ACT-R Software Framework Specification v 1.1

Dan Bothell
Carnegie Mellon University
Psychology Department
db30@andrew.cmu.edu

Table of Contents

Page #

0. Introduction *4*

1. The Framework's Components *5*
- **1.1 Overview** *5*
- **1.2 Components of the Framework** *5*
  - ***1.2.1 meta-process*** *5*
  - ***1.2.2 scheduler*** *8*
  - ***1.2.3 events*** *9*
  - ***1.2.4 model*** *10*
  - ***1.2.5 device*** *10*
  - ***1.2.6 module*** *10*
  - ***1.2.7 buffer*** *11*
  - ***1.2.8 parameters*** *12*
  - ***1.2.8 chunks*** *12*
  - ***1.2.10 chunk-type*** *13*
  - ***1.2.11 chunk specification*** *13*

2. Implementation Assumptions *14*

3. API of the Framework *15*
- **3.1 General Commands** *15*
- **3.2 Running the Scheduler** *18*
- **3.3 Event Accessors** *21*
- **3.4 Generating Events** *22*
- **3.5 Event Hook Functions** *29*
- **3.6 Text Output** *30*
- **3.7 Multiple Meta-Processes** *30*
- **3.8 Models** *32*
- **3.9 Chunk-types** *33*
- **3.10 Chunks** *37*
- **3.11 Extending the Chunk Representation** *42*
- **3.12 Chunk specifications** *43*

3.13 Device *47*
3.13.1 Introduction *47*
3.13.2 Setup *48*
3.13.3 Basic Methods *48*
3.13.4 Icon Construction *50*
3.14 Buffers *51*
3.14.1 Buffer Parameters *59*
3.15 Modules *60*
3.16 Parameters *64*

4. Framework Modules *66*
4.1 Printing module *66*
4.2 Naming module *68*
4.3 Random module *70*

Appendix *72*
A1. Demonstration Declarative Module *72*
A1.1 Declarative Module Source Code *73*

0. Introduction

This document is a follow up to the previous proposal for ACT-R 6, which is available on the ACT-R website at http://act-r.psy.cmu.edu/act-r6. This is a more technical document dealing directly with the implementation of the software. It does not directly address any of the issues with the operation of the procedural or declarative systems. It is a description and API for what I am calling a "framework" for implementing ACT-R 6. The material in this document is in many ways not a “user level” document and would be the basis for two manuals - a “Modeler’s manual” and a “Module Writer’s manual”.

The foundation for the software framework is that the implementation of ACT-R can be broken into three main components. The first is the code that has basically nothing to do with the theory. That is things like time management and general event coordination (the scheduler) as well as high level "model management" issues. The next piece is the constructs from the ACT-R 5 theory which dictate the overall operation. Those constructs are modules, buffers and chunks. The final piece is the implementation of the details of the theory in the specific modules and buffers of the system.

The framework is the description and documentation of the first two components. The high level view is that the framework exists to support the pieces of the system. On its own, it does essentially nothing, but through the addition of meaningful components it becomes a useable system. By separating and documenting those pieces there will be a well defined interface provided through which all of the components must operate, whether those components are part of the theory as we provide it or additions made by others. It can operate as a black box onto which meaningful pieces are attached, and it is those pieces of the system that are attached which embody the theory. Once the framework is operational, it should be unnecessary to "look inside" of it to understand how the theory operates or to add meaningful components to it. In addition, changes or modifications to the internals of the framework should not affect how the connected components operate - the framework can be independently modified, optimized and tested without affecting the operation of the theory components.

The framework obviously is not a full specification of the ACT-R 6 implementation, and there are several issues that were introduced in the previous ACT-R 6 proposal which still need to be addressed (things like how productions will look and some of the details about buffers, declarative memory and activations). A big benefit of separating the framework from the theory's implementation is that it allows for development and testing of the framework to progress in parallel with those discussions about the theory components. That should provide more opportunities for evaluating and refining the API of the framework and allow for a more stable implementation for the initial ACT-R 6 release.

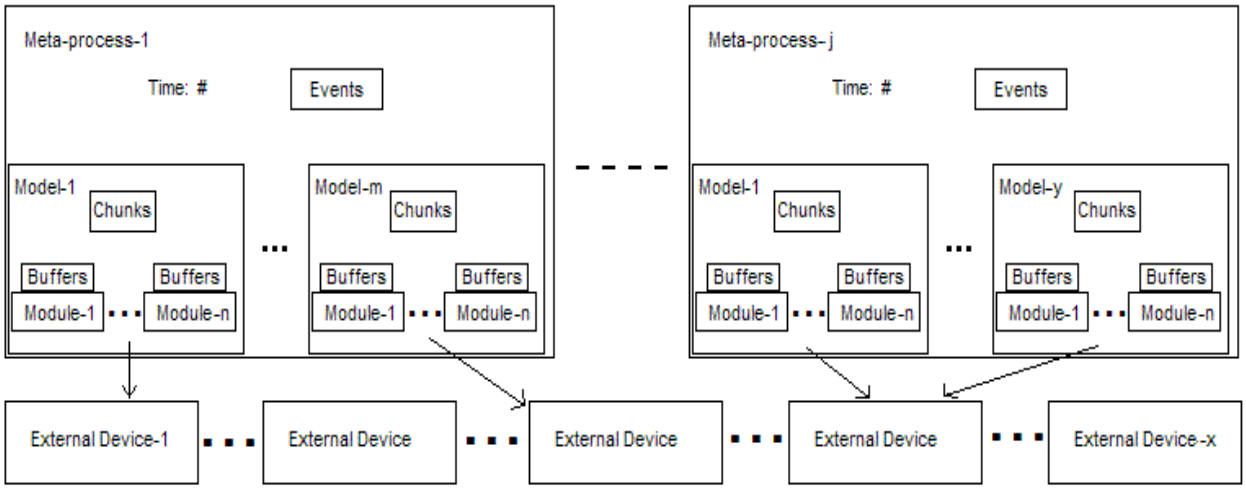
1. The Framework's Components

In many ways, the framework is a generic discrete event simulator with which one could implement any modeling or simulation system, but which contains components that are particularly important to ACT-R. In addition to the components of the framework which are derived from the theory (modules, buffers, and chunks) there are other constructs introduced which exist at the model management level. Those other constructs are the meta-process, events, a model, a device, and a parameter. This section will describe the organization of the components and provide an introduction to each of them.

