figure 3: Definition of a Drive action type. += denotes a contribution that sets a new value into truck.location. The underlined parts are specified in the domain definition (of the action type) input to the planner, while the overlaid parts are planner extensions that are implemented in the source code.

Action Component Relation (ACR)

In this section, we first introduce ACRs and provide an example of an action utilizing ACRs and other action constituents described above. We describe the kinds of extensibility supported by our architecture. Finally, we provided a detailed description of actions.

By the term action component, we mean a condition, a contribution, an actuator task or a read-in. An action component relation is a relation between the action components and variables within an action, and possibly has external attachments. We call these as ACR constituents. An example of an ACR is provided below.

An Example of an Action Type The pseudocode of the definition of an extended Drive action type is in Figure 3. The duration of the trip may depend on the traffic conditions at the time of day. The action refers to a dynamically-updated external table that returns an estimate of how long a trip from src to dest would take if the trip starts at reftime. We assume

¹In Excalibur, Contribution is referred to as State Task, Attribute is referred to as State Resource, Actuator Task is referred to as Action Task and Actuator is referred to as Action Resource.
that the dynamically-updated table is read into the ACR every half an hour. We use a half-closed-half-open time-interval representation (see section 11.3 of (Fox and Long 2003) for a discussion) that includes the start time point but excludes the end time point. An interval \([t, t+1]\) represents an instant. \(\text{dur}, \text{src}, \text{srcCity}\) and \(\text{speed}\) are internal variables of the action, whereas \(\text{truck}\) and \(\text{dest}\) are parameters used to instantiate the action. \(\text{riel}\) reads-in the location of truck at the start of the action. \(\text{riel}\) reads-in the city of \(\text{src}\) at the start of the action.  

In our planning architecture, unsatisfied ACRs, such as due to changes in the dynamic table, result in costs as due to changes in the dynamic table, result in costs. The fix is carried out by changing the value of some of the variables within the ACR constituents, as described in the section on The Planning Process. In general, some ACRs are required to be satisfied for the action to be feasible, while others do not affect the feasibility of the action and are relevant to optimization goals. Both kinds of ACRs are handled by the planning system via costs.

**Complex Constituents for ACRs** Since the ACR constituents are not just primitive variables, but complex structures like a condition, each of the constituents specifies a custom interface to access its values. For example, a ConditionType interface may define the start and duration variables for the use of ACRs as shown in Figure 4. A domain specification and planner-extension specification framework was proposed in (Vidal and Nareyek 2010). We utilize this framework for the specification of interfaces of ACR constituents, and the input specification of ACRs.

**Filters** In general, an ACR constituent can be more complex. For example, in the 8-puzzle domain, the attribute value may consist of a 3x3-matrix. To obtain a part of the value, such as the tile at \((1,0)\), a filter can be utilized. The filter specifies the input required to obtain the part of the value. In this case, the filter is \(\{\text{tileAt}\}, (1,0)\) and returns an integer denoting the value of that tile.

### A Detailed Description of Action

This section defines an action type and an action instance. A few simplifications have been put in place for ease of presentation, but the essential aspects are covered by these definitions.

