Activity Library

Documentation and Implementation Status

Overview

The activity library is responsible for scheduling actions to occur within a simulated world, and for making these actions actually happen at the right time in the right order. It provides the foundation of dynamic, object-oriented simulation within Swarm.

> Actions consist of messages to objects, calls to functions, or groups of actions in some defined order. The activity library guarantees that all these actions, and the state changes they produce, occur at predictable points in time. Time is defined by the relative order of actions, and may also be indexed by the discrete values of a world clock.

Documentation

The activity library follows the Swarm `library interface conventions`

(defobj/library.html)
and has the following standard sections of documentation:

- Usage Guide
(activity/guide.html)
- Advanced Usage Guide
(activity/adv.html)
- Interface Reference
(activity/ref.html)
- Subclassing Reference
(activity/subclass.html)
- Interface Design Notes
(activity/design.html)
- Implementation Notes
(activity/impl.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Activity Library: Documentation and Implementation Status

The activity library is in process of being converted to draw even more of its support from underlying types and classes in the defobj and collections libraries. The basic interface has already been changed to reflect this consolidation, so the programming interface for currently available function is expected to remain stable.

The activity library follows the documentation structure of the Swarm library interface conventions
(../defobj/library.html)
. There are placeholders for each section of documentation so that all links should at least link up with something, whether or not there's anything there. The Interface Reference section is already fairly complete, and there is a start toward a more complete Usage Guide section. The Advanced Usage Guide and other sections will come at a later time.

Throughout the documentation, a parenthesized comment that starts with (.. indicates an editorial comment on the current status of implementation or documentation. In this version, however, an attempt has been made to remove all discussion of future capabilities and to document instead only the capability that is already there.

Source for code examples in the Usage Guide is included in the text where it occurs. Source consists of fragments taken from example programs. The example programs themselves may be downloaded from the web pages and compiled on your own system, using links provided for that purpose. (.. currently there are no code examples in the Usage Guide, which so far is only a general overview of the activity library ..)

There is also a directory of test programs (GridTurtle test programs
(../grid.html)
, contained within the documentation release directory) that provides additional code examples of many basic features of the defobj, collections, and activity libraries. These code examples are very rough and subject to change, however, since their main use is with each new release of the libraries to test various new or existing features. They are sometimes the only source, however, for examples of some library features that have not yet been used anywhere else.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Activity Library: Usage Guide

(.. This is an overview of activity library function. Code examples needed by a full usage guide will follow later.)

Role of the activity library in Swarm

The activity library provides a foundation for dynamic, object-oriented simulation in Swarm. Swarm assumes that a user defines an object-oriented representation for the structure of a world to be simulated. Once such a representation is established, activity library components are responsible for generating all state changes and flow of information within it. These state changes must be carefully controlled to ensure that they occur only at appropriate times and places in the model. The activity library provides mechanisms to establish this control.

> Once the simulation of a dynamic model begins, everything that happens to the model occurs as a direct result of messages sent to world objects by activity library components. These components are also represented by objects, since this is the most effective means for their representation as well, but their purpose is not to represent the current state of the simulated world, but rather to generate changes in world objects by sending messages in a valid order. The activity library has two basic categories of components: those that represent messages to be sent, including constraints on permissible order for sending them, and those that execute the message sends these representations specify, making sure they conform with all constraints.
>
> All changes to a model occur as a result of messages sent to objects within it. These messages invoke compiled methods of receiving objects, which may change local state or send messages to other objects to propagate effects as widely as needed. Only top-level function calls or message sends that initiate model processing are the concern of the activity library. Because the activity library finally interacts with the world model just by invoking its defined methods, the model can prepackage as much behavior as it can in the form of fast compiled methods. The activity library just triggers the prepackaged behavior at proper times under its own explicit, dynamically interpreted representation.
>
> As conditions shift during execution of a model, the model can also examine and alter the representation of its future behavior contained in the activity structures that drive it. Any changes to this future behavior, however, still occur only as a result of some currently executing action initiated by the activity library.

Activity library components

The activity library defines one set of basic object types to provide a very rich representation of the kinds of message sending patterns it could generate. Its other major category of object types controls and tracks a current state of processing within these representations.

Both these representations are more abstract than typical object representations, since they deal not with any constant state which can be statically analyzed, but with shifting patterns of messages to be fired to generate such state. While the basic structure of a Swarm simulation is that of a discrete event simulation, its activity representation is also more complicated than most such systems.

There are two basic reasons for the potential complexity of a Swarm activity representation One is that Swarm supports a decentralized representation of activity to be generated, both to reflect the nature of much of the behavior that it simulates, and so that execution may be distributed across multiple parallel processors. To avoid any need for synchronizing its decoupled activities more often than necessary, Swarm enables a model designer to avoid overspecifying constraints on the patterns of message sends which might potentially be valid. The partial order representation it adopts for a distributed and decoupled plan of activity is inherently more complicated than the single centralized event list adopted by many discrete event simulation systems.

The other reason for potential added complexity is that Swarm's representation of intended activity may be broken into many separate, modular components, which can be bound together in various ways to create larger components. The Swarm composition structure, when fully implemented, will be fundamentally as powerful as the modular abstractions found in most programming languages, with the added complexity of controlling the time at which various events occur.

In spite of this high potential complexity, none of these features is needed for many simple models, such as those that contain only a small variety of basic behaviors, or which define all their behavior to occur at regular, repeating timesteps. The Swarm representation is extremely rich to enable it to scale to large, multi-level models with a variety of dependent behaviors built into the model, and also to facilitate running models on massively parallel machines. Many of the features which seem complex, moreover, are also especially well-suited to building modular and reusable library components. It is anticipated that many of the more advanced features will find their heaviest usage in pre-built libraries that hide internal complexity from applications that use them.

