Goal Manager Documentation

Luca Brusatin

February 6, 2023

The Goal Manager (also called the Action Manager or simply "Action") manages and executes agents' tasks (goals) in ADE. In addition to simple action execution, the Goal Manager keeps track of the agents' state (using the Belief Component) and verifies that all goals and actions are acceptable and that the conditions for their execution are met. When interfaced with a Planner Component[1], the Goal Manager can plan as well as avoid and resolve conflicting actions.

Contents

1 Using the Goal Manager **3**
- 1.1 Building, testing and launching the Goal Manager . . . 3
- 1.2 Arguments . . . 3
- 1.3 Interfacing with the Goal Manager . . . 3
- 1.4 Goal Manager GUI/Editor . . . 3

2 Understanding the Goal Manager **5**
- 2.1 Goals . . . 5
  - 2.1.1 Post-conditions . . . 5
  - 2.1.2 Submitting goals by action signature . . . 5
  - 2.1.3 Goal submission . . . 5
  - 2.1.4 Goal status . . . 5
- 2.2 Actions . . . 6
  - 2.2.1 Action status . . . 6
  - 2.2.2 Action primitives . . . 7
  - 2.2.3 Annotations for action primitives . . . 7
  - 2.2.4 Action scripts . . . 9
  - 2.2.5 Action and Operator Specification . . . 11
- 2.3 Operators . . . 12
- 2.4 Control elements . . . 14
  - 2.4.1 If, Elseif, Else . . . 14
  - 2.4.2 For . . . 14
  - 2.4.3 For Each . . . 15
  - 2.4.4 While, do . . . 15
  - 2.4.5 And, Or . . . 15
  - 2.4.6 Not . . . 15
  - 2.4.7 Try, Catch, Finally . . . 16
  - 2.4.8 Exit . . . 16
  - 2.4.9 Return . . . 16
  - 2.4.10 Async and Join . . . 17

[1]TODO: Currently the (RPI) planner is integrated within Action

2.4.11 True . . . 18
2.5 Imports and Name Aliasing . . . 19
2.6 Beliefs, Conditions and Effects . . . 19
2.6.1 Conditions . . . 19
2.6.2 Effects . . . 19
2.6.3 Binding of predicates in conditions and effects . . . 20
2.7 Handling multiple agents . . . 20
2.7.1 Agent-specific action primitives . . . 21
2.8 Observations and Observer actions . . . 21

3 Internal workings of the Goal Manager 21
3.1 Goal Management . . . 21
3.1.1 GoalManager(Impl) . . . 22
3.1.2 BasicGoalManager . . . 22
3.2 Action Execution . . . 23
3.2.1 ActionInterpeter . . . 23

1 Using the Goal Manager

1.1 Building, testing and launching the Goal Manager

The Goal Manager component can be built with the `action`. The tests action be built with the `action-test` target.

1.2 Arguments

The Goal Manager requires/supports the following arguments:

`-agentname [name]` (optional, default: self)
: Specifies the name of the agent for which the Goal Manager is responsible. The Goal Manager will default to this name if no specific agent name is provided during the execution of an action (`?actor` variable). Sometimes referred to as "super agent name" in multi-agent settings.

`-badaction [name]` (optional, repeatable)
: Forbids the Goal Manager from using this action.

`-badstate [predicate]` (optional, repeatable)
: Forbids the Goal Manager from ending up in a state described by the predicate.

`-belief` (optional)
: Register for Belief Component notifications.

`-beliefargs [args]` (optional)
: Arguments to provide to the Belief Component instances spawned by the Goal Manager.

`-beliefcomponent` (optional)
: Specifies the name of the Belief Component to be used by the Goal Manager for the first ("main"/"real") state machine. If this argument is used, the associated Belief Component has to be manually started.

`-dbfile [path]` (optional, repeatable)
: Loads the contests of the file into the Action Database.

`-selector [class]` (optional, default: `com.action.selector.UtilitarianActionSelector`)
: Uses the specified Action Selector mechanism to choose the best action to execute.

`-editor` (optional)
: Shows the Goal Manager GUI. Shows the contents of the Action Database and provides a script editor.

`-goal [predicate]` (optional, repeatable)
: Executes the specified goal as an initial goal.

1.3 Interfacing with the Goal Manager

The Goal Manager interface provides a set of methods that allow other ADE components to submit goals and query for the currently running goals and their status. See `com/action/GoalManager.java`

1.4 Goal Manager GUI/Editor

The GoalManager GUI (figure 1) has functionality for exporting existing entries to new files, deleting entries, adding, duplicating, editing, and saving ActionDBEntries.

In order to run the GUI, run the and the Goal Manager:

```
ant launch -Dmain=com.action.GoalManagerImpl -Dargs="-editor"
```