1.1 Overview

The meta-process is the top level abstraction and basically each instantiation of a meta-process is a complete system. The meta-process maintains the time and events for a set of models. Each model consists of a set of chunks, an instance of each module and its corresponding buffer(s), the code which specifies the initial state of the model (declarative, procedural, parameters, etc), and an indication of an external device with which to interact if necessary. Essentially, it is the modules which define what a model can do and which will interact with that external device. That is where the theory and additions will be implemented - as modules.

Here is a diagram showing the relations between the components:



1.2 Components of the Framework

1.2.1 meta-process

The meta-process is a complete instantiation of the system. A meta-process holds a set of models, the current simulated time for those models, and the sequence of actions to perform generated by those models (which will be called the event queue). It provides the mechanisms for adding and removing

events from the event queue, running through the events on the queue, resetting the system (the time, event queue, and associated models), and displaying the trace and other messages.

This is not really a new piece of ACT-R relative to the previous versions. They have all had these same types of mechanisms and operations. The meta-process is based heavily upon the master-process of ACT-R/PM as used in ACT-R 5 as well as the general commands available in ACT-R 5 and earlier. It is essentially just grouping together and documenting those operations under a single name.

1.2.1.1 The current meta-process

All operations will be relative to the "current" meta-process. There may be at most one current meta-process. If there exists only one meta-process, then it will always be the current one. If there is more than one meta-process defined, then the user will be responsible for indicating which meta-process is the current one. Meta-processes will have a name when they are created, and that is how they will be referenced for setting and getting which one is the current meta-process.

1.2.1.2 The default meta-process

The framework will always have at least one meta-process which will be called the default meta-process and its name will be **default**. One will only need to create additional meta-processes if it is necessary to run multiple versions of the system concurrently with separate clocks (models which are asynchronous). That is probably not a situation most modelers will need, so it is likely that the existence of the meta-process abstraction can be largely ignored by the average user because the default meta-process will be sufficient for running any number of synchronous models.

1.2.1.3 The meta-process components

There are several attributes of a meta-process that may have significance for a user. However, because of the "black box" nature of the framework, the representation of a meta-process and its attributes are not specified as part of the API. Thus these attributes should only be accessed or changed through the commands that are provided as documented later.

time

The current time of the simulation. The initial setting is time 0. The time will be recorded in seconds, and the resolution will be one millisecond.

models

The set of models that exist within this meta-process. Initially it is an empty set.

event queue

The event queue is the set of events that have been scheduled to occur. Initially it is an empty set.

delayed events

This is a set of events that are not currently scheduled to occur, but will be scheduled if their conditions are met. The initial setting is an empty set.

pre-event hooks

This is a set of functions that will each be called prior to every event's processing. The initial setting is an empty set.

post-event hooks

This is a set of functions that will each be called after every event's processing. The initial setting is an empty set.

version number

This will be used for version control and bug tracking. Every update to the framework will increment the version number of the meta-process in some way, and those updates will be documented.

1.2.2 scheduler

The scheduler is essentially the component of the meta-process which executes the events and manages the advancement of time and not a specific component in and of itself. It is not dependent on the modules or models which make up the system other than the fact that they are the source of the events.

Here is some pseudo-Lisp code to describe the operation of the scheduler:

```
(while (events remain and no stopping condition has been met)
  (remove event with lowest time and highest priority from the queue)
  (set the clock to that time)
  (set the current model to the one in which that event was generated)

  (dolist (hook (a list of pre-event hook functions))
    (funcall hook (copy of the current event)))

  (when (event's output is set)
   (print a description of this event for the trace))

  (execute the event)

  (dolist (hook (a list of post-event hook functions))
    (funcall hook (copy of the current event)))

  (unset the current model if there is more than one model defined)

  (when (there are events waiting to be scheduled)

     (loop (dolist (waiting-event (the list of waiting events))
             (when (waiting condition of waiting-event satisfied)
                (remove waiting-event from the set of waiting events)
                (add waiting-event to the queue)))

           (when (or (no waiting-events left)
                     (no waiting-events were added to the queue))
             (return)))))
```

That will be used by a few different functions. The differences between them will be the stopping conditions.

While the scheduler is running it will print out a trace of the events that are executed. The trace is generated through the print module which will be described in detail later. Each event will be summarized in one line of text unless it is marked to not be displayed. The line of text will look like this:

Time {meta-process name} {model name} module details

The time is always the first item on the line. If there is more than one meta-process defined, then the second item will be the name of the meta-process in which this event exists. If there is only one meta-process defined then the meta-process name is omitted from the trace. Then the name of the event's model will be included if there is more than one model defined in that meta-process. If there is only one

model in that meta-process then the model name is omitted from the trace as well. The next item, which will always be included, is the name of the module which generated the event. The remainder of the line is then a description of the event. That will be either the printing of the event's action and parameters or text that was provided when the event was generated.

1.2.3 events

The events are generated through scheduling functions which are detailed later. An event consists of the following attributes:

time

the simulation time at which the event will occur in seconds.

priority

priority is used to order events that are scheduled to occur at the same time.

The priory can be any number or the keyword **:max** or **:min**. An event with priority **:max** is guaranteed to occur before any event with the same time and a priority other than **:max**. An event with priority **:min** will occur after any event with the same time and a priority other than :min. An event with a numerical priority will occur before any event with the same time and a numerical priority less than its own. Events with identical times and identical priorities may occur in any order.

action

a function that will be called when this event is executed.

module

the name of the module which created this event.

destination

if this action is one provided by a module or device then the destination can be used to name that module. The correct instance of that named module will be passed as the first parameter to the action if one is provided.

parameters

the list of parameters which will be passed to the action function when the event is executed.

details

a string that will be displayed in the trace for this event in place of the action and parameters.

As with the meta-process, the representation of events and the event queue are not specified as part of the framework's API, and thus events should never be created or manipulated directly. They must be generated and scheduled using the provided functions and the components of an event should only be retrieved using the accessors provided.

There is also a special type of event which can be scheduled called a break event, which may be useful for debugging. A break event has no action to execute, but when a break event occurs it will cause the scheduler to stop after handling the break event.

1.2.4 model

The model is a new abstraction for ACT-R 6, and it is basically just the encapsulation of an ACT-R model as they have existed in previous versions. The addition of this abstraction allows for the easy creation and running of multiple models within a single instantiation of the system. A model contains an instance of each module, all of the corresponding buffers, a set of chunks, and an arbitrary list of forms to evaluate (those forms are basically everything that would have followed clear-all in the older versions of ACT-R).

1.2.4.1 The current model

Along with the current meta-process, there is also a current model. The current model is relative to the current meta-process, and will always be a member of the current meta-process. As is the case for the current meta-process, as long as there is only one model in a meta-process it will always default to the current one. However, unlike the meta-process, there is not a default model that automatically exists in each meta-process. All models must be defined by the user.