No matter how complex the structures built from them, all the activity library components finally result in the direct execution of messages sends to objects in a model. Because of this direct execution, very precise meaning can be defined for each of them, in terms of message sends that can or must occur. Every component of a structure to be executed, and every event of its execution, can also be accessed using the interfaces of an object-oriented representation. This means that tools can be built to understand and interact with any components to be executed, and also to trace and debug activity as it occurs. None of these tools have been built yet as part of Swarm, but many of the activity library interfaces are present to support them, not because normal use of the library requires the added components.

Action plan components

The first set of activity library components represent messages to be sent to objects in a model, together with constraints in the order in which they may be sent. All these components are defined as subtypes of the one generic type called ActionPlan.

The two basic kinds of action plans are a simple group of actions to be performed in some order, called an ActionGroup, and a series of actions to be performed at discrete points in time, called a Schedule. The basic element of an action plan is a simple object called an Action. An action defines a particular message to be sent to an object or objects.

In its representation of action plans, Swarm relies heavily on the dynamic message sending capabilities of the Objective C language. The support of Objective C for dynamic message sends is absolutely crucial to Swarm's implementation of generic activity structures. Objective C defines a special kind of data value called a selector, which identifies a message according to its name. One of these selector values is stored in each action of an action plan. During execution of the plan, Objective C machinery performs the actual message send using its selector.

Action plans are free-standing objects that may be created directly by a user program, using a variety of selectable create-time options. Once created, action plans may also refer to each other, by containing special kinds of actions to start another action plan or to perform all the actions within it.

Individual actions can be created only as completely controlled components of some action plan. They are created not by a standard create message, but by special messages (each containing the phrase createAction in its name) sent to an action plan that creates the action as part of the action plan. If the same action is needed in more than one plan, it has to be created in each plan where it is needed. If the entire action plan is dropped, all its actions are also automatically dropped. So the only action plan components which need to be directly managed by an application are the action plans themselves.

Action plans are relatively straightforward components, because they are entirely passive. The only actions they ever contain are those which an application has created in them. These actions stay in them indefinitely unless a special option is requested to clean them out as each one is executed. Because these plans are passive, read-only components to all execution machinery, if the same pattern of actions needs to be triggered at multiple points in a model, it's perfectly valid to create one copy of the actions in an action plan, and then refer to the plan anywhere the actions may be needed.

Even though action plans are passive, containing only what has been placed in them, there is no requirement that their contents remain fixed. Both of the action groups and schedules are implemented directly as collections of their actions. New actions may be added at any time, and existing ones may be dropped or moved from position to another. The contents of action plans may be as dynamic as they need to be to represent the future actions needed in a model. None of these shifting contents has any effect on the model until actually processed during model execution.

Model execution components

The main responsibility of model execution is just to perform the actions specified by an action plan, in an order of execution consistent with any requirements of the plan. Each action plan may specify as few or as many constraints as it likes over the possible order in which its actions could be performed. Given these specifications, the execution objects are entirely responsible for selecting each action to be performed and then performing them.

There are two simple cases of ordering that account for most all usage, including that of the current Swarm demo programs. One is to permit the actions to be performed in any order, including concurrent execution if hardware and software support were available to do this. The other is to require them to be performed in some specific sequence, one after another, so that the effects of one action could depend on actions which preceded it. This sequence could be either fixed and predetermined, or dynamically established each time the action plan is performed. If a dynamically determined sequence is needed, perhaps even selecting which actions are to be performed at all, users can provide custom subclasses for an action plan and an execution object that performs it. A builtin option is to generate an entirely random sequence each time a plan is executed.

Each time an action plan is being processed, a special kind of object called an activity is created entirely automatically by the runtime processing machinery. These activity objects implement the internal machinery of a virtual processor which has the ability to execute action plans. To get any activity started on a model, a processor is first initialized to run a single top-level plan. All other activity during the lifetime of a model must occur as a result of actions initiated by this plan (which may include the startup of other plans).

Because the activity objects are internal to the processing machinery, an application can usually just ignore their existence. They come and go dynamically as various action plans are started and completed, all arranged in a stack or tree of current activities controlled by a single top-level processor. The activity objects are potentiall useful, however, to obtain information about the context in which an action is currently running, or to build debugging or tracing tools to understand actions being performed.

One of the kinds of context information available from an activity object is the current time of a clock value incremented as a schedule is processed. A schedule is a kind of action plan in which all actions occur at specific points in time explicitly established within the schedule. As a schedule is executed, the activity object keeps a current time clock, which holds a global time from the start of all model execution, regardless of the time when the schedule itself was started. An activity object that processes a schedule is called a timeline activity, because its time continually increases from a global base time regardless of times contained in the schedule.

New activities are created whenever an action plan being processed contains an action to perform another action plan, or to start another action plan for autonomous execution. If one action plan performs another, its own processing is stopped until a new activity processing the other action plan completes. If one action plan starts another, the new action plan is started as an autonomous activity controlled only by a higher-level, containing activity.

A Swarm is an activity that exists only to control and coordinate other started subactiviti Unless the subactivities have some special form of explicit synchronization, none of their internal actions has any required ordering relative to those of other plans except as explicitly established during activity execution. The swarm can serve as a simple container of started subactivities which only occasionally synchronize for messages they send to each other. The swarm can be used to hold collections of objects as well as its subactivities as needed to help them coordinate with one another.