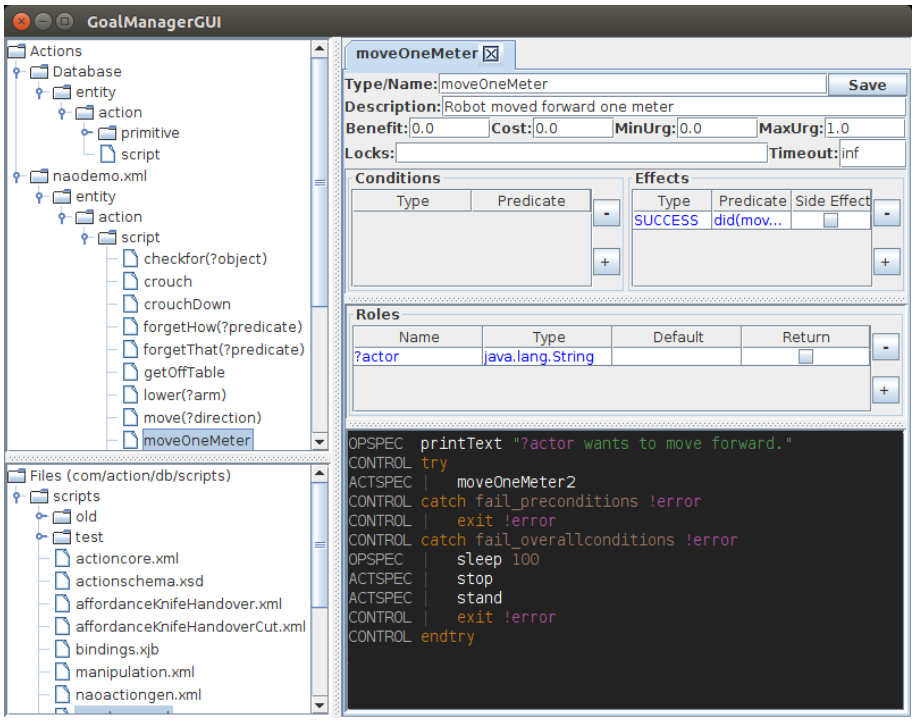


Figure 1: Goal Manager GUI/Editor. The editor has access to the Action Database and shows the available actions in the top-left tree view. The bottom-left tree view shows the available actions scripts. Double-clicking a file loads it into the editor and the file appears in the top-left tree view, where nodes can be expanded. Double-clicking on an action opens the script editor (right). Context menus (right-click) provide additional functionality, such as loading a script file into the Action Database, or removing an action from the Database.

2 Understanding the Goal Manager

2.1 Goals

Goals are described by DIARC first order logic (FOL) predicates. They represent a desired state that the Goal Manager has to achieve using the actions available to it. Goals are sometimes called "post-conditions". If multiple actions can achieve the same goal, the Goal Manager will choose one using the Action Selector mechanism.

A central principle in Goal Manager is the execution of goals, not "actions". By this, we mean that it is the end effect that we're interested in, not the specific implementation of the action. Instead of telling the robot to `moveTo X`, we'd rather have the robot be `at(X)`. Let's expand our `moveTo` example a little bit to explain things. Say the robot has a `moveTo` action that requires the robot to be at a specific starting location for it to work. Now let's say the robot also has another `moveTo` action that does not require a specific starting location, but would for instance be slower to execute. Both actions result in `at(X)`. By submitting a task through a goal specification, e.g. `at(home)`, we give the Goal Manager the freedom to choose the most appropriate action to fulfill this goal. It would, in this case, look up actions that result in at(`X`) and, for each action, check if it is permissible. If the starting location condition of `moveTo` is met, the Goal Manager would have the choice between both `moveTo` implementation and – if the Action Selector is setup appropriately – choose the faster one. If now the starting location condition is not met, the Goal Manager would choose the slower but more generic `moveTo` action.

2.1.1 Post-conditions

A post-condition is a predicate that represents an effect of an action. The term "post-condition" is used because we would like to check that the execution of an action actually resulted in the **intended** effect; it is a condition necessary for the success of an action. Currently, post-conditions are only checked if they are observable (i.e. an observer exists for the predicate). Otherwise the effects are just assumed to hold.

2.1.2 Submitting goals by action signature

As explained prevously, actions are executed when a corresponding goal is submitted to the Goal Manager. Sometimes we would like to be able to "call" an action by name, generally because we do not have a post-condition. This is the case for instance when a goal is submitted through speech input (Dialogue), where the user input might be "walk forward". Before the use of generic post-conditions, we would translate "walk forward" to "walking forward" or "walked forward" which would be the post-condition of an action. This required manual coding of the present progressive or past tense for each action submitted through Dialogue. To avoid this, we allow goals to be action signatures as well as post-conditions.

This used to be handled by auto-generating generic post-conditions for every action, but this is no longer how the Goal Manage handles this case. The Goal Manager now automatically detects if a goal is a post-condition or an action signature.

2.1.3 Goal submission

As explained above, goals – described by predicates – are submitted to the Goal Manager for execution. Figure 2 describes the goal submission process. Note that the execution of actions associated with a Goal happens in a separate thread (ActionInterpreter). Goals can therefore be executed simultaneously, in parallel.

2.1.4 Goal status

Using the getGoalStatus() method, DIARC Components can query the state of the execution of a goal. The following states are defined:

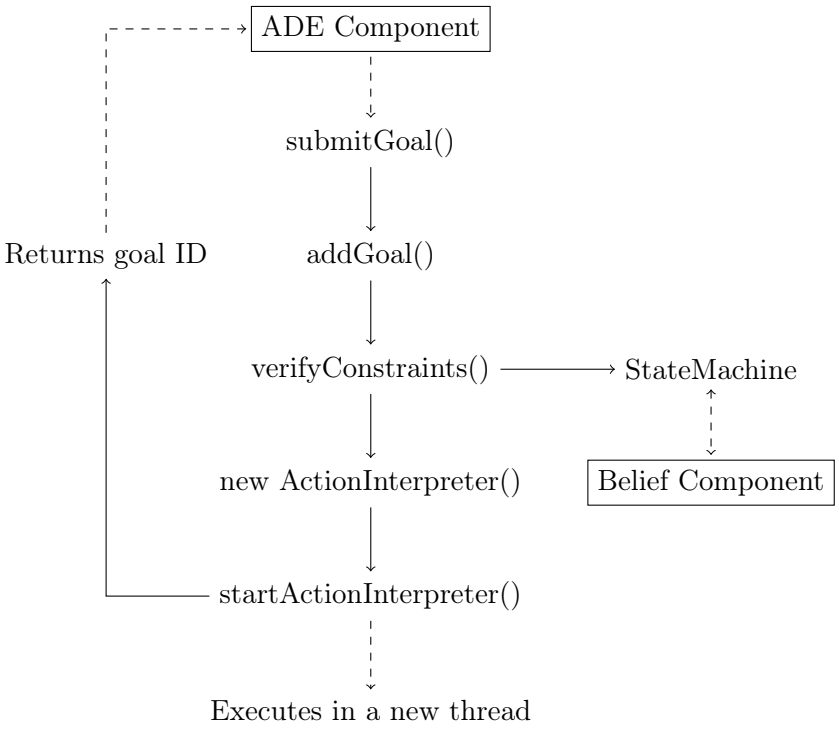


Figure 2: Goal sumission process.

`UNKNOWN` when a goal is not found

`PENDING` submitted but not yet being executed

`ACTIVE` execution of goal is in progress

`SUSPENDED` execution of goal has been suspended

`CANCELED` execution of goal has been cancelled

`SUCCEEDED` goal executed successfully

`FAILED` execution failed

2.2 Actions

Actions can be defined in two different ways, as "primitives" or "scripts". An action primitive is nothing more than a Java method annotated with the `@Action` (and `@TRADEService`) annotation (directly in the java interface). An action script is an ASL (action script language) file describing a sequence of instructions which can be actions (primitives or scripts), operators (e.g +, -, ...) or control-flow elements (`if`, `while`, ...).

2.2.1 Action status

Actions are executed as steps towards achieving a desired goal state. Using the getActionStatus() method, DIARC Components can query the state of the execution of an action (which is being executed as part of some goal). The following states are defined:

`UNKNOWN` when an action is not found

`INITIALIZED` execution of the action just started

`APPROVED` execution of action is approved

`PROGRESS` execution of action is in progress

`VERIFYING_EFFECTS` observable effects are being verified

`SUCCESS` action executed successfully

`FAIL` execution failed

`FAIL_FORBIDDEN` action is not permitted or no permissible action found

`FAIL_PRECONDITIONS` pre conditions for action do not hold

`FAIL_OVERALLCONDITIONS` overall conditions for action do not hold

`FAIL_POSTCONDITIONS` post conditions for action do not hold (observation of effects failed)

`FAIL_NOTFOUND` could not find action

`FAIL_ANCESTOR` an ancestor action failed

`FAIL_CHILD` a child action failed

`FAIL_SYNTAX` action failed because of syntax error in script

`CANCEL` action is cancelled

`SUSPEND` action execution is suspended

2.2.2 Action primitives

Action primitives reside in Java classes (usually DIARC Components) and are available to the Goal Manager when annotated with the `@Action` annotation. Action primitives are only available to the Goal Manager if the Java class that provides them are instantiated and available in TRADE. Actions are dynamically added or removed whenever components/classes come up or go down.

2.2.3 Annotations for action primitives

Java annotations are used to identify action primitves and to list conditions, effects and observations. The following annotations are available:

@Action Annotation used to identify which ADE Component methods are available to the Goal Manager. No arguments.

@Condition Annotation used to specify a condition for an action. Can be repeated.

We use the conjunctive normal form (CNF) to specify. Each `@Condition` annotation has to hold true for the action to be executed. Within the `@Condition` annotation, at least one predicate has to hold true (disjunction). To put it simply, we're doing ANDs of ORs: (C1 and (C2.1 or C2.2) and C3).

By default, each predicate part of the condition will tentatively be observed first, and, if not found, inferred in the StateMachine/Belief Component. It is possible to force the observation of a predicate by adding it to the `observable` array. In this case, the predicate will only be observed. If it can't be found, the StateMachine/Belief will not be checked. It is also possible to force a predicate to be only checked in Belief by adding it to the `inferable` array. Examples:

```
@Action
@Condition(condition={"on(table,?object)"}, type = ConditionType.PRE)
public void pickUp(String object);
```

Annotation 1: Standard condition. The free variable (`?object`) will be bound to the passed-in value with the corresponding name (`String object`)

```
@Action
@Condition(condition={"on(table,?object)", "on(shelf,?object)"},
           type = ConditionType.PRE)
public void pickUp(String object);
```

Annotation 2: Condition with two predicates. At least one of them has to hold true (disjunction).

```
@Action
@Condition(condition={"on(table,?object)", "on(shelf,?object)"},
           type = ConditionType.PRE,
           observable = {"on(table,?object)"})
public void pickUp(String object);
```

Annotation 3: Same as above but `on(table,?object)` has to be observed, e.g. through Vision and will not be checked in Belief.

```
@Action
@Condition(condition={"on(table,?object)", "on(shelf,?object)"},
           type = ConditionType.PRE,
           inferable = {"on(table,?object)"})
public void pickUp(String object);
```

Annotation 4: This time `on(table,?object)` has to be inferred (in StateMachine/Belief) and will not be observed.

@Effect Annotation used to specify an effect of an action. Can be repeated.