**Action Type** An action type is a tuple \((D, C, E, I, A, R)\), where:

- Let \(n = n_{\text{param}} + n_{\text{intr}}\) the sum of the number of parameters and the number of internal variables of the action. \(D\) is a \(n\)-tuple \((D_0, ..., D_{n-1})\) denoting the range \(D_{i-1}\) of the \(i\)th variable. \(Reference\ time\) is a predefined parameter with the range of natural numbers.
- \(C\) is a tuple \((C_0, ..., C_{\#C_1})\) of condition types \((O, A, I, P)\). \(O\) is a \(D_j\) for some \(j\), such that \(D_j\) is the range of all objects of an object type and \(A\) is an attribute type.
  - Let \(Range(A)\) denote the range of the projected values of \(A\). For a numerical attribute, e.g., this may be the set of natural numbers. \(P\) is a lambda function \(\lambda x. P(x)\) such that \(P(x) : Range(A) \rightarrow bool\) is a test function and denotes whether the condition is satisfied at the projected value of \(A\). The test \(?\) is described above as an example of \(P\). The valid range of \(x\) is user-defined and we denote it as \(Dom(P)\).
  - A terminal is a variable or a fixed value. \(I\) is a pair of terminals \(\text{term}_{\text{start}}, \text{term}_{\text{duration}}\) denoting the start and duration of an interval, e.g., \([10+, v+]\).
- \(E\) is a tuple \((E_0, ..., E_{\#E_1})\) of contribution types \((O, A, I, F)\). \(O, A, I\) are the same as in condition type. \(F\) is a lambda function \(\lambda x. F(x)\). \(F(x)\) describes the contributed value, e.g., \(\text{increaseBy(x)}\) on a numerical attribute. The range of \(x\) is user-defined and we denote it as \(Dom(F)\). Every attribute type \(A\) defines a contribution merging function \(M\) that maps, for every time point, all the contributions \(F(0)(x_0), ..., F(k)(x_k)\) available at that time point or before to a new value, i.e., \((F(0)(x_0), ..., F(k)(x_k)) \mapsto \text{newValue}\).
- \(I\) is a tuple \((I_0, ..., I_{\#I_1})\) of read-in types \((O, A, T, K)\). \(O, A, T\) are the same as in condition type. \(T\) is a time point terminal. \(K \in \{0, ..., n - 1\}\) denotes

```csharp
interface ConditionType { 
    Variable start: Integer;
    Variable duration: Integer;
    // ...
} // Similar interfaces for ContributionType // and ActuatorTaskType ...

acr-type-interface SimpleActionACRTyp e { 
    Variable startT ime: Integer;
    Variable dst: Integer;
    ConditionType cl;
    ContributionType cl;
    ActuatorTaskType cl;
}

c1.start := startTime; c1.start := startTime;
c1.duration := 0; c1.duration := 0;
c1.duration := dur;
}

Figure 4: An example of an ACR type for a commonly-occurring simple action type.

\(^2\)In case there is confusion that \(\text{truck}.location\) changes not “during the action” but only right at the end, we point to the view expressed in (E. Smith 2003) that all modeling is abstraction. It is up to the modeler to select an appropriate working model. In a different model, the changes during the action may be important.
the variable index of the action, which is assigned by the read-in.

- \(A\) is a tuple \((A_0,...,A_{#A-1})\) of actuator task types \((O,A,I)\). \(O,I\) are the same as in condition type. \(A\) is the actuator type involved in this task.
- \(R\) is a tuple \((R_0,...,R_{#R-1})\) of action component relation types \((R,S_0,...,S_{#S-1})\), where \(S_i\) is an ACR constituent. \(R\) is a function that maps \((S_0,...,S_{#S-1})\) to bool.

**Action Instance** An action instance is a tuple \((\text{type},V,C,E,I,A,R)\), where:

- \(\text{type}\) is the action type of this instance.
- \(V\) is a tuple \((v_i,...,v_{n-1})\) of values such that \(v_i \in \text{type}.\mathcal{D}_i\), \(3\)
- \(C\) is a \((\text{type}.#C)\)-tuple of condition instances \((\text{type}.\mathcal{C}_i,\ obj,\ start,\ dur,\ x)\). Here \(\ start,\ dur\) are the time values of the start and duration of the interval of this condition; and \(\ obj\) is an object instance with \(\text{type}.\mathcal{C}_i.\mathcal{O}\) object type. \(x \in \text{Dom}(\text{type}.\mathcal{C}_i.\mathcal{P})\).
- \(E\) is a \((\text{type}.\mathcal{E})\)-tuple of contribution instances \((\text{type}.\mathcal{E}_i,\ obj,\ start,\ dur)\). Here \(\ start,\ dur,\ obj\) are similar to the condition case. \(x \in \text{Dom}(\text{type}.\mathcal{E}_i.\mathcal{F})\).
- \(I\) is a \((\text{type}.\mathcal{T})\)-tuple of read-in instances \((\text{type}.\mathcal{T}_i,\ t,\ obj,\ b)\). Here \(\ t\) is a time point and \(\ obj\) is similar to the condition case. \(b\) denotes whether the value of the assigned variable is same as the projected value of the attribute \(\ obj.A\) at time \(t\).