One special form of synchronization within a swarm is built into the virtual processing machinery. This synchronization interleaves the actions that occur for every successive time value during processing of timeline subactivities. This merging of actions is often relied on to interleave display and analysis processing with the scheduled actions of a base model. Since no other mechanisms for subactivity coordination are implemented in the current version of Swarm, synchronizing subschedule activities is the major current role of the Swarm activity type. In later versions of Swarm, a swarm will also serve as an important means for organizing a large models into clusters of more densely interacting components, and will also provide a basis of decomposition for parallel execution.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Activity Library: Advanced Usage Guide

(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Activity Library: Interface Reference

Dependencies

Following are the other header files imported by <activity.h>:

```
#import <collections.h>
```

The activity library follows the Swarm library interface conventions
(../defobj/library.html)
of the defobj library
(../defobj.html)
. The activity library relies heavily on the basic collection types defined in the collections library
(../collections.html)
, and the collection interfaces are an integral part of the interfaces defined within this library. Initialization of the activity library is done automatically by the containing Swarm libraries, and automatically initializes the collections and defobj libraries as well.

List of Reference Pages

- ProcessType
(ProcessType.html)
- ActionType
(ActionType.html)
- CompoundAction
(CompoundAction.html)
- ActionGroup
(ActionGroup.html)
- Schedule
(Schedule.html)
- SwarmProcess
(SwarmProcess.html)

- Action Types
(Action.html)
- Activity Types
(Activity.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

ProcessType – specification of a process

Synopsis

```
//
// ProcessType -- specification of a process
//
@deftype ProcessType
@end
```

Description

(.. For now, ProcessType is just a placeholder for future capability. ..)

ProcessType will eventually define support for parameterization of all processes. A process is a uniquely identified course of events that conforms to a specification of external and internal behavior.

Processes include both objects and executable actions. Objects change their state and behavior based on a sequence of externally initiated actions. An action, in contrast, typically has no external behavior other than to be executed in its entirety as a unit (yielding either an observable result or some change of state in a surrounding environment).

Parameterization will be based a uniform framework that defines all input parameters, internal state variables, and/or final results, and that provides for binding of these data in a context of execution. This parameterization framework is still being established but ProcessType objects will eventually record the specification of any process, to the extent that this specification is defined.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

ActionType – specification of an executable process

Synopsis

```
//
// ActionType -- specification of an executable process
//
@deftype ActionType <ProcessType>- activateIn: swarmContext;
@end
```

Description

An action type is a type of process that may be initiated as a unit of execution by an external request. A typical action has a well-defined duration determined by a fixed set of actions that execute within it. Externally initiated interaction typically occurs only at the start or end of the overall process. A typical action is executed in its entirety once an external request that initiates it has occurred. Some actions may also have internal events that cannot begin or complete until other actions from a containing environment have also begun or completed their execution. Such ordering constraints can be defined either within an action type or as part of a dynamic context of execution.

Executable actions include both actions compiled in a host language (such as C functions or Objective C messages) and compound actions built at runtime for interpretation by the Swarm abstract machine.

(.. For now, the only subtype of ActionType is CompoundAction. Types for compiled actions such as functions and messages have not been defined yet. ..)

Inherited behavior

```
@deftype ActionType <ProcessType>
```

ActionType inherits standard parameterization support from ProcessType. (.. Currently, no such support is defined. ..)

ActionType behavior

```
- activateIn: swarmContext;
```

The activateIn: message is used to initialize a process for executing the actions of an ActionType. This process is controlled by an object called an Activity. The activateIn message initializes an activity to run under the execution context passed as the swarmContext argument, and return the activity object just created. If the execution context is nil, an activity is returned that allows complete execution control by the caller. Otherwise, the execution context must be either an instance of SwarmProcess or SwarmActivity. (These objects are always maintained in one-to-one association with each other, either one of the pair is equivalent to the other as a swarmContext argument.)

If a top-level activity is created (swarmContext is nil), the created activity may be processed using activity processing commands such as run, step, etc. If an activity is created to run under a swarm context, the swarm itself has responsibility for advancing the subactivity according to its requirements for synchronization and control among all its activities. Activating a plan for execution under a swarm turns over control to the swarm to execute the subactivity as a more-or-less autonomous activity.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

CompoundAction – collection of actions to be performed in some order

Synopsis

```
//
// CompoundAction --
//   a collection of actions to be performed in any order consistent with a
//   set of ordering constraints
//
@deftype CompoundAction <;ActionType, Collection>CREATING
- (void) setDefaultOrder: aSymbol;
- (void) setAutoDrop: (BOOL)autoDrop;
USING
- getDefaultOrder;
- (BOOL) getAutoDrop;
@end
```

Description

An compound action is the supertype of ActionGroup and Schedule. A compound action defines an executable process that is composed from the execution of a set of actions in some defined order.

Inherited behavior

```
@deftype CompoundAction <ActionType>
```

CompoundAction is not directly creatable. One of its subtypes must be created instead. ActionPlan inherits the basic ability to be activated for execution from ActionType
(ActionType.html)
.

Create-time options

```
CREATING
- (void) setDefaultOrder: aSymbol;
- (void) setAutoDrop: (BOOL)autoDrop;
USING
- getDefaultOrder;
- (BOOL) getAutoDrop;
```

DefaultOrder

The DefaultOrder option indicates the ordering to be assumed among actions of the plan when no other explicit ordering has been assigned. Beyond this initial ordering, additional ordering constraints can be added selectively using partial order specifications on individual actions. (.. Partial order order constraints are not yet implemented.)