The predicates in the `effect` array will be added to the StateMachine/Belief. A predicate can be specified as observable, by adding it to the `observable` array. In this case, it will first be observed that the effect does indeed apply by calling an observer, e.g. Vision. If the predicate cannot be observed, it will not be added to the StateMachine/Belief and the corresponding action will fail.

A predicate can be specified as a side-effect by adding it to the `sideEffect` array. In this case, the predicate will not be a part of the post condition of an action and will not be available to call the action.

An effect can retract a predicate by placing it within a `not()` operator, e.g. `not(on(table,mug))` will retract `on(table,mug)`. Examples:

```
@Action
@Effect(effects={"at(?object,?locB)", "not(at(?object,?locA))"},
        type = EffectType.SUCCESS)
public void moveTo(String object, String locA, String locB);
```

Annotation 5: Standard effect. The free variables (`?object`, `locA`, `?locB`) will be bound to their respective values before assertion/retraction.

```
@Action
@Effect(effects={"at(?object,?locB)", "not(at(?object,?locA))"},
        type = EffectType.SUCCESS,
        observable={"at(?object,?locB)"}
public void moveTo(String object, String locA, String locB);
```

Annotation 6: Same as above but `at(?object,?locB)` has to be observed before assertion, e.g. through Vision.

@Observes Annotation used to specify an observation that an action can make. Cannot be repeated.

Should only be used with java methods of the following form:

```
List<Map<Variable,Symbol» method(Term term);
```

i.e., the method has to take a single Term (or Predicate) as an argument and return a list of maps from Variable to Symbol. Methods that do not respect this will not be available as observers in action and a warning will be shown.

```
@Action
@Observes({"on(table,?object)", "on(shelf,?object)"})
public List<Map<Variable,Symbol» lookforObject(Term object);
```

Annotation 7: Annotation for an action that will observe if `?object` is on the table or on the shelf.

2.2.4 Action scripts

Action scripts are programs, written in a custom ASL (action script language), that are interpreted by the Goal Manager. Scripts are currently stored in the `com/action/db/scripts` folder.

The following is an example ASL script showing the basic language features available:

```
import com.fol.Symbol;

(java.lang.String var2 = "def") = actionName["description"](Symbol ?var1) {
    benefit = 4.0;
    cost = 3.0;
    minurg = 0.6;
    maxurg = 0.7;
    timeout = 0;

    conditions : {
      pre : precond();
      or : {
        pre false : notObservable();
        pre true : observable();
        pre : default();
      }
      overall : overall();
    }
    effects : {
      always : alwayseffect();
      always : alwayssideeffect();
      success : successeffect();
      success true : successobservable();
      failure : failureeffect();
      nonperf : nonperformance();
    }

    locks : motionLock;

    if (op:gt(1, 0)) {
        ?actor.act:action(arg1, arg2, 3.1415);
    }
}
```

`import` import java class. See 2.5 (optional)

`actionName` name of the action (required)

`description` description (optional)

`roles (arguments)` (optional), with the following properties

- `name` has to start with "?" (input or return variable) or "!" (local variable)
- `type` has to be a java type, e.g. `java.lang.String`, `int`, `double`, ...
- `default value` default value if not provided when calling action (optional)
- `return` return variable (optional), there can be multiple return variables

`benefit` benefit of action (optional)

`cost` cost of action (optional)

`minurg` min urgency of action (optional)

`maxurg` max urgency of action (optional)

`timeout` timeout (optional) [currently not implemented]

`conditions` set of predicates (optional) that must hold for the action to execute

`pre` pre-condition (optional), checked right before execution of the action

`overall` overall-condition (optional), continuously checked during the execution of the action

`or` disjunction operator [2] , has to contain at least two `pre` or `overall` conditions. Different types of conditions cannot be mixed.

optional arguments for `pre` or `overall`:

`observable` boolean, specifies whether a condition should be observed (only) or inferred (only). By default a condition is observed if possible, otherwise inferred.

`effects` set of predicates (optional) that represent the result of an action, will be asserted into the action State Machine/Belief Component

`always` always effect, applies as soon as the action is in progress (optional)

`success` success effect, applies if the action is a success (optional)

`failure` failure effect, applies if the action failed (optional)

`nonperf` non performance effect, applies if the action is not executed (optional) [not implemented]

optional arguments for `always`, `success`, `failure`, and `nonperf`:

`observable` boolean, specifies whether an effect has to be observed before getting asserted. By default effects are verified if possible, otherwise just assumed to hold.

`observes` indicates what predicates this action can observe, can be repeated if different predicates can be observed.

`locks` resource locks (optional) [currently not implemented]

`goal` (previously state:) goal state specification (optional), see section **??**

`obs` observation specification (optional), see section **??**

`act` action specification (optional), see section 2.2.5

`op` operator specification (optional), see section 2.2.5 and 2.3

`control` control-flow specification (optional), see section 2.4

2.2.5 Action and Operator Specification

Action and operator specifications (`act:` and `op:`) are used to invoke actions (or operators) in scripts. The usual syntax is the following:

- for an action,

```
?returnvalue = act:actionName(?argument1, ?argument2);
```

```
(?returnvalue1, ?returnvalue2) = act:actionName(?argument1,
?argument2);
```

[2] Used to specify disjunctions in the conjunctive normal form. By default, all conditions are chained together in a conjunction (C1 and C2 and ...). The `or` element chains conditions together in a disjunction (C3 or C4 or ...). It is thus possible to have conditions such as (C1 and (C2 or C3) and C4).