- \(A\) is a \((\text{type}.\mathcal{A})\)-tuple of actuator instances \((\text{type}.\mathcal{A}_i,\ obj,\ start,\ dur)\). Here \(\ start,\ dur,\ obj\) are similar to the condition case.
- \(R\) is a \((\text{type}.\mathcal{R})\)-tuple of ACR instances \((\text{type}.\mathcal{R}_i,b)\). Here \(b\) is a bool that denotes where the ACR is satisfied or not at the present value of variables in \(V\).

### The Planning Process

In this section, we provide an overview of the process of planning with our expressive actions. A plan consists of action instances. In a basic planning task, the target is to find a plan that satisfies the specified goals, and all the conditions and ACRs of the action instances within the plan. Additionally, none of the actuator instances should have an over-subscribed resource usage and there should not be any impossible state transitions (see, the Action Model above).

#### Iterative Repair

The planning process is based on local search using iterative repair of a plan. Imperfections in the plan, such as unsatisfied goals, conditions and ACRs, and others described above, are stored as costs having a non-zero cost value. In every iteration, a cost is selected to be improved, a suitable plan change is found, and then the plan change is applied into the plan. The separation of finding a suitable plan change and application of the change into the plan allows multiple plan changes to be evaluated parsimoniously before one (or more) is applied into the plan. A plan change may involve adding an action, removing an action, moving an action across time, or changing a variable within an action. A detailed overview of the iterative repair process can be found at (Kumar and Nareyek 2009; Nareyek 2001a); here we focus on the action-related aspects.

In our implementation, (unsatisfied) ACRs have costs within the action instances, read-ins may also have costs (a cost arises in case the read-in projected value is not equal to the value of the read-in assigned variable), while other imperfections result in costs outside of the action instances. The influences of the various action components, action parameters and ACRs are shown in Figure 5.

**Example of an Iteration** Let us consider the Drive action (Figure 3), and suppose a satisfaction goal is to deliver a package to a depot by 8pm, but the goal is not satisfied in the current plan. Suppose the planner heuristics select to change one of the Drive action instances to produce its contribution contr1 earlier by reducing the duration of the action, so that the package can reach the goal depot in time (the planner may produce the contribution earlier in an alternative way, namely, by reducing the runtime; we assume that this alternative is not currently being considered by the heuristics in our stochastic searching process).

We list a brief pseudocode for the
class VariableValue {
  var: Variable; value: Value;
}

ActionInstance::applyChangeVariable(
  s: SelectedValues, newVarValue: VariableValue) {
  this.updateVarChange(newVarValue);
  this.improveCosts(s);
}

ActionInstance::updateVarChange(
  newVarValue: VariableValue) {
  for all ACRs a in act { a.updateCost(newVarValue); }
  for all read-ins r in act { r.updateCost(newVarValue); }
  for all action components c in act {
    c.handleVarChange(newVarValue);
  }
}

ActionInstance::improveCosts(s: SelectedValues){
  while (time for iteration is remaining, and none of the other termination criteria is reached) {
    a = this.selectACROrReadInForCostImprovement(s);
    Set<VariableValue> changes = a.improveCost(s);
    for all Variables newVV in changes:
      this.updateVarChange(newVV);
  }
}

Figure 6: Pseudocode for applyChangeVariable in an action instance.

applyChangeVariable function in Figure 6. This function is used to update the value of a variable of an action instance; in this case, the duration of the action instance. Suppose the current value of the duration is 5hrs. When the change is requested, the target value of the duration is specified, e.g., 4hrs. The request also restricts the values of certain parameters, e.g., the value of truck is not allowed to be changed; this is done using the s: SelectedValues parameter, which contains a map from (some of the) parameters to their restricted ranges or fixed values; truck is fixed to the existing value.

