(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: 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

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.

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 subactivities. 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: 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)

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

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

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

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 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.)

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;
```

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 ConcurrentGroupType 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 concurrentGroupType 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.

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

Last modified: 1998-01-08

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 constraints 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;
```

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
```

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;
```

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,
```

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
```

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: 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)