- for an operator,

  `?returnvalue = op:operatorName(?argument1, ?argument2);`

- `(?returnvalue1, ?returnvalue2) = op:operatorName(?argument1, ?argument2);`

Action names are either script names (actionName) or primitive (method) names. Operator names are the method names or symbols defined in `com.action.operators`. Only actions or operators that are available in the Action Database can be called at run-time, i.e. all necessary action scripts have to be parsed and all necessary ADE components have to be up. If an action is not available at run-time, the execution of the action/goal will fail. A list of operator names is available in section 2.3. Arguments need to be provided in the same order as they are declared in scripts or in a method's parameter list. Arguments can be variables (as shown above) or values in String form (e.g. `3.1415`, `"text"`, `myPredicate(?arg)`). Variables inside strings (`""`) are bound to their value if possible, e.g. `"Hello ?subject"` will be bound to `"Hello Jim"` if a `?subject` variable is present and has a value (`Jim`). Variable binding can be avoided if the `?` or ! is escaped with a \. Return values of actions or operators are available after the arguments being passed in. Note that action scripts can return multiple values.

As explained in section 2.7, it is possible to call an action on an agent in particular if there is more than a single agent handled by the Goal Manager. In that case, prefixing the agent name will indicate that the action has to be executed for that agent:

`agentName.act:actionName(?argument1, ?argument2);`

`?actor.act:actionName(?argument1, ?argument2);`

The agent name can be a string constant or a string variable. The `?actor` variable is reserved for that purpose and contains the name of the agent for the current execution context. See section 2.7 fore more details.

2.3 Operators

The following operators are available in action:

Arithmetic Operators

`+` addition, arguments: java.lang.Number, java.lang.Number, returns double

`-` subtraction, arguments: java.lang.Number, java.lang.Number, returns double

`*` multiplication, arguments: java.lang.Number, java.lang.Number, returns double

`/` division, arguments: java.lang.Number, java.lang.Number, returns double

`%` modulus, arguments: java.lang.Number, java.lang.Number, returns double

`++` increment, arguments: int, returns int -- decrement, arguments: int, returns int

`round` rounding, arguments: java.lang.Number, returns int

Comparison Operators

`==` equal, arguments: java.lang.Object, java.lang.Object, returns boolean

`!=` not equal, arguments: java.lang.Object, java.lang.Object, returns boolean

`gt` greater than (>), arguments: java.lang.Object, java.lang.Object, returns boolean

`ge` greater or equal (>=), arguments: java.lang.Object, java.lang.Object, returns boolean

`lt` lower than (<), arguments: java.lang.Object, java.lang.Object, returns boolean

`le` lower or equal (<=), arguments: java.lang.Object, java.lang.Object, returns boolean

`equals` java's equals, arguments: java.lang.Object, java.lang.Object, returns boolean

`notEquals` java's (not) equals, arguments: java.lang.Object, java.lang.Object, returns boolean

Logical Operators

`!` not, arguments: java.lang.Boolean, returns boolean

`and` and, arguments: java.lang.Boolean, returns boolean

`||` or, arguments: java.lang.Boolean, java.lang.Boolean, returns boolean

Misc Operators

`sleep` sleep, arguments: long delay

`randomInt` random integer, arguments: int lower, int upper, returns int

`randomDouble` random double, arguments: int lower, int upper, returns double

`log` log text using standard log4j2 mechanism, arguments: java.lang.String (logging level), java.lang.String (logging message)

Standard Java Methods Operators

`newObject` instantiate new java object via class constructor.

Arguments: java.lang.String (specifying java class), Object...(optional constructor arguments).

Returns: java.lang.Object

`invokeMethod` invoke method on a java object.

Arguments: java.lang.Object (java object instance), java.lang.String (method-name), Object... (optional method args).

Optionally returns: java.lang.Object

`invokeStaticMethod` invoke a java static method.

Arguments: java.lang.String (specifying java class), java.lang.String (method-name), Object... (optional method args).

Optionally returns: java.lang.Object

Collections operators

Generic

`clear` arguments: java.util.Collection

`isEmpty` arguments: java.util.Collection, returns boolean

`size` arguments: java.util.Collection, returns int size

List

`add` arguments: java.util.List<T> list, T element, returns boolean

`add` arguments: java.util.List<T> list, int index, T element

`contains` arguments: java.util.List list, java.lang.Object element, returns boolean

`get` arguments: java.util.List<T> list, int index, returns T element

`remove` arguments: java.util.List<T> list, int index, returns T element

`remove` arguments: java.util.List<T> list, T element, returns boolean

`set` arguments: java.util.List<T> list, int index, T element, returns T element

Set

`add` arguments: java.util.Set<T> set, T element, returns boolean

`contains` arguments: java.util.Set set, java.lang.Object element, returns boolean

`remove` arguments: java.util.Set<T> set, T element, returns boolean

Map

`containsKey` arguments: java.util.Map map, java.lang.Object key, returns boolean

`containsValue` arguments: java.util.Map map, java.lang.Object value, returns boolean

`entrySet` arguments: java.util.Map<K,V> map, returns java.util.Map.Entry<K,V>

`get` arguments: java.util.Map<K,V> map, K key, returns V value

`keySet` arguments: java.util.Map<K,?> map, returns java.util.Set<K>

`put` arguments: java.util.Map<K,V> map, K key, V value, returns V

`remove` arguments: java.util.Map<K,V> map, K key, returns V value

Operators are called with the op: specification.

2.4 Control elements

The flow of control – the order in which statements or instructions are executed – can be controlled with the following elements. In a script, control elements are similar to those used in Java. The following subsections show examples of each.