1.2.5 device

The device is the abstract representation of the world. It defines the operations which modules can use for gathering information about the world and the operations available to the modules for manipulating that world. It is based on the current ACT-R/PM implementation of the device interface. In addition, for ACT-R 6 the device interface should provide mechanisms for multiple models to communicate, for example, if one model speaks other models connected to the same device should be able to hear that. That is probably going to require some extensions and additions be made to the current device interface specification, but has not been addressed at this point.

1.2.6 module

A module is a generalization of the module class from the ACT-R/PM implementation. It defines a meaningful subsystem which can be connected to the framework. Every model will have its own instance of each module (what an "instance" is depends on how the module is written and need not be an actual CLOS object) and it is the collection of modules which are attached to the framework that produces a meaningful system. In terms of the software framework, a module is more general than the description of a module from the theory because a module from the software framework's perspective may not have any psychological significance and exist only as a support mechanism for modeling.

By implementing all of the meaningful pieces of the system through a single interface (modules) it puts everything on the same level and gives no special treatment to any one piece. However, a module could assume a special role and implement a particular control structure that was meaningful to the system which is being implemented in the framework. For instance, a procedural module in ACT-R would coordinate the actions of all the other modules. It would still be implemented through the same module interface, but would essentially be the master module because its functioning would coordinate most of the functioning of the other modules.

There are essentially no restrictions on how a module is implemented, how it operates or what it does. The only thing necessary is that it be defined with a unique name relative to the other modules. However, there will need to be some guidelines to ensure that as a whole the system operates reasonably. The main guideline that is seen as necessary at this point is that a module use the meta-process's scheduler and clock for any activities that occur "over time". There are some additional guidelines included with the detailed description later, and others may be introduced as the framework is finalized. Eventually, there will be a "Module Implementer's Manual" which will clearly state those guidelines.

Although there are no hard constraints on how a module is implemented, there are two additional constructs available which can be used while creating a module to allow it to be tied into the system. The first is buffers, which can be used as an interface to a module that will be accessible from any other module, and the other is a parameter specification mechanism that allows the module's parameters to be set using the same mechanism as the other modules.

1.2.7 buffer

The buffers of ACT-R 5 had two main functions: they were a means of making a request of a module from a production and they where able to hold a chunk, which was typically put there as the result of a request to the corresponding module. In ACT-R 6 they are similar, but they are being extended to be a more general interface to the modules and their role as the source of chunks for declarative memory is being made concrete in the implementation.

There are now three types of requests that can be sent to a module through a buffer. The first type of request is the one that was used in ACT-R 5 – a request for the module to take some action which usually results in a new chunk being placed into the buffer. The second is a request for the module to immediately respond to a query about its internal "state". This is similar to what was done with the *-state buffers in ACT-R 5, but now will be done through the same buffer. The module should respond to those requests without changing the contents of the buffer itself. The third type of request is for the module to make a change to the chunk that is currently in the buffer. This operation is a modification of the mechanism that was provided in ACT-R 5 which allowed one to change how the RHS "=" operator worked on a buffer by buffer basis. The idea now is that all operations that a module is to perform will be made as explicit requests of the module through the buffer.

Buffers are not created explicitly, but are implicitly created with the definition of a module. A buffer is given a name, which must be unique among buffers i.e. two modules cannot have buffers of the same name. It is not necessary that a module use a buffer (or buffers), but by using buffers it will allow it to

be seamlessly used by other modules. In particular, for the implementation of ACT-R 6, using a buffer(s) will make the module automatically available for use in productions.

Any chunk may be placed into a buffer and there are no protections or guarantees built into the framework for the chunks in a buffer. Any module (or non-module code) is allowed to check, modify, or set the contents of any of the buffers at any time. Thus, one should not assume anything about the contents of a buffer when writing a module. Another of the guidelines for module creation is that a module should only modify its own buffer as a result of a request. However, a big exception to that will be the procedural module because the productions can be used to directly access and modify any buffer.

Finally, each buffer will automatically have a parameter associated with it, which can be used by the declarative memory module, that indicates the activation spread from that buffer. The default value for the parameter will be 0, but it can be set by the modeler or module writer to any value. This is one point where the theory dictates something that must be specified as part of the framework to provide a general mechanism instead of the buffer specific operation that exists in ACT-R 5. The idea is that every buffer is a potential source of activation instead of just the goal buffer. By having the default value for the spread be 0 the system will operate like the older ones, but now the modeler has the option of exploring other possibilities.

1.2.8 parameters

Because of the support for multiple models, having the parameter values global is not a practical situation because it may be desirable to have different models with different parameters. Since each model has a copy of each module and its buffers the plan is to have the parameters belong to the modules and buffers. In the framework, that will be accomplished by a module reporting the parameters it owns when it is instantiated within a model. Then, the only thing necessary to guarantee the independence of the parameters is that the modules need to keep the parameters encapsulated within an "instance" of the module. A single mechanism will be provided for setting and getting the parameters from the modules so that from the stand point of the user they will all be treated equally.

1.2.9 chunks

Before describing what a chunk is, I feel it is necessary to justify making them part of the framework. In old versions of ACT-R (4 and earlier) chunks were the storage units of declarative information, and that has not changed with ACT-R 5 and is still true in ACT-R 6 as well. In the implementation of the old versions every chunk, upon creation, was instantly a member of ACT-R's declarative memory. In ACT-R 5, with the addition of the buffers and modules that handled perceptual and motor information, the theory now says that chunks enter the declarative memory system from the buffers. The chunks that are in the buffers go to declarative memory when they are cleared from the buffers. However, the implementation of ACT-R 5 still operates as the old systems did i.e. chunks are instantly part of the declarative memory system upon creation and not as a result of having been in a buffer.

That has lead to some tricky situations in modeling with ACT-R 5, particularly with regards to the goal buffer. Since there is no longer a goal stack to maintain past goals they must be retrieved from declarative memory. The problem occurs because the current goal chunk is always a member of declarative memory and is likely to be very active due to activation spreading from itself. When

attempting to retrieve a past goal which is of the same type as the current goal, that can result in the need for some extra tests and possibly some additional flags in the goal chunk to avoid retrieval of that current goal, and making that work well can be difficult.

Because the theory says that buffers hold chunks it seems that a chunk should be part of the framework because modules will be required to process and handle them for use in buffers. The proposal is that the framework contain a very basic definition of a chunk which is sufficient for communicating between modules through buffers and that the definition of a chunk itself be made part of the API. In addition, to support the functionality that the declarative memory module will need, there will be a mechanism through which a module can be informed of any buffer's clearing so that it can do something with that chunk. There will also be a way for a module to extend the definition of a chunk so that it could include any other components that it might need to support its functioning. For example, the declarative memory module will need to include additional parameters to the chunks to record the activation related information, and other modules may want to do so as well.

