ADE Action Manager

Paul W. Schermerhorn

August 26, 2011

Contents

1 Introduction

The ADE Goal Manager (commonly referred to as the Action Manager) performs scheduling implicitly, based on the priorities of the agent's currently active *top-level goals* (TLG). A TLG represents a goal of the agent that is generally not being pursued in service of any other goal. A TLG may be explicitly instantiated on behalf of the user (e.g., a script for the agent to execute) or may be implicitly instantiated by some system-level motivation (e.g., the obedience motivation can cause a TLG to be instantiated in response to a verbal command from the user).

Associated with each TLG is an *action interpreter* (AI), the job of which is to execute the actions (typically encoded in the form of a script) required to accomplish the goal. AIs execute in parallel in their own threads, allowing simultaneous progress toward the achievement of multiple TLGs.

The Action Manager continually reevaluates the *priority* of each TLG, based on factors including the utility of accomplishing the goal, the time remaining within which to accomplish the goal, and the agent's current affective state. Priority scores are used to mediate resource conflicts between TLGs; before attempting an action, the AI checks to see whether another AI already has the lock for the contested resource and, if so, compares the priorities of the two. For example, a robot may be executing a low-priority wandering script that has control of the wheel motor lock. If the robot receives a command from a person instructing it to perform a task requiring it to stay in place, the higher priority of the follow-commands goal will allow that AI to stop the wheels and keep the robot from wandering off. in the meantime, the AI for the wandering goal is still functioning, trying to issue velocity commands at random intervals. When the follow-commands goal has completed, then, the wandering behavior automatically proceeds in virtue of having regained control of the resource lock.

2 Writing AI Scripts

Action Interpreter scripts are stored in XML format. Hence, a description of how to write an AI script must include descriptions of both the syntax of the scripts and the syntax of the XML format in which they are stored. For example:

```
<type>
  <name>findDoor</name>
  <dbkind>action</dbkind>
  <desc>?mover looks for a door at a given heading ?heading</desc>
  <var>
    <varname>?mover</varname>
    <vartype>actor</vartype>
  </var>
  <var>
    <varname>?heading</varname>
    <vartype>coordinate</vartype>
  </var>
  <var>
    <varname>!headingFrom</varname>
    <vartype>coordinate</vartype>
  </var>
  <var>
    <varname>!headingTo</varname>
    <vartype>coordinate</vartype>
  </var>
  <!--  Here's where the events start -->
  <actspec>getHeading ?mover !headingFrom</actspec>
  <actspec>printText "headingFrom: !headingFrom"</actspec>
  <actspec>getHeadingTo ?mover !headingFrom ?heading !headingTo</actspec>
  <actspec>printText "headingTo: !headingTo"</actspec>
  <control>if</control>
  <actspec>  gt !headingTo 15</actspec>
  <control>then</control>
  <actspec>  timeTurn ?mover !headingTo</actspec>
  <control>else</control>
  <actspec>  printText "Not worth turning"</actspec>
  <control>endif</control>
  <actspec>startMove ?mover</actspec>
  <control>while</control>
  <actspec>  not</actspec>
  <actspec>    getNearestDoor ?mover</actspec>
  <actspec>  endnot</actspec>
  <control>do</control>
  <actspec>  printText "Still looking for a door"</actspec>
  <control>endwhile</control>
  <actspec>stop ?mover</actspec>
</type>
```

This simple script has the robot turn to the heading indicated and move forward (presumably down a hallway) until it encounters something matching the description of a doorway. The XML representation of the script (above) really just packages a more traditional scripting language:

```
:name findDoor
:desc ?mover looks for a door at a given heading ?heading
:var ?mover actor
:var ?heading coordinate
:var !headingFrom coordinate
:var !headingTo coordinate
# Here's where the events start
getHeading ?mover !headingFrom
printText "headingFrom: !headingFrom"
getHeadingTo ?mover !headingFrom ?heading !headingTo
printText "headingTo: !headingTo"
if
  gt !headingTo 15
then
  timeTurn ?mover !headingTo
else
  printText "Not worth turning"
endif
startMove ?mover
while
  not
    getNearestDoor ?mover
  endnot
do
  printText "Still looking for a door"
endwhile
stop ?mover
```

2.1 Action Scripting Language

The AI steps through scripts, looking for primitives or syntax elements (below). When it encounters a primitive, it executes the action associated with it.