2.4.1 If, Elseif, Else

The `if`, `elseif`, and `else` elements execute the corresponding set of statements only if some statement (condition) evaluates to `true` (short: "is true"). A statement is true if the evaluated instruction returns true (e.g. an action or operator returns true). A statement is false if the execution of the instruction fails or it evaluates to false (e.g. an action or operator returns false).

The syntax of the if, elseif, else control element is as follows:

```
if (op:lt(?var, 10)) {
  op:log("info", "?var is lower than 10");
} elseif (op:leq(?var, 100)) {
  op:log("info", "?var is lower or equals to 100");
} else {
  op:log("info", "?var is greater than 100");
}
```

2.4.2 For

The for statement provides a compact way to iterate over a range of values.

```
for (!i=0; !i lt 10; !i ++) {
  op:log("info", "One-by-one: !i");
}

for (!i=0; !i lt 10; !i +=2) {
  op:log("info", "Two-by-two: !i");
}

for (!i=100; !i geq 0; !i /=2) {
  op:log("info", "Half: !i");
}
```

2.4.3 For Each

The for-each statement provides a compact way to iterate over all the items in a collection.

```
!list = op:newArrayList("java.lang.Integer");

for (!i=0; !i lt 10; !i ++) {
  op:add(!list, !i);
}

foreach(!elem : !list) {
  op:log("info", "Element: !elem");
}
```

2.4.4 While, do

Instructions can be executed repeatedly while a condition is true.

```
while (op:lt(?i, 10)) {
  op:log("info", "i=?i");
  ?i = op:++(?i);
}
```

2.4.5 And, Or

The `and` and `or` elements evaluate to true if the conjunction or disjunction of contained the statements evaluate to true. These elements can be used like this:

```
if (op:gt(?var, -10) && op:lt(?var, 10)) {
  op:log("info", "?var is between -10 and 10");
} elseif (op:lt(?var, -10) || op:gt(?var, 10)) {
  op:log("info", "?var is l.t. -10 or g.t. 10");
}
```

2.4.6 Not

The `~` element inverts the result of the evaluation of a statement. It can be combined with the `and` and `or` operators.

```
if (~op:lt(?var, 10)) {
  op:log("info", "?var is NOT lower than 10");
}

if (~act:isMoving()) {
  !failCond = op:invokeStaticMethod("com.fol.Factory", "createPredicate", "not
  exit(FAIL, !failCond);
}
```

2.4.7 Try, Catch, Finally

The `try`, `catch`, and `finally` elements allow handling of errors in action scripts. Action scripts stop executing as soon as an error happens unless the error is caught and handled by a try-catch block. The instructions between try and the first catch element are executed sequentially. If an instruction fails, the execution jumps to the catch block corresponding to the failure status. The catch statement can take two (optional) arguments. The first argument is the failure status to catch in this block and the second is the name of the variable in which to store information about the failure (currently a simply a predicate, the "fail condition"). This variable does not have to be declared as a local variable in the usual way. Just like in Java, the finally block will be executed regardless of failure or success in the try block.

```
try {
  act:walkForward();
} catch(FAIL_PRECONDITIONS, !error) {
  op:log("info", "Cannot walk forward: !error");
  exit(FAIL_PRECONDITIONS, !error);
} catch(FAIL_OVERALLCONDITIONS, !error) {
  act:stopMoving();
  op:log("info", "Cannot keep walking: !error");
  exit(FAIL_OVERALLCONDITIONS, !error);
} finally {
  op:log("info", "In finally block.");
}
```

2.4.8 Exit

The `exit` control element stops the execution of an action script. It takes two (optional) arguments, the ActionStatus Enum to exit with and a predicate ("fail condition"). Refer to the example given for the try, catch block and have a look at how the `exit` control is used.

2.4.9 Return

The `return` control element will return from a script in much the same way as in Java or most other languages.

```
() = exitTest() {
  if (true) {
    op:log("info", "this should execute");
    return;
  }
  op:log("error", "this should NOT execute");
}
```

2.4.10 Async and Join

The `async` control element is used to launch an asynchronous execution within an action script. Code within an async block is all executed in a new child ActionInterpreter which is executed inside a new thread. Async returns a single argument that gets bound to a unique identifier for the async context.
The `join` control element provides a mechanism to wait on an async context to finish execution and takes up to two args. The first arg is required and specifies the particular async context to join on (identifier returned from the async element). The second (optional) arg specifies the time to wait before returning (in ms). If no timeout is passed in, the join will wait until termination of the async context. The ActionStatus of the completed (or timed-out) async context is returned.

```
() = asyncTest() {
  java.lang.Long !asyncId;
  com.action.ActionStatus !actionStatus;

  !asyncId = async {
    op:sleep(5000);
    op:log("info", "first step in async");
    op:log("info", "second step async");
  }
  op:log("info", "before join. async id: !asyncId");

  join(!asyncId, 1000);
  op:log("info", "after join with timeout. async id: !asyncId status: !actionS

  !actionStatus = join(!asyncId, 1000);
  op:log("info", "after join with timeout. async id: !asyncId status: !actionS

  !actionStatus = join(!asyncId);
  op:log("info", "after join. async id: !asyncId status: !actionStatus");
}

() = asyncTest2() {
  java.lang.Long !asyncId;
  com.action.ActionStatus !actionStatus;

  !asyncId = async {
    async {
      op:sleep(5000);
    }
    async {
      op:log("info", "first async");
    }
    async {
      op:log("info", "second async");
    }
  }
  op:log("info", "before join. async id: !asyncId");

  !actionStatus = join(!asyncId, 3000);
  op:log("info", "after join with timeout. async id: !asyncId status: !actionS

  !actionStatus = join(!asyncId);
  op:log("info", "after join. async id: !asyncId status: !actionStatus");
}
```

2.4.11 True

The `true` control element evaluates to true (hence its name). It can be used for things like infinite while loops, using the `true` element as a condition.