So, here is the actual description of a chunk from the perspective of the framework. A chunk is a named collection of data whose elements are referenced by tags called slots. The name of a chunk must be unique within a model. The set of slots which a particular chunk has is determined by the chunk-type of that chunk. The chunk-type is specified when a chunk is created, and cannot be changed. Each slot can hold one value, which can be anything. There may be additional parameter values associated with all chunks regardless of chunk-type that are added and used by the modules.

1.2.10 chunk-type

Because chunks are part of the framework, chunk-types must also be included. A chunk-type is an abstract type for a chunk. A chunk-type specifies the slots that chunks of that chunk-type will have, any default values for those slots, and may include an inheritance from one other existing chunk-type. A chunk-type is referenced by a name, and the name must be unique among chunk-types within a model. A chunk-type that inherits from an existing chunk-type is called a subtype of that existing chunk-type and that existing chunk-type is called the parent of the new chunk-type. The subtype will have all of the slots of its parent (including their default values) and may also have additional slots or specify different default values for the slots inherited from its parent.

A new proposal for ACT-R 6 is that all chunk-types are implicitly a subtype of a chunk-type called chunk which has no slots. By making that change one will be able to test that a buffer holds a chunk without having to know what chunk-type it is. Being able to do that may be important for generalizing the production compilation mechanism.

1.2.11 chunk specification

A chunk specification is a mechanism for describing a chunk. Chunk specifications are what will be sent to a module through its buffer as a request. A chunk specification (or chunk-spec for short) is used to describe the chunk-type, slot values, and any modifiers or variables for the conditions of the slots in the request.

2. Implementation Assumptions

Before detailing the API of the framework, there are a few assumptions about how I see things being implemented which should be pointed out. The first is that the framework will be programmed in Lisp (a fairly obvious assumption, but noted for completeness). Also, the names of "things" (chunks, models, modules, etc.) will be symbols. There can be problems with that, but through the use of a naming module (which will be instantiated in each model like all modules) I think most of those can be handled, and the overall simplicity that it provides is hard to ignore. Along with that then comes the use of macros to alleviate the user (modeler) from having to quote names everywhere. Commands that would be likely to be used by modelers (as opposed to those that are there mostly for the module writers) and require names will have corresponding macro and functional forms. The macro will quote the arguments before passing them to the functional form, so that the user does not have to do so. Finally, the default operation of the framework will not create any new packages or place the components in any particular package. A mechanism will likely be built in which could be enabled to load everything into a specific package (probably called :act-r), but by default will not be used.

3. API of the Framework

Here are the commands which will make up the framework. They are grouped based on their purpose. There is no attempt to separate the "user" commands from those which are only likely to be used by module implementers. The plan is to create two additional documents which are derived from this one that more fully detail the components that are important for modelers and module implementers.

An example showing a very simplified version of the declarative memory module is included in the appendix to help demonstrate how this all ties together.

3.1 General Commands

The following commands are for general modeling use, and many look and act like similar commands from previous versions of ACT-R.

clear-all

```
(defun clear-all ())
```

This function restores the framework to its initial state. It removes all models and meta-processes other than the default meta-process, and all of the attributes of the default meta-process are set to the initial state.

It also records the value of the Lisp variable ***load-pathname**** (which will be bound if **clear-all** occurs at the top level in a file) so that the **reload** command can load that file.

It returns nil.

reset

```
(defun reset ())
```

If there is a current meta-process, then the time, event queue, and waiting queue of that meta-process are returned to initial values and every model in its set of models is reset. If there is not a current meta-process a warning is signaled.

When a model is reset the following actions are performed:

- that model is set to the current model
- all events generated by that model are removed from the event queue
- all chunk-types and chunks are removed from that model
- all of the model's buffers are emptied (similar to being cleared but no modules are notified about the clearing)
- the default chunk-types are created
- the default chunks are created

- all parameters are set to their default values
- all modules of the model are reset
- the forms of the model are evaluated in order

It returns the name of the meta-process that was reset or nil if there was no current meta-process.

reload

```
(defun reload ())
```

If a file was recorded by **clear-all**, this function calls load with that file as the parameter and the value of the if-does-not-exist keyword set to nil.

If there is no pathname recorded, then a warning is displayed and the keyword :none is returned. Otherwise, it returns the result of calling load on the recorded pathname, which will be **t** if the file was loaded, or **nil** if it did not exist.

sgp

```
(defmacro sgp (&rest parameters))
(defun sgp-fct (&optional parameters-list))
```

parameters is any number of parameter names or sequence of parameter names and values
parameters-list is a list of either parameter names or parameter names and values

sgp is used to set or get the value of the parameters from the modules of the current model.

If no parameters are provided, all of the current model's parameters are printed to ***standard-output*** organized by module.

If all of the elements of **parameters** or **parameters-list** are keywords, then it is a request for the values of the parameters named. Those parameters are printed to ***standard-output*** and the list of their values in the order requested is returned. If any of the names are not of valid parameters then a warning is displayed and the keyword :error is returned for that position in the list.

If there are elements of **parameters** or **parameters-list** which are not keywords, then it is assumed that this is an attempt to set parameters. If there are an even number of elements in **parameters** or **parameters-list** then they are assumed to be pairs of a parameter name and a parameter value. If the list is not even, then a warning is displayed and no parameters are set. Each of those parameter values will be passed to the corresponding parameter's owning module and all monitoring modules. The return value will be the current settings of those parameters in the order given (the values may or may not be the same as the values passed in to set them depending on the module).

*Because the test to determine that a call to **sgp** is a request for parameter values is that all the values passed in are keywords, a module should never have a parameter accept a keyword as a possible value because it will not be possible to set such a parameter value on its own.*

If there is no current model at the time of the call, then a warning is displayed and **nil** is returned.

[In the previous versions of ACT-R **sgp** stood for set global parameters. In ACT-R 6 that name does not really apply since the parameters are now specific to the modules and can also vary between different models if more than one is defined. However, since it is an important function and will still be used to set all of the parameters that it set previously changing the name does not seem like a useful move.]

buffers

```
(defun buffers ())
```

buffers returns a list of the names of all the buffers in the current model.

If there is no current model a warning is displayed and it returns **nil**.

mp-time

```
(defun mp-time ())
```

mp-time returns the current time of the current meta-process in seconds.

If there is no current meta-process, then a warning is displayed and **nil** is returned.

mp-models

```
(defun models ())
```

mp-models returns a list of the names of all the models defined in the current meta-process.

