# 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 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

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.

---

# 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

# 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

*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. ..)

# 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)
.

---

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

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

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

Last modified: 1998-01-08

@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

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.)

# 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.
```

### 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

# 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)

```
- (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

```
        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

# 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: 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