2.5 Imports and Name Aliasing

You can import classes you want to use later to avoid typing the full name repeatedly. This is only useful for declaring variables (input variable, local variable, or return variable).

```
import java.lang.String;

String !str;
```

You can also rename an imported class as another name (this will reduce the readability of code).

```
import java.lang.String as MyStr;

MyStr !str;
```

2.6 Beliefs, Conditions and Effects

Part of the Goal Manager's task is to keep track of the current state of the world, check if actions are permissible and update the state with the effects of actions. Action's StateMachine does all of this and also keeps a history of states to be able to justify its choices. The logic reasoning happens in the Belief Component, which is automatically instantiated by the StateMachine; there is no need to start a separate Belief Component. As explained in section 2.2, conditions and effects can be specified with java annotations or within an action script.

2.6.1 Conditions

Conditions are a set of predicates that must hold at certain points in time in order for an action to be executed. These predicates will be observed (if possible) or inferred using the StateMachine and the associated Belief Component. The following condition types exist:

- Pre Conditions: conditions that must hold before the execution of an action
- OverAll Conditions: conditions that must hold during the execution of an action
- Post Conditions: conditions that must hold after the execution of an action

Note that post conditions cannot be directly specified, they consist of a subset of effects (union of Always and Success effects), see section 2.6.2 as well as sections 2.1.1 and 2.1.2. If a post condition is observable (i.e. there is an observer mechanism available), it will be checked and the action will fail if the observation is not successful. If a post condition is not observable, it is just assumed to hold, unless it is explicitely flagged as observable, in which case the post condition check will fail.

2.6.2 Effects

Effects are a set of predicates that represent the result of an action. The following types of effects exist:

- Always Effects: effects that are true from the start of an action (no matter the outcome, success or fail).
- Success Effects: effects that are true at the end of an action if it succeeds.
- Failure Effects: effects that are true from the point in time when an action fails.
- Non Performance Effects: effects that are true if an action is not executed (not implemented).

Effects will be observed if possible and the action will fail if the observation if not successful. If there is no observer mechanism available for an effect, the effect is just assumed to be true, unless it is explicitely flagged as observable, in which case the effect will not be applied and the action will fail (post-condition failure).

Effects (predicates) can be retracted (removed) from the State Machine and the Belief Component by wrapping the predicate in a `not()`.Currently, because of this retraction mechanism, there is no way to directly assert (add) `not` predicates.

2.6.3 Binding of predicates in conditions and effects

Arguments in predicates are bound to their respective variables. For instance, an action named `moveTo` with arguments `?start` and `?destination` can have the following conditions and effects:

- Pre Condition: `at(?start)`
- Success Effect: `at(?destination)`
- Success Side Effect: `not(at(?start))`

When executing such an action with the following actspec:

```
act:moveTo(A,B);
```

the Goal Manager will first check that the agent is `at(A)`, then execute the action and finally, if the move was successful, add the effect `at(B)` and retract `at(A)` (because of the `not()`)to the agent's State. Note that the variable binding happens when the condition or effect is used, i.e. at check-time or state-update-time (which depends on effect type).

Unbound variables in conditions are treated as free variables during condition checking and bound to a value if a solution is found. If more than a single solution is found for a condition, the first result is used. This mechanism can be used in the following way, again taking the example of a moveTo action:

- Pre Condition: `at(!start)`
- Success Effect: `at(?destination)`
- Success Side Effect: `not(at(!start))`

Here, the `!start` (local) variable can be looked up during pre-condition checking, bound to a value found in Belief (e.g. `A`) and then retracted as a success side effect, without having to pass in its value during an action call: `act:moveTo(B);`.

2.7 Handling multiple agents

The Goal Manager is capable of handling multiple agents running at the same time within a single instance. Each action has an implicit `?actor` variable that tracks the actor executing this action. When `the ?actor` binding is looked up, it will recurse up to the nearest context where it is defined. If the lookup reaches the RootContext, `?actor` is set to the "super" agent name set with the `-agentname` command line argument (see section 1.2).

The actor name can be set in two different ways:

- Calling an action through a postcondition containing the `?actor` variable. The generic post condition always has `?actor` as the first argument (see section 2.1.2).
- Calling an action the "object-oriented" way: `?actor.act:standUp();` will ask `?actor` to stand up (can be a string constant). Effects of actions can (should) contain this `?actor` variable to be agent-specific. This way, it is possible to keep track of each agent's state of the world. The "nao shared mental model" demo uses this to allow robots to reason about other robot's state.

2.7.1 Agent-specific action primitives

It is possible to identify DIARC components (and their action primitives) as belonging to an agent in particular. This can be useful when two robots of the same type are running simultaneously. For instance, in the "nao shared mental model" demo, two naos and their NaoComponents are up and running at the same time. In order to allow action to call the action primitives on the right components, each component is placed in a group corresponding to the agent it is in charge of, with the following syntax: `agent:robotname`. In the "nao shared mental model" case, one NaoComponent is in the group `agent:shafer` and the other in the group `agent:dempster`. In the event where no agent-specific action can be found, a generic action (not belonging to any agent in particular) will be used, if available. Otherwise the execution will fail.