The value for DefaultOrder is a symbol that may have one of the following values:

```
extern id <Symbol>  Concurrent, Sequential, Randomized;
```

The Concurrent value of the DefaultOrder option indicates that can actions be run in any order (including at the same time, if hardware and software to do this is available) without no impact on the net outcome of the actions. The claim that action results are independent of their execution order gives the processing machinery explicit leeway to execute them in any order it chooses. In the current implementation on a single, serial processor, actions are always processed sequentially even if marked concurrent, because that is the only way they can be. In future versions, however, special runtime processing modes may be defined even for a serial processor, which would mix up execution order just to confirm the independence of model results.

The Sequential value for the DefaultOrder option is the default. It specifies that the actions must always be executed in the same order as they occur in the plan. This order is ordinarily the same order in which actions are first created in the plan, unless actions are explicitly added elsewhere the collection that underlies a plan. This option is always the safest to assure predictability of results, but it excludes the ability to run the actions in parallel. To better understand and document a model design, it is worth annotating action plans with an explicit indication as to whether they do or do not depend on a Sequential order.

The Randomized value for the DefaultOrder option specifies that the model results do depend on execution order, but that the order in which the actions were created or added has no special significance. Instead, the method of dealing with order dependence is to generate a random order each time a collection of same-time actions is processed. The random order will be generated from an random number generator internal to the processing machinery.

AutoDrop

The AutoDrop option specifies that as soon as any action been processed by a running activity, the action is removed from the plan and dropped so that it will never appear again. This option is useful for plans that are created for a one-time use, never to be used again. This option is especially useful for a dynamic schedule that continually receives new actions to be executed at future times, but will never repeat actions that were previously scheduled. Depending on the underlying implementation of the schedule, making sure the old actions get dropped using AutoDrop can considerably improve the performance of the schedule.

When an option like AutoDrop is used, or whenever the contents of an action plan undergo a significant amount of dynamic update, the action plan would ordinarily be intended only for a single point of use. If AutoDrop is specified, a restriction against multiple active references is enforced. An error will be raised if two activities ever attempt to process a plan with AutoDrop enabled at the same time.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

ActionGroup – collection of actions under partial order constraints

---

Synopsis

```
//
// ActionCreating -- protocol shared by ActionGroup and Schedule
//
@deftype ActionCreating
- createAction: anActionType;

- createActionCall: (func_t)fptr;
- createActionCall: (func_t)fptr : arg1;
- createActionCall: (func_t)fptr : arg1 : arg2;
- createActionCall: (func_t)fptr : arg1 : arg2 : arg3;

- createActionTo: target message: (SEL)aSel;
- createActionTo: target message: (SEL)aSel : arg1;
- createActionTo: target message: (SEL)aSel : arg1 : arg2;
- createActionTo: target message: (SEL)aSel : arg1 : arg2 : arg3;

- createActionForEach: target message: (SEL)aSel;
- createActionForEach: target message: (SEL)aSel : arg1;
- createActionForEach: target message: (SEL)aSel : arg1 : arg2;
- createActionForEach: target message: (SEL)aSel : arg1 : arg2 : arg3;
@end

//
// ActionGroup --
//   a collection of actions under total or partial order constraints
//
@deftype ActionGroup <CompoundAction, ActionCreating, OrderedSet, CREAT-
ABLE>@end
```

Description

An action group is an action plan whose basic representation is a sequence of actions that have been created within it. Even though a sequence is used to represent the actions of the collection, this does not necessarily mean that the execution machinery will process them in this order, since the permissible order of execution is still controlled by the DefaultOrder option inherited from ActionPlan. This initial order may also be further constrained by partial order constraints among the actions. (.. Partial order constraints are currently unimplemented.)

Inherited behavior

```
@deftype ActionGroup <CompoundAction, ActionCreating, OrderedSet, CREATABLE>
```

ActionCreating defines the createAction messages for ActionGroup just so that this protocol may be shared with Schedule, where they provide a convenience interface for the creation of actions in the schedule at time zero. These messages will be documented here in ActionGroup as if they had been defined on ActionGroup itself.

An action group inherits its underlying representation from the OrderedSet
(../collections/OrderedSet.html)
type of the collections
(../collections.html)
library. All the members of the ordered set must consist only of actions that are created by one of the createAction messages defined on ActionGroup itself. Once the actions are created, they may be accessed or traversed using standard messages of the OrderedSet type. Partial orders on these actions are also established by inherited messages. (.. Currently, partial orders are not implemented on either collections or action plans, but the messages inherited from OrderedSet will be the basic method for maintaining them once implemented.)

ActionGroup behavior

Actions are created within the ActionGroup using one of the createAction messages. A createActionTo message specifies that the action to be performed is defined by an Objective C message selector to be performed on a receiving object plus any required arguments. The message selector is specified by the message: argument and the receiving object is specified as the first argument, target.

Given a variable aTurtle already initialized to a valid object, following is an example of a createActionTo message:

```
[createActionTo: aTurtle message: M(print)];
```

M() is a macro defined in defobj.h which is just a convenient shorthand for the Objective C expression @selector(message-name). As in any selector, the message name must include all parts of the message including colons and subsequent argument portions. An example of a message with one additional argument is the following:

```
[createActionTo: aTurtle message: $m(move:) : (id)10];
```

The createAction messages declare all arguments of the message to be of object id type, but you are free to cast other pointers and values up to the limits defined by the global portability assumptions
(../defobj/globalport.html)
. These is not portable across all machine architectures, but is expected to be portable across the 32-bit and 64-bit architectures on which Swarm will be supported. The message you send must still be declared to receive the type of argument you actually pass, before you cast it to the id type.