If there is no current meta-process it prints a warning and returns nil.

meta-process-names

```
(defun meta-process-names ())
```

meta-process-names returns a list of the names of all the existing meta-processes.

mp-show-queue

```
(defun mp-show-queue ())
```

mp-show-queue prints the events that are on the event queue of the current meta-process to ***standard-output*** in the order that they would be executed.

If there is no current meta-process a warning is displayed and **nil** is returned.

Otherwise it returns the length of the event queue.

mp-show-waiting

```
(defun mp-show-waiting ())
```

mp-show-waiting prints the events that are in the waiting queue of the current meta-process along with a description of the condition for which each needs to be added to the event queue to ***standard-output***.

If there is no current meta-process a warning is displayed and **nil** is returned.

Otherwise it returns the length of the waiting queue.

mp-print-versions

```
(defun mp-print-versions ())
```

mp-print-versions prints the version number of the framework and the name, version number, and documentation of each module which is currently defined to ***standard-output***.

It returns nil.

3.2 Running the Scheduler

The following commands specify the ways in which the scheduler can be advanced through time.

run

```
(defun run (run-time &key (real-time nil)))
```

run-time is a time in seconds
real-time is either **t** or **nil**

If there is a current meta-process this will run the scheduler for that meta-process with a stopping condition that the clock will be advanced by no more than **run-time** seconds. If **real-time** is **t**, then the clock will be advanced in step with actual time instead of simulating time as fast as possible.

After running the scheduler a line will be output in the trace indicating why the run stopped, which will be one of the following conditions: the requested amount of time has passed, there are no events left to process, or a break event occurred.

run returns three values.

If there is no current meta-process all three are nil.

If there is a current meta-process, then the first value is the amount of simulated time that passed during the call to **run**. The second is the number of events that were processed from the event queue, and the third will be **t** if the run terminated due to a break event or **nil** if it terminated for any other reason.

run-full-time

```
(defun run-full-time (run-time &key (real-time nil)))
```

run-time is a time in seconds
real-time is either **t** or **nil**

If there is a current meta-process this will run the scheduler for that meta-process with a stopping condition that the clock will be advanced by exactly **run-time** seconds unless a break event occurs. If **real-time** is **t**, then the clock will be advanced in step with actual time instead of simulating time as fast as possible.

After running the scheduler a line will be output in the trace indicating why the run stopped, which will be one of the following conditions: the requested amount of time has passed or a break event occurred.

run-full-time returns three values.

If there is no current meta-process all three are nil.

If there is a current meta-process, then the first value is the amount of simulated time that passed during the call to **run-full-time**. The second is the number of events that were processed from the event queue, and the third will be **t** if the run terminated due to a break event or **nil** if it terminated for any other reason.

run-until-time

```
(defun run-until-time (end-time &key (real-time nil)))
```

end-time is a time in seconds
real-time is either **t** or **nil**

If there is a current meta-process this will run the scheduler for that meta-process with a stopping condition that the clock will be advanced to the explicit **end-time** if the current time is not already greater than **end-time** and no break event occurs. If **real-time** is **t**, then the clock will be advanced in step with actual time instead of simulating time as fast as possible.

After running the scheduler a line will be output in the trace indicating why the run stopped, which will be one of the following conditions: the requested **end-time** has already been passed, the requested **end-time** has been reached, or a break event occurred.

run-until-time returns three values.

If there is no current meta-process all three are nil.

If there is a current meta-process, then the first value is the amount of simulated time that passed during the call to **run-until-time**. The second is the number of events that were processed from the event queue, and the third will be **t** if the run terminated due to a break event or **nil** if it terminated for any other reason.

run-n-events

```
(defun run-n-events (event-count &key (real-time nil)))
```

event-count is a positive integer
real-time is either **t** or **nil**

If there is a current meta-process this will run the scheduler for that meta-process with a stopping condition that at most **event-count** events are executed. If **real-time** is **t**, then the clock will be advanced in step with actual time instead of simulating time as fast as possible.

After running the scheduler a line will be output in the trace indicating why the run stopped, which will be one of the following conditions: there were no more events to execute, **event-count** events were processed, or a break event occurred.

run-n-events returns three values.

If there is no current meta-process all three are nil.

If there is a current meta-process, then the first value is the amount of simulated time that passed during the call to **run-n-events**. The second is the number of events that were processed from the event queue, and the third will be **t** if the run terminated due to a break event or **nil** if it terminated for any other reason.

run-step

```
(defun run-step ())
```

If there is a current meta-process **run-step** will execute events from the event queue one at a time displaying each before execution on ***standard-output*** and prompting the user to decide whether to process the event, delete it, or end the stepping (other features may be added over time). It will continue until the queue is empty, a break event is executed, or the user requests it to quit.

run-step returns three values.

If there is no current meta-process all three are nil.

If there is a current meta-process, then the first value is the amount of simulated time that passed during the call to **run-step**. The second is the number of events that were processed from the event queue (not

counting those that were deleted), and the third will be **t** if the run terminated due to a break event or **nil** if it terminated for any other reason.

3.3 Event Accessors

The following commands are the accesors for the components of an event. For all of the event accessors, the **event** parameter is an actual event which may have been returned by one of the scheduling commands or which was passed to one of the hook functions. For all of the accessors, if the **event** parameter is not a valid event then the accessor will print a warning and return **nil**.

Because a function with the name event-*something* is likely to conflict with existing functions in some Lisp and the default operation of the system is to not separately package the components, the names of the accessors begin with evt- instead of event-.

evt-time

```
(defun evt-time (event))
```

evt-time returns the time in seconds at which the event is scheduled to occur.

evt-priority

```
(defun evt-priority (event))
```

evt-priority returns the priority of the event.

evt-action

```
(defun evt-action (event))
```

evt-action returns the action function that will be executed for the event.

evt-model

```
(defun evt-model (event))
```

evt-model returns the name of the model in which this event was created.

evt-module

```
(defun evt-module (event))
```

evt-module returns the name of the module which created this event.

evt-destination

```
(defun evt-destination (event))
```

evt-destination returns the destination which was specified for the event.

evt-params

```
(defun evt-params (event))
```

evt-params returns the list of parameters which will be passed to the event's action function when the event is executed.

evt-details

```
(defun evt-details (event))
```

evt-details returns the details string that will be displayed in the trace for this event if it has one, or nil if it does not.

evt-print-trace

```
(defun evt-print-trace (event))
```

evt-print-trace returns the output value of this event, which indicates whether to print this event in the trace.