The function applyChangeVariable first updates the cost of ACRs and read-ins, updates the action components using updateVarChange, and then calls improveCosts, which repeatedly does the following (till a termination condition is not met): select an ACR or a read-in whose cost is to be improved next and apply the change suggested by the selected ACR or read-in.

In the pseudocode, each programmed extension of ACR is required to implement the improveCost and other interfaces, which we described below.

4We use R::f(a: A, b: B): C to denote a function f implemented by a “class” R that takes two values (a of class A and b of class B) and returns a value of type C. C is omitted in case a function f does not return any value. By this within a function, we mean an object of the class that implements the function. The notation of class and object is close to C++ and Java programming languages. This notation is useful to describe the interface implemented by ACR.

Figure 7: Pseudocode for applyInstantiateAction an action type.

Requirements on the Interface of ACR The important operations related to an action during planning are instantiation of an action type, and adapting an existing action instance in order to reduce the cost of the plan. The instantiation operation is performed by applyInstantiateAction as described in Figure 7. The function receives a s: SelectedValues parameter, similar to applyChangeVariable.

Each ACR (and read-in) needs to implement an improveCost function that receives a SelectedValues input and returns a set of VariableValues containing variables that need to be changed and their new values. Each ACR (and read-in) also needs to implement an updateCost function used within updateVarChange.

For the selection of an ACR (or a read-in) in selectACROrReadInForCostImprovement, the function can use one of multiple selection heuristics to select an ACR (or a read-in), such as, first select the ACR (or read-in) with the highest cost, or first select the ACR (or read-in) with the least number of unfixed variables, or first select the ACR containing temporal variables whose values are “earliest” in the plan, etc. To be able to implement these heuristics (or a combination thereof), the following functions are needed from each ACR: ACR::getCostValue(): CostValue and ACR::getVariablesInACR(): List<Variable>. The action instance utilizes these ACR functions to implement the selection heuristics.

A proof-of-concept implementation has been made using C++.

Sample Application: Story Planning For Computer Games

In this application, the story of a game is generated by the planner in an online manner with the target to maximize the experience of a human game-player. The player’s motivation is included as an object in the planning model. This motivation object has the following attributes: the current projected value of the motivation: currentValue; the projected susceptibility of the player: susceptibility; and the system’s confidence for...
the player’s susceptibility: \textit{confidence}.

\begin{verbatim}
EyeBlinkByPrincess(m: Motivation, dur: Time) {
  Variable sus: Integer;
  Variable mChange: Integer;
  ReadInType rin
    {sus := readIn(m.susceptibility at time 0+);} 
  Constant blinkMotRate: Integer = 3;
  ACRType acr1
    {mChange = blinkMotRate \times sus;} 
  ContributionType cont1 { 
    add-slope mChange to m.currentValue 
    during [0+, dur];
  } //... }
\end{verbatim}

\textbf{Figure 8}: A Target Motivation Graph in the Story Planning Domain and its Linear Segment Approximation

\textbf{Figure 9}: Example of a Cue Action in the Story Planning Domain. The add-slope continuous-change contribution adds a slope of \textit{m.currentValue} during the time interval of the contribution. The contribution of this action is computed after multiplication with a read-in susceptibility value of the player. Other action types help in increasing the \textit{confidence} value via utilizing an optimization ACR \textit{confidence = 100}. This optimization is assigned a lower priority than reaching the satisfaction goal by placing the costs related to this ACR in a different cost collection from which costs are selected with a lower priority.

\textbf{Discussions and Related Work}

\textbf{Related Work} As discussed in the Introduction, other frameworks have included an expressive model of action. (Frank and Jónsson 2003) introduce a constraint-based representation in the CAIP framework. Actions and states are described by \textit{intervals}. A \textit{compatibility} defines the constraints within an interval and the relations to other intervals such as the next and previous intervals. The higher-level constructs used in our framework differentiate it from CAIP. For example, an attribute can realistically take a single value at any time. In CAIP, this is enforced by mutual exclusion rules adding to the size of the constraint set. In our framework, this is enforced by using a projected value of each attribute. Within the interval-based representation in CAIP, it does not seem easy to represent complex actions involving multiple time intervals as one coherent action.