2.8 Observations and Observer actions

The observer mechanism in action provides functionality to explicitely check if a fact holds by observing the world, through an DIARC Component such as Vision. Actions can provide observations, either by specifiyng a predicate in the `@Observes` annotation in the case of an action primitive or with the `observes` element in the case of a script.

Actions that can observe if a predicate holds ("observers") have to take a single Term (or Predicate) as an argument and return a list of maps between Symbols and Variables. The passed in predicate is what has to be observed and the returned list of maps provides the bindings in the case when free variables are present in the predicate, similarly to how the Belief Component works.

By default, Action will always try to observe a condition or effect. If no observer is available for a condition or effect it will be inferred via Belief. This mechanism can be changed by explicitely specifing the condition or effect as "observable", in which case it has to be observed and the action will fail if an observation cannot be made. A condition or effect can also be specified as "inferrable", in which case it will only be inferred through Belief.If a condition or effect cannot be observed the action will fail with thecorresponding error (FAIL).

Just like regular conditions, observable conditions can be used to get bindings for free variables. For instance a `on(!object, table)` pre-condition can be used to retrieve the object that is on the table via an observation. The observer will return a binding for the `!object` variable if there is such an object on the table.

3 Internal workings of the Goal Manager

The section describes the inner workings of the Goal Manager. It is intended for the developer interested in understanding, debugging and expanding the Goal Manager.

The GoalManager code can be divided in two parts:

1. The “goal management” part, which initializes the Action Database by reading action scripts and importing action primitives, handles the submission of new goals and responds to queries for currently running goals. Other DIARC Components interface with this part of the Goal Manager.

2. The “action execution” part, which sets up the instructions and contexts (scopes) for the ActionIntepreter to run. This side is not exposed to external DIARC Components.

3.1 Goal Management

The Goal Manager (DIARC Component) initializes the Action Database and collects information about the action primtives provided by the DIARC Components currently registered with TRADE. It also parses action scripts and loads them into the Action Database. The Basic Goal Manager, which is responsible of goal submission and goal tracking, is instantiated by the Goal Manager.

3.1.1 GoalManager(Impl)

The GoalManager is the top-level DIARC Component. It provides the functionality to set up the Action system (parsing action scripts, searching for action primitives, instantiating the BasicGoalManager).

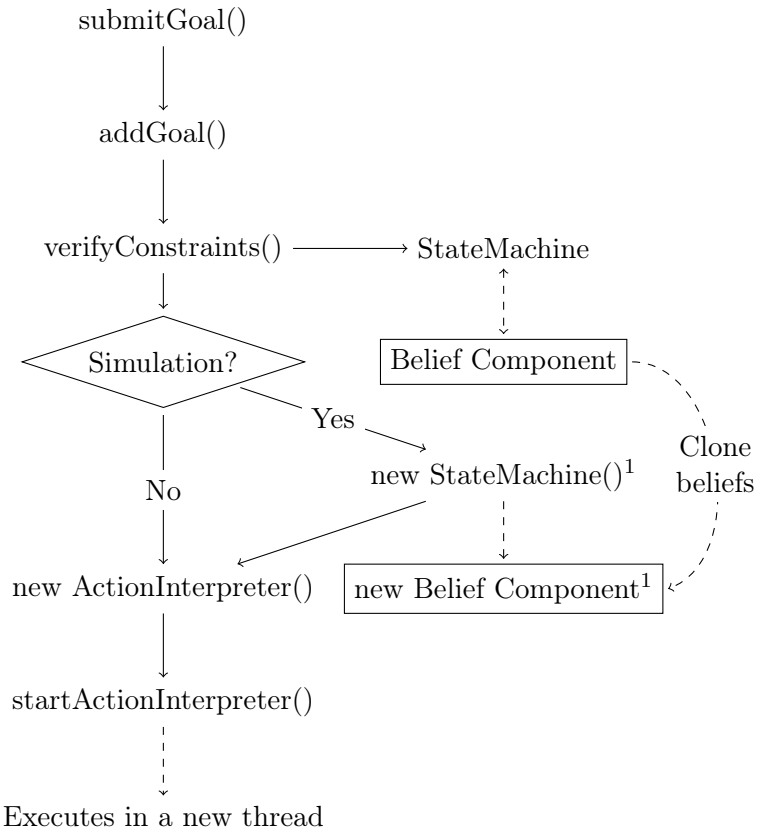
3.1.2 BasicGoalManager

The BasicGoalManager handles goal submission and tracking. Eventually, we would like to have one BasicGoalManager instance in each DIARC Component, to allow the components to locally manage goals. When the BasicGoalManager in instantiated, it creates a StateMachine which will track the state of the world, using a BeliefComponent (automatically spawned). When a goal is submitted, it verifies the goal against the constraints and, if it is acceptable, instantiates a new ActionInterpreter. If the goal is to be simulated, the StateMachine is cloned, a new Belief Component is instantiated and used for tracking the state in the ActionInterpreter for that goal, see figure 3.

3.2 Action Execution

3.2.1 ActionInterpeter

The ActionInterpreter executes the action scripts, step-by-step. It implements the `Callable` interface to execute the goals in a concurrent, non-blocking manner. For each cycle, the runCycle() method is called. It handles the action execution stack, calls the context execution method and fetches the next step, if there's one.



[1]New StateMachine/Belief instance will be used in the ActionInterpreter instead of the usual (real world) one.

Figure 3: Goal sumission process within BasicGoalManager.