3.4 Generating Events

The following functions define the ways to generate and schedule events.

schedule-event

```
(defun schedule-event (time action
                       &key (module :none) (destination nil)
                            (priority 0)(params nil) (details nil)
                            (output t)))
```

time a time in seconds which should be a non-negative number
action a function or function-name
module a symbol that names the module which is scheduling the event
destination a symbol that names a module or the keyword :device. The current instance of that module or the current model's device will be passed as the first parameter to the action.
priority a number or the keywords :max or :min
params a list of parameters for the action function
details a string describing the event for the trace or **nil**
output either **t** or **nil**

schedule-event creates a new event using the supplied parameters for its corresponding attributes and the current model will be used for its model. It will then be added to the event queue of the current meta-process.

If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If any of the parameters are invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and nil is returned.

The scheduled event is returned when successfully created and scheduled.

schedule-event-relative

```
(defun schedule-event-relative (time-delay action
                                &key (module :none) (destination nil)
                                     (priority 0)(params nil) (details nil)
                                     (output t)))
```

time-delay a time in seconds which should be a non-negative number
action a function or function-name
module a symbol that names the module which is scheduling the event
destination a symbol that names a module or the keyword :device. The current instance of that module or the current model's device will be passed as the first parameter to the action.
priority a number or the keywords :max or :min
params a list of parameters for the action function
details a string describing the event for the trace or **nil**
output either **t** or **nil**

schedule-event-relative creates a new event with a time that is equal to the current time plus **time-delay** and using the other supplied parameters for its corresponding attributes and the current model for its model which is then added to the event queue of the current meta-process.

If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If any of the parameters are invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and nil is returned.

The scheduled event is returned when successfully created and scheduled.

schedule-event-after-module

```
(defun schedule-event-after-module (after-module action
                                    &key (module :none) (destination nil)
                                         (priority 0) (params nil)
                                         (details nil) (output t)
                                         (delay t)))
```

after-module a symbol that names a module
action a function or function-name
module a symbol that names the module which is scheduling the event
destination a symbol that names a module or the keyword :device. The current instance of that module or the current model's device will be passed as the first parameter to the action.
params a list of parameters for the action function
details a string describing the event for the trace or **nil**
output either **t** or **nil**
delay one of **t**, **nil**, or **:abort**.

schedule-event-after-module creates a new event using the supplied parameters for its corresponding attributes and the current model for its model.

If there is an event currently in the event queue with the module name of **after-module** and the same model as the current model then this new event is placed into the event queue at the time of the next such event (lowest time) with a priority of **:min**. If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If there is no event in the event queue that matches on both model and module then the value of **delay** determines what happens to the new event.

If **delay** is **t** then the new event is placed into the set of waiting events to be scheduled after an event which matches **after-module** and the current model.

If **delay** is **nil** then the new event is added to the event queue for immediate execution. Its time will be set to the current time and its priority will be **:max**.

If **delay** is **:abort** then the new event is discarded without being scheduled or placed onto the waiting queue.

schedule-event-after-module returns 2 values.

If there is no current model or current meta-process or any of the parameters are invalid, then no event is scheduled and both values are nil.

If an event is scheduled then the first value will be the event and the second value will be **t** if the event is in the waiting queue or **nil** if it is in the event queue.

If the event is aborted the first value will be **nil** and the second value will be **:abort**.

schedule-event-after-change

```
(defun schedule-event-after-change (action
                                    &key (module :none) (destination nil)
                                         (params nil) (details nil)
                                         (output t) (delay t)))
```

action a function or function-name
module a symbol that names the module which is scheduling the event
destination a symbol that names a module or the keyword :device. The current instance of that module or the current model's device will be passed as the first parameter to the action.
params a list of parameters for the action function
details a string describing the event for the trace or **nil**
output either **t** or **nil**
delay one of **t**, **nil**, or **:abort**.

schedule-event-after-change creates a new event using the supplied parameters for its corresponding attributes and the current model for its model.

If there is any event currently in the event queue with the same model as the current model then this new event is placed into the event queue at the time of the next such event (lowest time) with a priority of **:min**. If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled.

If there is no event in the event queue that matches the current model then the value of **delay** determines what happens to the new event.

If **delay** is **t** then the new event is placed into the set of waiting events to be scheduled after an event which matches the current model.

If **delay** is **nil** then the new event is added to the event queue for immediate execution. Its time will be set to the current time and its priority will be **:max**.

If **delay** is **:abort** then the new event is discarded without being scheduled.

schedule-event-after-change returns 2 values.

If there is no current model or current meta-process or any of the parameters are invalid, then no event is scheduled and both values are nil.

If an event is scheduled then the first value will be the event and the second value will be **t** if the event is in the waiting queue or **nil** if it is in the event queue.

If the event is aborted the first value will be **nil** and the second value will be **:abort**.

schedule-periodic-event

```
(defun schedule-periodic-event (period action
                                &key (module :none) (destination nil)
                                     (priority 0) (params nil)
                                     (details nil) (output t)
                                     (initial-delay 0)))
```

period a time in seconds which should be a non-negative number
action a function or function-name
module a symbol that names the module which is scheduling the event
destination a symbol that names a module or the keyword :device. The current instance of that module or the current model's device will be passed as the first parameter to the action.
priority a number or the keywords :max or :min
params a list of parameters for the action function
details a string describing the event for the trace or **nil**
output either **t** or **nil**
initial-delay a time in seconds which should be a non-negative number

schedule-periodic-event creates a new event with a time that is equal to the current time plus **initial-delay** and using the other supplied parameters for its corresponding attributes and the current model for its model which is then added to the event queue of the current meta-process. After that event occurs a new event will automatically be scheduled to occur **period** seconds after that time with the same parameters as the initial one. That rescheduling will continue every **period** seconds until the event is deleted.

If there are any events waiting to be scheduled they are checked to see if this new event allows them to be scheduled, and every time that it is rescheduled there will be a check of the waiting events.

If any of the parameters are invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and nil is returned.

The scheduled event is returned when successfully created and scheduled.

schedule-break

```
(defun schedule-break (time &key (priority :max) (details nil)))
```

time a time in seconds which should be a non-negative number
priority a number or the keywords :max or :min
details a string describing the event for the trace or **nil**

schedule-break creates a new break event at the specified **time** with the priority and details provided. The model of the event will be the current model and the module is set to :none. A break event does not have an action and is only used to stop the scheduler. That new event is then added to the event queue of the current meta-process.

If any of the parameters are invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and nil is returned.