Flow Control The semantics of script syntax elements is as follows:

Conditional: if α then β elseif γ then δ else ϵ endif

Loop: while α do β endwhile

Negation: not α endnot

Conjunction/Disjunction: α and β, α or β

Return return TRUE|FALSE endreturn

Note the absence of a test primitive; any action can serve as the condition for loops and conditionals, with its exit status serving to determine the outcome of the branch.

Variables Action script variables are typed, and come in two varieties: locals (created at call time and discarded when leaving the script) and parameters (passed into the script at call time). Local variable names begin with the character '!'. Locals can be used as parameters for script invocations from the current script context.

Parameter variable names begin with the character '?'. Parameters are passed by reference, so the binding of a variable may be different when a script invocation returns from what it was before the invocation. This, in fact, is how values are "returned" from scripts—the user passes in the variable to which the desired return value should be bound. For example, in the code above, after `getHeading ?mover !headingFrom` completes, the current heading of the robot bound to `?mover` is stored in the variable `!headingFrom`. Although somewhat awkward, this design facilitates two features important to ADE action script semantics. First, because every event in a script is considered an action, including sensing acts, there is the possibility of failure. The more traditional (i.e., as in C or Java) return value of an action, therefore, is a truth value indicating the success or failure of the action. This requires that other values be passed back by alternate means. Second, in many cases it is imperative in the context of embodied agents to have the flexibility to adapt to changes in the situation. Take the example of an interaction script that takes a person-type variable as a parameter. If the original interlocutor were to leave during the interaction while another person carried on the conversation, changing the binding for the calling context would prevent the robot from using the wrong name or saying something inappropriate after the script returns.

While it is always best to specify all parameters at invocation time, it is possible to call a script without all parameters. When this happens, the AI tries to bind the variable itself, for example, by looking for a variable of the same name in the parent's context and looking for (differently-named) variables of the same type in either the child or parent context. This is based on the assumption that, if the binding is unspecified, the context itself may be able to provide a reasonable guess (analogous to anaphora resolution). However, this can lead to unexpected results, so it is advised to pass these values in whenever possible.

Actions An action statement is just the name of the action followed by the names of the parameters to be passed in. As stated above, the return value is true or false, depending on the success or failure of the action. Hence, *any action can be used as a test for the* `if` *and* `while` *constructs*.

2.2 Action Database File Syntax

An Action database file is a hierarchical specification of bits of knowledge (primarily primitive actions and scripts, but also other types such as numbers, etc.). ADE action script files are usually found in the directory `$ADE/com/action/db`. The default set of action types, primitives, and scripts is loaded automatically when the goal manager is initialized. Additional files can be specified on the command line using the `-dbfilesupp` parameter. The XML structure if these files includes:

`type` Specifies the type being described (e.g., action, thing).

`subtypes` Contains a list of types that are hierarchical subtypes of the current type.

`name` The name of the current type (this is case INSENSITIVE).

`desc` A description that can be used by Action.

`postcond, precond` Specify pre- and post-conditions. These are not used used consistently right now, but their specification is encouraged.

`benefit, cost, timeout` As you would expect.

`minurg, maxurg` The minimum and maximum urgency for a script. The defaults are 0.0 and 1.0.

`var` A variable definition. Variables are typed, and must include the following:

`varname` The name of variable.

`vartype` The variable's type (e.g., integer, double).

`actspec` An action specification. Can be a primitive action or a script. When using variables, include the '?' or '!' character (which is part of the name), and the ActionInterpreter will look up and pass the appropriate variable binding into the action.

`control` A flow control element (e.g., if, while).

When the goal manager loads the script files, it uses the tags to aid in parsing. However, it is not necessary to use the XML format when creating new scripts; a script specified using the non-XML format shown above can be converted to an action script file using the `Scriptify` utility included with ADE (in `$ADE/com/action/db`).

2.3 Writing Action Scripts without XML

`Scriptify` is a simple parser that allows you to specify Action scripts as non-xml text, similar to more traditional scripting languages. Comment lines start with '#'. All tags, except for ¡var¿, ¡control¿, and ¡actspec¿, are represented by ":tag [value]" at the start of a line. A var is represented as ":var [varname] [vartype]" at the start of a line. Any line that isn't matched to one of the other tags is assumed to be an actspec or control–no tag required.