(.. Alternative approaches to argument typing are currently in development, but these will supplement rather than replace the current forms of createAction messages.)

The createAction: message specifoes that processing of another action type is to be performed by the action. The referenced action type is performed in its entirety, from start to finish, as the effect of the single created action.

The createActionCall: messages are similar to the createActionTo messages, except they specify the action to be performed as a binding of a C function to a list of argument values. The correct number of arguments for the function pointer passed as the initial argument must be supplied.

The createActionForEach: messages define a message to be sent in the same way as the createActionTo messages, but they assume that the object passed as the target argument is a collection object. They specify that each object available from that collection, using the standard messages of the Collection
(../collections/Collection.html)
type in the collections library, is to receive the specified message.

The createAction: messages specify that processing of another action plan is to be performed by the action. The referenced action plan is performed in its entirety, from start to finish, as the effect of the single created action. If parameters are defined on the action plan, arguments for the action can also be given. (.. Currently this is not supported, so only the single argument form of createAction: should be used.)

Each of the createAction messages returns the action object which it creates. Each different kind of createAction message returns a different type of Action object with a matching name. These Action objects provide access to all the information with which the Action was initialized. The complete set of Action object types is defined below together with the messages that may be used to access their contents. (.. The implementation of the Action objects is currently undergoing change as the responsibility for parameter and return value typing gets taken over by ActionType in defobj.)

The action objects are an integral, controlled component of the action plan in which they are created. If they are removed from the action plan collection using a remove message, the only collection in which they may be reinserted is the same collection from which they came. It is permissible, however, to modify the base representation sequence by removing from one position and reinserting at another.

The contents of the action objects would ordinarily be referenced only by general-purpose tools to display or analyze an action plan structure. The action object id, however, can also be used to establish additional ordering constraints on any of the actions in a plan. These constraints supplement any initial order constraints established by the DefaultOrder option. Constraints may be established not only within the same plan, but between actions in different plans. (.. Partial order constraints are not yet implemented.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

Schedule – collection of actions ordered by time values

Synopsis

```
//
// Schedule -- collection of actions ordered by time values
//
@deftype Schedule <CompoundAction, ActionCreating, Map, CREATABLE>CREATING
- create: aZone setRelativeTime: (BOOL)relativeTime;
- create: aZone setRepeatInterval: (timeval_t)repeatInterval;
SETTING
- (void) setConcurrentGroupType: groupType;
- (void) setSingletonGroups: (BOOL)singletonGroups;

- (void) setRelativeTime: (BOOL)relativeTime;
- (void) setRepeatInterval: (timeval_t)repeatInterval;
USING
- getConcurrentGroupType;
- (BOOL) getSingletonGroups;

- (BOOL) getRelativeTime;
- (timeval_t) getRepeatInterval;

- at: (timeval_t)tVal createAction: anActionType;

- at: (timeval_t)tVal createActionCall: (func_t)fptr;
- at: (timeval_t)tVal createActionCall: (func_t)fptr:arg1;
- at: (timeval_t)tVal createActionCall: (func_t)fptr:arg1:arg2;
- at: (timeval_t)tVal createActionCall: (func_t)fptr:arg1:arg2:arg3;

- at: (timeval_t)tVal createActionTo: target message: (SEL)aSel;
- at: (timeval_t)tVal createActionTo: target message: (SEL)aSel:arg1;
- at: (timeval_t)tVal createActionTo: target message: (SEL)aSel:arg1:arg2;
-
at: (timeval_t)tVal createActionTo: target message: (SEL)aSel:arg1:arg2:arg3;

- at: (timeval_t)tVal createActionForEach: target message: (SEL)aSel;
- at: (timeval_t)tVal createActionForEach: target message: (SEL)aSel:arg1;
-
at: (timeval_t)tVal createActionForEach: target message: (SEL)aSel:arg1:arg2;
- at: (timeval_t)tVal createActionForEach: target message: (SEL)aSel:
                                                              arg1:arg2:arg3;
@end
```

Description

A schedule is compound action whose basic representation is a sorted Map of actions that have been created within it. The key value associated with each of these actions is an unsigned integer value for which the typedef timeval_t

is supplied.

Inherited behavior

@deftype Schedule <CompoundAction, ActionCreating, Map, CREATABLE>
A schedule inherits its underlying representation from the Map
(../collections/Map.html)
type of the collections
(../collections.html)
library. All the members of the ordered set must consist only of actions that are created by one of the createAction messages defined on Schedule itself. Once the actions are created, they may be accessed or traversed using standard messages of the Map type. The key values of this collection, however, must be cast to and from the id type defined for key values by the Map type. Partial orders on these actions can also established by messages inherited from Map. (.. Currently, partial orders are not implemented on either collections or action plans, but the inherited messages will be the basic method for maintaining them once implemented.)

Create-time options

RelativeTime

The RelativeTime option specifies that all the times in the schedule are relative to the time when processing of the entire schedule begins. Otherwise, the times are assumed to be absolute times, with their base in the starting time of the entire model.

(.. This option is currently supported only on schedules, not swarms.)

RepeatInterval

The RepeatInterval option specifies that as soon as all actions in the schedule have completed, it is to be rescheduled at a time computed as the time at which the schedule was last started, plus the value specified as the RepeatInterval argument. This option overrides the normal default that times are considered absolute, as if the RelativeTime option had also been specified at the same time. All scheduled times must be less than the specified repeat interval, or an error will be raised. The RepeatInterval option can continue to be reassigned to different values after a schedule has been created, but the interval value must always be greater than the scheduled times of any actions which it contains.