The scheduled event is returned when successfully created and scheduled.

schedule-break-relative

```
(defun schedule-break-relative (time-delay
                                &key (priority :max) (details nil)))
```

time-delay a time in seconds which should be a non-negative number
priority a number or the keywords :max or :min
details a string describing the event for the trace or **nil**

schedule-break-relative creates a new break event with a time set to the current time plus the specified **time-delay** with the priority and details provided. The model of the event will be the current model and the module is set to :none. A break event does not have an action and is only used to stop the scheduler. That new event is then added to the event queue of the current meta-process.

If any of the parameters are invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and nil is returned.

The scheduled event is returned when successfully created and scheduled.

schedule-break-after-module

```
(defun schedule-break-after-module (after-module
                                    &key (details nil) (delay t)))
```

after-module a symbol that names a module
details a string describing the event for the trace or **nil**
delay one of **t**, **nil**, or **:abort**.

schedule-break-after-module creates a break event with the supplied details and the current model for its model.

If there is an event currently in the event queue with the module name of **after-module** and the same model as the current model then this new event is placed into the event queue at the time of the next such event (lowest time) with a priority of **:min**.

If there is no event in the event queue that matches on both model and module then the value of **delay** determines what happens to the new event.

If **delay** is **t** then the new event is placed into the set of waiting events to be scheduled after an event which matches **after-module** and the current model.

If **delay** is **nil** then the new event is added to the event queue for immediate execution. Its time will be set to the current time and its priority will be **:max**.

If **delay** is **:abort** then the new event is discarded without being scheduled.

schedule-break-after-module returns 2 values.

If there is no current model or current meta-process or any of the parameters are invalid, then no event is scheduled and both values are nil.

If an event is scheduled then the first value will be the event and the second value will be **t** if the event is in the waiting queue or **nil** if it is in the event queue.

If the event is aborted the first value will be **nil** and the second value will be **:abort**.

schedule-break-after-all

```
(defun schedule-break-after-all (&key (details nil)))
```

details a string describing the event for the trace or **nil**

schedule-break-after-all creates a new break event with the provided details. The event's model is set to the current model and the module is set to :none. A break event does not have an action and is only used to stop the scheduler. The time for this new event is the greatest time of any event currently in the event queue of the current meta-process and its priority is :min. It will be inserted into the event queue such that it will occur after all of the events currently scheduled.

If **details** is invalid or there is no current model or current meta-process then a warning is printed, no event is scheduled, and nil is returned.

The scheduled event is returned when successfully created and scheduled.

delete-event

```
(defun delete-event (event))
```

event is an event that was returned by one of the scheduling functions

If **event** is in either the event queue or the waiting queue of the current meta-process it is removed.

If there is no current meta-process a warning is displayed.

If the event is removed from either queue **t** is returned, otherwise it returns **nil**.

3.5 Event Hook Functions

These functions are provided for adding and removing hook functions from the meta-process.

add-pre-event-hook

```
(defun add-pre-event-hook (hook-fn))
```

hook-fn a function which accepts one parameter or the name of such a function

hook-fn function is added to the set of functions to be called before each event is processed in the current meta-process. An event that is about to be processed will be passed to the **hook-fn** function before it is executed by the scheduler.

If there is no current meta-process a warning is displayed and **nil** is returned.
If there is a current meta-process, then an id for **hook-fn** is returned.

add-post-event-hook

```
(defun add-post-event-hook (hook-fn))
```

hook-fn a function which accepts one parameter or the name of such a function

hook-fn function is added to the set of functions to be called after each event is processed in the current meta-process. After an event is executed it will be passed to the **hook-fn** function.

If there is no current meta-process a warning is displayed and **nil** is returned.
If there is a current meta-process, then an id for **hook-fn** is returned.

delete-event-hook

```
(defun delete-event-hook (hook-fn-id))
```

hook-fn-id a hook function id that was returned from either **add-pre-event-hook** or **add-post-event-hook**

If the hook function with **hook-fn-id** is still a member of either set of hook functions in the current meta-process then it is removed from the set of hook functions.

If there is no current meta-process a warning is displayed and **nil** is returned.

If the hook function with **hook-fn-id** has already been removed or **hook-fn-id** is otherwise invalid **nil** is returned.

It a function is successfully removed then that function (or function name) is returned.

3.6 Text Output

The text output of a meta-process will depend on the existence of the printing module (described later) in a model. These commands send an output to each model in the current meta-process and are probably not for general use, but there may be occasions where that is useful.

mp-output

```
(defmacro mp-output (control-string &rest args))
```

control-string is a control-string as would be passed to the format function
args are the arguments to use in that control string

control-string and **args** are passed to **model-output** for each model in the current meta-process.

If there is no current meta-process or no models in the current meta-process, a warning is printed to ***standard-error*** and **nil** is returned.

Otherwise it returns **t**.

mp-warning

```
(defmacro mp-warning (control-string &rest args))
```

control-string is a control-string as would be passed to the format function
args are the arguments to use in that control string

control-string and **args** are passed to **model-warning** for each model in the current meta-process.

If there is no current meta-process or no models in the current meta-process, a warning is printed to ***standard-error*** and **nil** is returned.

Otherwise it returns **t**.

3.7 Multiple Meta-Processes

If multiple meta-processes are needed, then the following commands are available.

define-meta-process

```
(defmacro define-meta-process (mp-name))
(defun define-meta-process-fct (mp-name))
```

mp-name is a symbol

If there is no meta-process with the name **mp-name** already defined then one is created. If a meta-process with that name already exists or the provided name is not a symbol a warning is output.

A newly created meta-process has the initial values for all of its attributes.

define-meta-process returns **mp-name** if a meta-process is successfully created, otherwise **nil** is returned.

delete-meta-process

```
(defmacro delete-meta-process (mp-name))
(defun delete-meta-process-fct (mp-name))
```

mp-name is a symbol

If there is a meta-process with the name **mp-name**, then all of the models in that meta-process are deleted and then the meta-process itself is removed. If no meta-process with that name exists a warning is output. Note, the default meta-process can not be deleted.

delete-meta-process returns **t** if that meta-process is successfully deleted, otherwise **nil** is returned.

with-meta-process

```
(defmacro with-meta-process (mp-name &body body))
(defun with-meta-process-fct (mp-name forms-list))
```

mp-name is a symbol that names a meta-process
body is any number of forms to evaluate
forms-list is a list of forms to evaluate

If **mp-name** is the name of a meta-process then the forms are evaluated in order with the current meta-process set to the one named by **mp-name**. If **mp-name** does not name a meta-process, then none of the forms are evaluated.

with-meta-process returns the value of the last form evaluated or nil if **mp-name** does not name a meta-process.

current-meta-process

```
(defun current-meta-process ())
```

current-meta-process returns the name of the current meta-process or **nil** if there is no current meta-process.