Quickly, the elements that are required for a script are:

`:name` the name

`:var <varname> <vartype>` the "variables" for the script

`:actspec` the actions the script calls (primitives or other scripts); note that the `:actspec` tag is optional for Scriptify

`:control` flow control elements (e.g., if, while) note that the `:control` tag is optional for Scriptify

Other possible elements are:

`:locks` specify which resource locks should be required before executing

`:cost` the cost of running the script

`:benefit` the expected benefit

`:posaff` initial value for positive affect associated with this action

`:negaff` initial value for negative affect associated with this action

`:minurg` minimum urgency for this action

`:maxurg` maximum urgency for this action

`:timeout` time allotted in which to complete action

`:precond` predicate list specifying preconditions

`:postcond` predicate list specifying postconditions

Most of these have default values and need only be specified if you want to tweak the goal selection process, etc.

Remember that when referring to parameter variables in the actspecs, they must be preceded with a '?'. Also note that "local" variables (ones that the Action Interpreter should not expect to find in the call stack) should be defined with a leading '!'.

Generating XML Ensure the utility is compiled (`./mkade scriptify`) and then run it on your input to generate XML output that can be read by the Action Manager. For example, to translate the file `test.as`:

```
java -cp .:jars/xercesImpl.jar com.action.db.Scriptify test.as
```

in $ADE. The result is printed to System.out. It can then be pasted into an appropriate xml ActionDB file. Alternatively, to output a full file, pass the "-s" flag for a new script, or the "-p" flag for a new primitive, and redirect the output to a new xml file.

3 Semantic Description

Note that in every case below, the absence/unavailability of the underlying ADE component (e.g., the motion server) results in action failure. Similarly, if the AI in which an action is attempted does not hold the appropriate locks, the action fails. One postcondition of any failure is that the action's exit status is set to false. In many cases, a failure has no effect at the architecture level, but may be meaningful at the script level.

Note also that actions commonly take the agent itself as a parameter (exceptions to this rule of thumb are mostly found in the "miscellaneous other actions" category). One important reason for this seemingly extraneous parameter is to allow for simulation of other agents' actions. If the agent is imagining what some other agent would do in a given situation, it is important to avoid performing those actions in the real world. Hence, most primitives will check to make sure that the actor parameter matches the name of the agent before dispatching the action. The practical impact of this is that you need to make sure the value passed for the actor parameter is correct; the default value is "robot" but it can be specified when the Action Manager is started using the `-agentname` command-line parameter.

3.1 Variable Types

The Action Manager maintains a type hierarchy for variables used in scripts. There is (currently) no strict type checking, however, because other components rely on typing to function properly (e.g., the planning server determines what to pass to scripts based on the types it's given). It is, therefore, advisable to adhere to the type hierarchy to the extent possible.

```
object: type
 location: type
  hallway: type
  room: type
  zone: type
 doorway: type
  door: type
 box: type
 property: type
  color: type
  open_pr: type
  closed_pr: type
 agent: type
  actor: type
  interactor: type
  listener: type
 thing: type
 data: type
  text: type
  keys: type
  unit: type
  perspective: type
  direction: type
  relationship: type
  datatype: type
  percept: type
 boolean: type
 number: type
  integer: type
  long: type
   visionTs: type
   millisec: type
  double: type
   coordinate: type
```

3.2 Actions Related to Motion

There are three primary types of motion-related commands: *targetless*, *targeted*, and *informational*. Targetless motion actions attempt to initiate some basic motion (e.g., move forward, turn right) that has no specific target location and will continue until explicitly terminated (e.g., move forward will cause the agent to move forward until stopped by a command, obstacle avoidance, etc.). Targeted motion commands initiate actions that will not continue indefinitely, but will stop when the target requirements have been satisfied. Note that an AI passes such targeted motion actions to the motion server, which handles the specifics of reaching the