(.. This option is currently supported only on schedule, not swarms.)

ConcurrentGroupType

The ConcurrentGroupType option is used to control handling of multiple actions which end up being scheduled at the same time value. As far the schedule representation is concerned, these actions are assumed by default to be concurrently executable, and the processing machinery is free to process them as such if no ConcurrentGroupType option is specified.

If a different interpretation of actions at the same time step is needed, the ConcurrentGro option may be specified. The argument of this option must be an object that when given a standard create: message will return an object having all the structure of a standard ActionGroup object. For example, if you want to specify that actions at the same time step are to be processed in random order, you could use the following procedure:

```
concurrentGroupType = [ActionGroup customizeBegin: aZone];
  [concurrentGroupType setDefaultOrder: Random];
  concurrentGroupType = [concurrentGroupType customizeEnd];

  aSchedule = [Schedule createBegin: aZone];
  [aSchedule setConcurrentGroupType: concurrentGroupType];
  aSchedule = [aSchedule createEnd];
```

Note that instead of using the normal createBegin/End sequence to create an instance, the concurrent is initialized using the customizeBegin/End sequence to customize a type without creating an instance. For more information, see the defobj library documentation.

In addition to overriding the standard ActionGroup type, the concurrent group type may be implemented by a custom subclass of the ActionGroup implementation which you supply yourself. A custom subclass is free to implement custom rules for how to combine all the actions which happen to arrive at the same time value. For example, it could decide that some actions are not to be executed at all, or it could create entirely new actions to be executed instead of or in addition to those which were originally scheduled. (.. Specific rules for writing a custom ActionGroup subclass will be documented at a future time, but all the apparatus to do so is present today. A concurrent action group is currently used to maintain the proper order of execution among subswarms of an owner swarm.)

SingletonGroups

The SingletonGroups option indicates that an action group should be created for every time value which is present, even when only a single action is present at the time value.

Ordinarily, a concurrent action group is created to process actions at the same timestep only if more than one action is scheduled at that timestep. The overhead of these action groups is relatively low, because it just creates a single new object to which actions are directly linked, but it is still faster to avoid creating them if only one action is present at a timestep. If a custom subclass is being provided however, it may need to examine the actions at a timestep even if there is only one.

---

timeval˙t – type for clock values

The following typedef is provided for convenience in declaring time values:

```
typedef unsigned   timeval_t;        // type for time values
```

Values of this type are used as keys when inserting actions into a schedule at a particular time, or for querying the current time value of a swarm or schedule during its execution. (The shorter name time_t has already been taken by one the standard C libraries.)

For very long running models using finely divided units of time, it is quite possible that a 32-bit unsigned time value could overflow. Special messages are provided in the execution machinery to reset all times to a different base value if a danger of overflow exists. If this machinery is exercised, all times of all referenced action plans are reset to a different base value in unison, which makes the shift as transparent as possible. Separate support is also available to declare time values that are subunits of the discrete clock values in a containing schedule or swarm. (.. Currently, all this extended time unit support is unsupported, though there is an example of scheduling at subunit times in a GridTurtle
(../grid.html)
test program.)

Schedule behavior

The messages to create actions within a schedule are essentially the same as those for ActionGroup

(ActionGroup.html)
, except for the presence of an initial at: argument indicating the time at which an action is to be performed. Except for the time associated with each action, meaning of the createAction messages is the same as for ActionGroup.

When multiple actions are all scheduled at the same time, they are all inserted into a concurrent action group created for that time value. The ConcurrentGroupType option may be used to override the default action group for these concurrent actions by a custom user-defined subclass. (.. Details of doing this are not yet documented, but there are examples.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

SwarmProcess – object that encapsulates a collection of processes

---

Synopsis

```
//
// SwarmProcess --
an object that holds a collection of concurrent subprocesses
//
@deftype SwarmProcess <ActionType, Zone, CREATABLE>CREATING
- setInternalZoneType: aZoneType;
- (void) setSynchronizationType: aScheduleType;

- (void) setInternalTimeMultiplier: (timeval_t)multiplier;
USING
- getSynchronizationType;
- (timeval_t) getInternalTimeMultiplier;

- getInternalZone;
- getActivity;
@end

//
// ActivationOrder -- default concurrent group type for swarm
//
@deftype ActivationOrder <ActionGroup, CREATABLE>@end
```

Description

A swarm is an object created to coordinate the processing performed on other action types. Just like any other action type, a swarm may be activated for running as a subactivity of another swarm. A swarm contains no actions of its own, since its only purpose is to coordinate the processing of other activities.

SwarmProcess is the type of an object that can be activated to run in some context, and which itself then defines a context in which other action plans can be activated. Subtypes of this generic type can each define a custom interface to activities running within the swarm. The generic swarm type defines only access to an underlying activity that keeps track of the activated or running subactivities, along with an option that may be set to a collection of objects currently being processed by the swarm.

If subactivities running within a swarm are indexed by time (running either schedules or other swarms), the default coordination performed by the swarm is to merge all the different scheduled actions so that the actions of one time value are all performed before the actions of any later time value. If the actions are not indexed by time, or if an subactivity has been released from strict time synchronization with other subactivities in the swarm, the default coordination is limited to partial order constraint among the action plans being run. (.. Currently only the merging of schedules and subswarms by time values is supported.)

A public superclass called Swarm is supplied in the objectbase library
(../objectbase.html)
for use in implementing custom swarm classes. Such a class can override the method for the activateIn: message of ActionType
(ActionType.html)
to automatically start subactivities to run within a swarm when the swarm is first activated.