(Verfaillie, Pralet, and Lemaître 2010b; Verfaillie, Pralet, and Lemaître 2010a) describe a planning framework called Constraint Network on Timelines (CNT) meant for discrete-event systems. In (Verfaillie, Pralet, and Lemaître 2010b), the authors note that higher-level constructs that would be closer to the modeler point of view and may be desirable to be added into their framework. Our current work may be relevant for such higher-level constructs. However, tasks like explaining the reason for planning-failure to a human user, etc., can be well-supported (in a future work) only using higher-level constructs during planning.

\textbf{Relations in Planning} In the context of ACRs, it is interesting to note the other kinds of relations that occur in planning. A static relation between the objects of the world such as in Logistics, \texttt{truck1} is always in \texttt{city1} may exist. Other dynamic relation between the objects of the world, such as \texttt{truck1} is in \texttt{depot2} at 5pm, may exist. Static and dynamic relations between the objects can be captured in our framework by attribute projection containing references to objects. Static \textit{trajectory constraints} (Gerevini and Long 2006) while planning, such as, in BlocksWorld, a fragile block can never have something above it, are represented using goals that last over all times.

The relation between an object and an action may be captured by a condition, a read-in or a contribution. A relation between multiple actions, such as, an actuator task of an action B starts 5 secs after an actuator task of an action A, may exist. Such relations can be captured within a higher-level hierarchical action involving the usage of ACRs.

Physical and metaphysical relations between attributes of objects, such as, if it rains, the roof of the house gets wet, may exist. These can be captured using a \textit{rules} subsystem described in (Nareyek and Sandholm 2003). Rules are external events that are inserted into the plan whenever the conditions of the rule (e.g., “it rains”) happen.

As noted before in the paper, several of the PDDL constructs are handled differently by our planning framework. Further to this, PDDL’s symbolic conditional effects are replaced by a transition table within attributes. For example, a traffic light attribute may receive a “toggle” contribution, which converts the pro-
ject values from red to green, green to yellow, etc. Other kinds of conditional effects (e.g., if an object is inside the briefcase, then move the object) and derived predicates (Hoffmann and Edelkamp 2005) from PDDL are not yet handled.

Conclusion and Future Work
The need for expressive models of actions required for applications has been previously expressed by multiple authors such as (Boddy 2003; E. Smith 2003; Frank, Golden, and Jonsson 2003). In this work, we have proposed an extensible architecture for actions, containing relations between action components. This work complements the previous work (Nareyek 2001a) on extensibility of attributes. To our knowledge, this is the first work towards extensibility of actions using high-level constructs. Our architecture has been implemented into a planning system called Crackpot and is available as an open-source software from http://sourceforge.net/projects/crackpot/.

The enhanced adaptability of action via ACRs immediately raises the question of adapting an action currently under execution. However, the adaptability of an ongoing action depends on the domain. For example, the flow-shop and job-shop scheduling domains have actions that may be pre-emptable, nowait or blocking (Hatzack and Nebel 2001). Also, in this paper, we have restricted to a fixed number of conditions, contributions, actuator tasks, etc. This severely limits the expressiveness in cases like: all objects related to an object have a condition or a contribution; or, an action involves looping over an actuator task till a certain condition becomes true. However, our framework is amenable to be such extended in the future.

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

References
Nareyek, A. 2001a. 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.