target–it is treated as a black box. Hence, there is a level of state below the level of the Action Manager to which it has no access beyond the current overall state of the targeted action (in progress, success, failure, etc.). Finally, informational actions attempt to provide some kind of information to the overlying script (e.g., the agent's current location).

```
setTV: set translational velocity
  Parameter vars:
    ?mover: actor
    ?tv: double - translational velocity
  Resource locks:
    motionLock


setRV: set rotational velocity
  Parameter vars:
    ?mover: actor
    ?rv: double - rotational velocity
  Resource locks:
    motionLock


setVels: set translational and rotational velocities
  Parameter vars:
    ?mover: actor
    ?tv: double - translational velocity
    ?rv: double - rotational velocity
  Resource locks:
    motionLock


getVels: get translational and rotational velocities
  Parameter vars:
    ?mover: actor
  Return vars:
    ?tv: double - translational velocity
    ?rv: double - rotational velocity
  Resource locks:
    motionLock


move: move in the specified direction (straight, forward, left, right, back)
  Parameter vars:
    ?robot: actor
    ?direction: direction
  Resource locks:
    motionLock


qmove: move in the specified direction (straight, forward, left, right, back)
       with no verbal confirmation
  Parameter vars:
    ?robot: actor
    ?direction: direction
  Resource locks:
    motionLock
```

```
turn: turn in the specified direction (left, right)
  Parameter vars:
    ?robot: actor
    ?direction: direction
  Resource locks:
    motionLock

qturn: turn in the specified direction (left, right) with no verbal
       confirmation
  Parameter vars:
    ?robot: actor
    ?direction: direction
  Resource locks:
    motionLock

stop: stop motion
  Parameter vars:
    ?mover: actor
  Resource locks:
    motionLock

qstop: stop motion with no verbal confirmation
  Parameter vars:
    ?robot: actor
  Resource locks:
    motionLock

safeRight: check whether there's an obstacle to the right
  Parameter vars:
    ?mover: actor

safeFront: check whether there's an obstacle to the front
  Parameter vars:
    ?mover: actor

safeLeft: check whether there's an obstacle to the left
  Parameter vars:
    ?mover: actor

getMotionTolerance: get max distance from destination for successful
                    motion
  Parameter vars:
    ?mover: actor
  Return vars:
    ?dist: double - motion tolerance

setMotionTolerance: set max distance from destination for successful
                    motion
  Parameter vars:
    ?mover: actor
    ?dist: double - motion tolerance
```

```
getCritDist: get the critical distance at which obstacle avoidance
             will engage (if active)
  Parameter vars:
    ?mover: actor
  Return vars:
    ?dist: double - critical distance


setCritDist: set the critical distance at which obstacle avoidance
             will engage (if active)
  Parameter vars:
    ?mover: actor
    ?dist: double - critical distance


getStall: check whether motors are stalled (adesim only)
  Parameter vars:
    ?mover: actor


atLocation: check whether current location is within epsilon of given
            coordinates
  Parameter vars:
    ?mover: actor
    ?xdest: coordinate - x coordinate
    ?ydest: coordinate - y coordinate
    ?epsilon: coordinate - allowable error


getLocation: get current location
  Parameter vars:
    ?mover: actor
  Return vars:
    ?xcoord: coordinate
    ?ycoord: coordinate


getHeading: get current heading
  Parameter vars:
    ?mover: actor
  Return vars:
    ?heading: double


getHeadingTo: get relative heading between two global headings
  Parameter vars:
    ?mover: actor
    ?t1: double - current heading
    ?t2: double - target heading
  Return vars:
    ?heading: double
```

```
getHeadingFrom: get global heading between one point and another
  Parameter vars:
    ?mover: actor
    ?x1: coordinate - current x coordinate
    ?y1: coordinate - current y coordinate
    ?x2: coordinate - target x coordinate
    ?y2: coordinate - target y coordinate
  Return vars:
    ?heading: double - target heading

getHeadingFromRel: get relative heading between one point and another
  Parameter vars:
    ?mover: actor
    ?x1: coordinate - current x coordinate
    ?y1: coordinate - current y coordinate
    ?x2: coordinate - target x coordinate
    ?y2: coordinate - target y coordinate
    ?heading: double - current heading
  Return vars:
    ?newHeading: double - target heading

getDistanceFrom: get distance between one point and another
  Parameter vars:
    ?mover: actor
    ?x1: coordinate - current x coordinate
    ?y1: coordinate - current y coordinate
    ?x2: coordinate - target x coordinate
    ?y2: coordinate - target y coordinate
  Return vars:
    ?dist: coordinate - distance

moveTo: move to a given global location
  Parameter vars:
    ?mover: actor
    ?xdest: coordinate - destination x coordinate
    ?ydest: coordinate - destination y coordinate

traverse: move until an obstacle is encountered ahead, avoiding
          obstacles to the sides
  Parameter vars:
    ?mover: actor

moveThroughRel: move through the nearest open doorway
  Parameter vars:
    ?mover: actor

moveRel: move to a location relative to the current one (heading
         assumed to be 0)
  Parameter vars:
    ?mover: actor
    ?xdest: coordinate - destination x coordinate
    ?ydest: coordinate - destination y coordinate
```

```
timeMove: move forward a given distance (dead reckoning)
  Parameter vars:
    ?mover: actor
    ?dist: double - distance to move (in meters)


timeTurn: turn a given amount (dead reckoning)
  Parameter vars:
    ?mover: actor
    ?heading: double - amount to turn (in radians, positive is
              counter-clockwise)
```