Inherited behavior

```
@deftype SwarmProcess <ActionType, Zone, CREATABLE>
```

SwarmProcess inherits the messages of both ActionType
(ActionType.html)
and Zone
(../defobj/Zone.html)
. Inheritance of zone behavior means that a swarm can be used as the argument of a create: or createBegin: message, for creation of an object within the internal zone of a swarm.

The default value for the SynchronizationType option is not a generic action group, but a special predefined subtype of ActionGroup called ActivationOrder. This concurrent group type guarantees that actions scheduled to occur at the same time from different action plans running in the same swarm are executed in the same order in which the action plans were first activated.

Create-time options

```
CREATING
- setInternalZoneType: aZoneType;
- (void) setSynchronizationType: aScheduleType;
- (void) setInternalTimeMultiplier: (timeval_t)multiplier;
USING
- getSynchronizationType;
- (timeval_t) getInternalTimeMultiplier;
```

InternalZoneType

The InternalZoneType option sets the type of zone which is created by the swarm to hold objects created within the swarm. If set to nil, no internal zone is created within the swarm, and all use of the swarm as if it were a zone raises an error. The default of this option is standard Zone type. (.. Since there is no other Zone type yet, there's no reason to set this option yet except to turn off the internal zone. ..)

SynchronizationType

Synchronization type sets the type of schedule which is used internally by the swarm to synchronize subschedules. Its default is a schedule with a concurrent group of ActivationOrder.

InternalTimeMultiplier

(.. This option will be used in the future to support an internal time unit within the swarm that can be more fine-grained than the context in which the swarm is included. ..)

SwarmProcess behavior

```
- getInternalZone;
- getActivity;
```

getInternalZone returns a Zone object that is used by the swarm to hold its internal objects. Even though the swarm itself inherits from Zone and can be used as a Zone for nearly all purposes, this message is also provided so that the zone itself can be obtained independent of all zone behavior.

getActivity returns the activity which is currently running of subactivities within the swarm. This activity is the same as the value returned by activateIn: when the swarm was first activated. It returns nil if the swarm has not yet been activated.

Unlike other action types, swarms and swarm activities always exist in a one-to-one relationship, provided that the swarm has been activated. This restriction to a single activity enables the swarm to do double-duty as a custom object that provides its own interface to the activities running within the swarm.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

Action – element of a compound action

Synopsis

```
//
// Action -- messages shared by ActionCall, ActionTo, and ActionForEach
//
@deftype Action <GetOwner>- (int) getNArgs;
- (void) setArg1: arg1;
- getArg1;
- (void) setArg2: arg2;
- getArg2;
- (void) setArg3: arg3;
- getArg3;
@end

//
// ActionCall -- an action defined by calling a C function
//
@deftype ActionCall <Action>- (void) setFunctionPointer: (func_t)fptr;
- (func_t) getFunctionPointer;
@end

//
// ActionTo -- an action defined by sending an Objective C message
//
@deftype ActionTo <Action>- (void) setTarget: target;
- getTarget;
- (void) setMessageSelector: (SEL)aSel;
- (SEL) getMessageSelector;
@end

//
// ActionForEach --
an action defined by sending to every member of a collection
//
@deftype ActionForEach <ActionTo>@end
```

Description

Action is a common supertype of all action types which may be created within an action plan. Each action is always controlled by a single action plan to which it belongs. This action plan is referred to as its owner. Given the action object, its owner plan may be obtained using the inherited getOwner message.

Actions are allocated in the same zone as their owner plan, and may be created only using one of the createAction messages on an ActionGroup or Schedule. Each of these messages returns the action that was created as its return value. Actions in an action plan may also be obtained by processing the plan using any of its messages inherited from its underlying collection. Actions may also be removed from an action plan using a remove message on the underlying collection.

(.. Note: currently, an action cannot be removed while it is currently being executed. This means that the function or message called by an action cannot itself remove that same action from its action plan. This restriction will be removed in the future.)
No matter what the type of action, the default ordering constraints established by its membership in an owning plan may be supplemented by explicit specification. These specifications are made directly between actions in the same plan, or between actions in different plans if they are suitably qualified. Given any action belonging to any plan, the set of specific explicit ordering constraints in which it participates may be obtained by one of the messages getSuccessors, getPredecessors, getCostarts, and Coends inherited from the KeyedCollection
(../collections/KeyedCltn.html)
type.

Separate subtypes of Action are defined for each of the various forms of createAction messages that create them. The current representation of these actions will be undergoing change as support for their parameter and return types is migrated into more basic support from the defobj
(../defobj.html)
library. Each action type provides messages to retrieve and set the values of all argument values bound into the action. These capabilities will remain, but different messages will eventually be supported. This documentation will be completed once the messages supported on Action types are finalized.

Roger Burkhart <rmb@santafe.edu>
(mailto:rmb@santafe.edu)

Last modified: 1997-12-19 (mgd)

Activity Types

Contents

The following object types are documented below:

- Activity
- GroupActivity
- ForEachActivity
- ScheduleActivity
- SwarmActivity

Name

Activity – state of processing within an action plan

Synopsis

```
//
// Activity -- state of processing within an action plan
//
@deftype Activity <DefinedObject, Drop>- run;
-               stop;
- (void) terminate;

- next;
- step;
- stepEntry;
- stepExit;

- getStatus;
- getHoldType;

- getAction;
- getActionType;
```

```
- (void) setOwnerActivity: ownerActivity;
- getOwnerActivity;

- getSubactivities;

- getControllingActivity;
- getTopLevelActivity;
- getSwarmActivity;
- getScheduleActivity;

- setSerialMode: (BOOL)serialMode;
- (BOOL) getSerialMode;

- getCurrentSubactivity;  // serial mode only
@end
```

Description

A object of type Activity implements the processing of actions within an action plan. Each subtype of action plan has a corresponding subtype of Activity that implements processing of its respective type of plan.

The Activity types are part of the virtual processing machinery of an action plan interpreter. All the important elements of this machinery, including their current state of processing, are exposed to aid in development of general-purpose tools such as debuggers. Except for applications that need to create and run their own reflective activities, direct use of activity types by a modeling application should be rare.

An new activity object is created by each activateIn: message sent to an action type. activateIn: initializes a new activity object and prepares it for processing, but does not itself execute any actions. Subsequent messages, such as run, must be used to initiate processing within an activity. If an activity is activated to run under a swarm context, the owning swarm itself controls all processing within the subactivity.

An Activity type holds a tree of currently running, or a potentially runnable, subactivitie Each activity object represents a state of processing within a single action plan. The structure is a tree because multiple activities could be running or eligible to run at the same time. This occurs whenever two activities are specified as concurrent, either because are being performed by concurrent actions in some other plan, or because they were started for autonomous execution under the same swarm. When parallel processing machinery is present, each activity could also be advancing its own state independent of those of any other activity.

The current implementation supports only a single serial processor. Under serial execution, only one activity may be active at one time. This means that the current state of processing may be represented by a single stack of current activities that traces a path to a single leaf of the runnable activity tree. When running in serial mode, messages are available to obtain the currently running leaf activity or useful context information such as the current time available from it.

Activity behavior

(.. Specific messages for activity objects are still subject to change as some of their underlying support migrates to the defobj

(../defobj.html)
library. For now, this section will document just the basic messages to start and control processing within an activity.)

```
- run;
-               stop;
- (void) terminate;
```

The run message continue processing of an activity from any state in which its execution has been suspended. An error is raised if the activity is already running. run returns either a Stopped or Completed status that the activity has when it is no longer eligible to be run further.

The stop message causes the a currently running tree of activities to stop further processing and return a Stopped status. terminate also stops a running tree of activities, but sets all activities within the tree to a status of Completed whenever the tree is next run. terminate may be used on either a running or stopped tree of activities. It is the standard way to terminate schedule that is endlessly repeating under the RepeatInterval option.

```
- next;
- step;
- stepEntry;
- stepExit;
```

The next executes actions within a single compound action while skipping over all processing of any complete action plans executed by those actions. The step message executes a single action within a tree of activities. The stepEntry message executes multiple actions within a single currently runnable activity, but stops the activity when a new level of action plan has just been started within the tree of activities. stepExit executes multiple actions in the same action plan until the activity for that action plan has completed, or cannot otherwise be run further. (.. Currently, stepEntry and stepExit are unimplemented.)

```
- getStatus;
- getHoldType;
```

The getStatus message returns one of the following codes for the current run status of a particular activity:

```
extern id <Symbol>  Initialized, Running, Stopped, Holding,
    Released, Terminated, Completed;
```

The getHoldType returns a code for the particular hold constraint under which an activity is currently holding. It returns nil if the basic status of the activity is not Holding. (.. Currently no hold constraints other than merging within a swarm are supported, and this message always returns nil.)

```
extern id <Symbol>  HoldStart, HoldEnd;
```

Activity context macros

The current running activity is always available for reference within any action running within the activity. (.. Currently, the getCurrentActivity() macro raises an error, but the other macros work as long as there is a currently running activity. ..) The following macros provide access to various information available from the current running activity and its owner activities:

```
#define  getCurrentActivity()
#define  getCurrentAction()
#define  getCurrentTime()
#define  getCurrentScheduleActivity()
#define  getCurrentSchedule()
#define  getCurrentSwarmActivity()
#define  getCurrentSwarm()
#define  getCurrentSwarmActivity()
#define  getTopLevelActivity()
```

Activity subtypes

(.. Only a synopsis of Activity subtypes is provided here for now. Note, however that access to a current time is available from a schedule or swarm activity, along with the ability to run a schedule until a particular time value is first reached.)

Name

GroupActivity – state of processing within an ActionGroup

Synopsis

```
//
// GroupActivity -- state of processing within an ActionGroup
//
@deftype GroupActivity <Activity>@end
```

Name

ForEachActivity – state of processing within an ForEach action

Synopsis

```
//
// ForEachActivity -- state of execution within a ForEach action
//
@deftype ForEachActivity <Activity>
- getCurrentMember;
@end
```

Name

ScheduleActivity – state of execution within a Schedule

Synopsis

```
//
// ScheduleActivity -- state of execution within a Schedule
//
@deftype ScheduleActivity <Activity>
- (timeval_t)   getCurrentTime;

- stepUntil: (timeval_t)tVal;  // unimplemented
@end
```

Name

SwarmActivity – a collection of started subactivities

Synopsis

```
//
// SwarmActivity -- a collection of started subactivities
//
@deftype SwarmActivity <ScheduleActivity>
@end
```

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Activity Library: Subclassing Reference

Subclassing is supported by the activity library as an integral technique for extending the framework it implements. There are two specific places where subclassing is expected as the normal technique of extension: definition of concurrent action groups, and definition of customized swarm objects. Concurrent action groups are not yet fully documented because their full interface is still being finalized. Swarm subclasses should inherit from the Swarm superclass defined in the objectbase (../objectbase.html) library; the activity library provides the underlying support packaged by this superclass.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Activity Library: Interface Design Notes

(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Activity Library: Implementation Notes

(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08 (mgd)