3.3 Actions Related to Vision

```
getTokenArea: get the area of the bounding box for the vision token
              corresponding to ?vKey
  Parameter vars:
    ?viewer: actor
    ?vKey: long - key for the vision token
  Return vars:
    ?area: double - area of the bounding box for the vision token


getTokenPanTilt: get the pan/tilt of the bounding box for the vision
                 token corresponding to ?vKey
  Parameter vars:
    ?viewer: actor
    ?vKey: long - key for the vision token
  Return vars:
    ?pan: double - pan angle of the 'center of gravity' of the
                   bounding box for the vision token
    ?tilt: double - tilt angle of the 'center of gravity' of the
                    bounding box for the vision token


getTokenColor: get the color identifier for the vision token
               corresponding to ?vKey
  Parameter vars:
    ?viewer: actor
    ?vKey: long - key for the vision token
  Return vars:
    ?color: text - the color identifier for the vision token


getAllByType: get the keys for all tokens of type ?vType in visual STM
  Parameter vars:
    ?viewer: actor
    ?vType: text - vision type
  Return vars:
    ?STMKeys: keys - list of keys for ?vType objects detected
```

```
getByColor: get the key for one token of color ?vColor in visual STM
  Parameter vars:
    ?viewer: actor
    ?vColor: text - blob color
  Return vars:
    ?STMKey: long - the key for a ?vColor object


getAllByColor: get the keys for all tokens of color ?vColor in visual STM
  Parameter vars:
    ?viewer: actor
    ?vColor: text - blob color
  Return vars:
    ?STMKeys: keys - list of keys for ?vColor objects detected
```

3.4 Actions Related to Speech Production

```
sayTextNow: say something to someone
  Parameter vars:
    ?speaker: actor
    ?target: interactor
    ?statement: text - the phrase to be spoken (vars will be bound)
  Resource locks:
    speechLock


changeVoice: change Festival voice
  Parameter vars:
    ?speaker: actor
    ?voice: data - new voice name


changeMood: change Festival voice mood (currently requires mbrola)
  Parameter vars:
    ?speaker: actor
    ?mood: data - mood string (happy, sad, etc.)
```

3.5 Miscellaneous Other Actions

These are primitives that are more conveniently handled by the interpreter.

```
getNearestDoor: get the nearest detected doorway
  Parameter vars:
    ?mover: actor
  Return vars:
    ?x: coordinate - x coordinate of the doorway
    ?y: coordinate - y coordinate of the doorway
    ?ex: coordinate - x coordinate of the 'exit' point (just through
                      the doorway)
    ?ey: coordinate - y coordinate of the 'exit' point (just through
                      the doorway)
    ?ax: coordinate - x coordinate of the 'approach' point (just before
                      the doorway)
    ?ay: coordinate - y coordinate of the 'approach' point (just before
                      the doorway)
    ?dist: coordinate - distance to the doorway

getTimeOfDay: get time of day in milliseconds
  Return vars:
    ?time: long

stringCompare: check whether two strings match
  Parameter vars:
    ?stringOne: text
    ?stringTwo: text

stringContains: check whether ?stringOne contains ?stringTwo
  Parameter vars:
    ?stringOne: text
    ?stringTwo: text

getNewList: create a list
  Return vars:
    ?list: object - the newly-created list

getListSize: get number of elements in a list
  Parameter vars:
    ?list: object - a list
  Return vars:
    ?val: integer - the number of elements in ?list

getListElement: get an element from a list
  Parameter vars:
    ?list: object - a list
    ?index: integer - the index of the element to retrieve
  Return vars:
    ?val: object - the requested element

addListElement: add an element to the end of a list
  Parameter vars:
    ?list: object - a list
    ?val: object - the element to be added to ?list
```

```
delListElement: delete an element from a list and return the element
  Parameter vars:
    ?list: object - a list
    ?index: integer - the requested element
  Return vars:
    ?val: object - the deleted element

actionSucceed: exit current script with success exit status

actionFail: exit current script with failure exit status

true: boolean value true

false: boolean value false

set: assign a value to a variable
  Parameter vars:
    ?value: entity - the new value of ?target
  Return vars:
    ?target: entity - the variable to set

printText: print the argument text (variables will be bound)
  Parameter vars:
    ?text: text

logText: log the argument text (variables will be bound)
  Parameter vars:
    ?text: text

+: arithmetic addition
  Parameter vars:
    ?arg1: number
    ?arg2: number
  Return vars:
    ?sum: number

-: arithmetic subtraction
  Parameter vars:
    ?arg1: number
    ?arg2: number
  Return vars:
    ?diff: number

*: arithmetic multiplication
  Parameter vars:
    ?arg1: number
    ?arg2: number
  Return vars:
    ?prod: number
```

```
%: arithmetic modulus
  Parameter vars:
    ?arg1: integer
    ?arg2: integer
  Return vars:
    ?quot: number

/: arithmetic division
  Parameter vars:
    ?arg1: number
    ?arg2: number
  Return vars:
    ?quot: number

round: round the parameter to the nearest long integer
  Parameter vars:
    ?arg: double
  Return vars:
    ?round: long

lt: arithmetic comparison: less than
  Parameter vars:
    ?arg1: number
    ?arg2: number

le: arithmetic comparison: less than or equal to
  Parameter vars:
    ?arg1: number
    ?arg2: number

gt: arithmetic comparison: greater than
  Parameter vars:
    ?arg1: number
    ?arg2: number

ge: arithmetic comparison: greater than or equal to
  Parameter vars:
    ?arg1: number
    ?arg2: number

=: arithmetic comparison: equal to
  Parameter vars:
    ?arg1: number
    ?arg2: number

randomInteger: return a random integer in the given range
  Parameter vars:
    ?lower: integer - lower bound
    ?upper: integer - upper bound
  Return vars:
    ?value: integer - random value
```

```
randomDouble: return a random double in the given range
  Parameter vars:
    ?lower: double - lower bound
    ?upper: double - upper bound
  Return vars:
    ?value: double - random value


startADEServerLogging: start ADE server logging


stopADEServerLogging: stop ADE server logging


getTeammateLoad: get teammate load (e.g., from fnirs server)
  Parameter vars:
    ?robot: actor
  Return vars:
    ?load: double


sleep: sleep
  Parameter vars:
    ?sleeper: actor
    ?sleepMillis: millisec - sleep duration in milliseconds
```

4 Variants of the Action Manager

There are three variants of the Action Manager, each distinguished by a different approach to calculating goal priority.

ActionManagerLinear assigns every TLG an identical priority regardless of time remaining, net benefit, etc. This effectively implements a first-come, first-served scheduling policy: no preepmtion is possible, because no higher-priority goal will ever be instantiated.

ActionManagerPriority assigns priority based on the net benefit of the TLG and its urgency. Urgency, in turn, is based on how much time is remaining (but may be bounded minimum and maximum urgency values).

ActionManagerAffective assigns priority based on the net benefit and urgency of the TLG, but the cost and benefit components are scaled according to the negative and positive affective states of the Action Manager. In addition, actions may have affective evaluations attached to them (e.g., depending on whether they were successful or not when executed previously); when multiple options are available to achieve a goal, action selection takes these affective evaluations into account.

5 Starting the Action Manager

The Action Manager is started like any other ADE server; this is often easiest achieved using the script `runadeserver`. [1] For example:

```
./runadeserver com.action.ActionManagerLinear
```

starts an Action Manager with no requested server references and no startup actions. The following parameters are accepted:

`-agentname <name>` set the name of the agent to <name>

`-subject <name>` set the default name for the interactor to <name>

`-script <name> [arg ...]` run the script <name> with parameters [arg] ...

`-dbfilesupp <name>` parse supplemental database file <name>

`-server <name>` acquire reference to server <name> (a fully-qualified interface name, either for a specific server, or from `com.interfaces`

[1] See the ADE Guide (`$ADE/doc/ADE Guide.pdf`) for details.