3.8 Models

define-model

```
(defmacro define-model (name &body model-code))
(defun define-model-fct (name model-code-list))
```

name a symbol that will be the name for the new model
model-code any number of forms that will be evaluated for the model
model-code-list a list of forms to evaluate for the model

define-model is used to create a new model in the current meta-process.

The name must not already be used for a model in the current meta-process. If the name is not a symbol or is already used to name a model in the current meta-process a warning will be displayed and the model will not be defined (the old model with that name will remain unchanged if one existed).

When a model is first defined the following sequence of events will occur:

- Create a new model with that name
- with that new model as the current model
    - create the default chunk-types
    - create the default chunks
    - create a new instance of each module
        - call its create function if it exists
        - call its reset function if it exists
    - request all of the parameters from each of the modules
    - evaluate the forms of the model in the order provided

If a model is successfully created then its name is returned otherwise **define-model** returns **nil**.

Every model will need to have a call to **define-model** before issuing any of the model commands because there is no default model in a meta-process. However, if one is working with only a single model then all that is necessary is to provide a name - it is not necessary to enclose all of the model code.

current-model

```
(defun current-model ())
```

current-model returns the name of the current model in the current meta-process or **nil** if there is no current model or no current meta-process.

delete-model

```
(defmacro delete-model (&optional model-name))
(defun delete-model-fct (&optional model-name))
```

model-name a symbol that names a model

If **model-name** is not provided the name of the current-model is used.

If **model-name** is the name of a model in the current meta-process then the following sequence of events will occur:

- the model with that name is set to the current model
- all events generated by that model are removed from the event queue
- each module of the model is deleted
- the model is removed from the set of models in the current meta-process

If **model-name** is valid then **t** is returned.

If **model-name** is not valid or there is no current meta-process then a warning is printed, nothing is done and **nil** is returned.

with-model

```
(defmacro with-model (model-name &body body))
(defun with-model-fct (model-name forms-list))
```

model-name a symbol that names a model in the current meta-process
body any number of forms to execute
forms-list a list of forms to execute

If **model-name** is the name of a model in the current meta-process then the forms are evaluated in order with the current model set to the one named by **model-name**. The value of the last form evaluated is returned.

If **model-name** does not name a model in the current meta-process, or there is no current meta-process then none of the forms are evaluated, a warning is printed and **nil** is returned.

3.9 Chunk-types

chunk-type

```
(defmacro chunk-type (&rest arguments))
(defun chunk-type-fct (&optional name-and-slots))
```

arguments a definition of a chunk-type using the syntax described below
name-and-slots a list with a definition for a chunk-type

chunk-type defines a new chunk-type for use in a model or displays all of the available chunk-types for the model.

If there is no current model then a warning is displayed and **chunk-type** returns nil.

If no **arguments** are provided in the call to **chunk-type** then all of the currently defined chunk-types in the current model will be printed to ***standard-output*** and a list of their names will be returned.

Here is the syntax for a chunk-type definition:

{<type name> | (<type name> (:include <parent name>))} {<doc string>}$^{+}$ {<slot | (<slot> <value>)}*

<type name> a symbol that will be the name of the new chunk-type
<parent name> the name of an existing chunk-type that is to be the parent for the new chunk-type
<doc string> a documentation string for the chunk-type
<slot> a symbol that will name a slot in this chunk-type
<value> a default value for the slot

If a **value** is a symbol then it is assumed to be the name of a chunk. If a chunk by that name does not exist then one with that name is defined of the default chunk-type chunk.

If a **value** is not provided for a slot then the default value is **nil**.

If **type name** is the name of an existing chunk-type, then a warning is displayed, no changes are made to that chunk-type and **nil** is returned.

If the syntax of the provided parameters is correct, then a new chunk-type is defined with the components specified and the name of that chunk-type is returned.

If the syntax is not correct then no chunk-type is defined, a warning is displayed, and **nil** is returned.

Examples:

(chunk-type number value)
(chunk-type (special (:include number)) new-slot)
(chunk-type word “a word chunk-type”)
(chunk-type goal (state start))

chunk-type-p

```
(defmacro chunk-type-p (chunk-type-name?))
(defun chunk-type-p-fct (chunk-type-name?))
```

chunk-type-name? a symbol to test against the names of chunk-types

If **chunk-type-name?** is the name of a chunk-type in the current model then **t** is returned and if it is not the name of a chunk-type then **nil** is returned.

If there is no current model, then a warning is displayed and **nil** is returned.

chunk-type-subtype-p

```
(defmacro chunk-type-subtype-p (chunk-subtype? chunk-supertype))
(defun chunk-type-subtype-p-fct (chunk-subtype? chunk-supertype))
```

chunk-subtype? a symbol which is the name of a chunk-type
chunk-supertype a symbol which is the name of a chunk-type

If either **chunk-subtype?** or **chunk-supertype** is not the name of a chunk-type in the current model, or there is no current model then a warning is displayed and **nil** is returned.

If **chunk-subtype?** is a subtype of **chunk-supertype** in the current model then **t** is returned otherwise **nil** is returned. A chunk-type is always a subtype of itself.

chunk-type-supertypes

```
(defmacro chunk-type-supertypes (chunk-type-name))
(defun chunk-type-supertypes-fct (chunk-type-name))
```

chunk-type-name a symbol which is the name of a chunk-type

chunk-type-supertypes returns a list of the chunk-type names for all of the chunk-types which are supertypes of **chunk-type-name**.

If **chunk-type-name** is not the name of a chunk-type in the current model or there is no current model then a warning is displayed and **nil** is returned.

chunk-type-subtypes

```
(defmacro chunk-type-subtypes (chunk-type-name))
(defun chunk-type-subtypes-fct (chunk-type-name))
```

chunk-type-name a symbol which is the name of a chunk-type

chunk-type-subtypes returns a list of the chunk-type names for all of the chunk-types which are subtypes of **chunk-type-name**.

If **chunk-type-name** is not the name of a chunk-type in the current model or there is no current model then a warning is displayed and **nil** is returned.

chunk-type-slot-names

```
(defmacro chunk-type-slot-names (chunk-type-name))
(defun chunk-type-slot-names-fct (chunk-type-name))
```

chunk-type-name a symbol which is the name of a chunk-type

If **chunk-type-name** is the name of a chunk-type in the current model then **chunk-type-slot-names** returns a list of the names of the slots in the chunk-type **chunk-type-name**.

If **chunk-type-name** does not name a chunk-type or there is no current model then a warning is displayed and **nil** is returned.

chunk-type-slot-default

```
(defmacro chunk-type-slot-default (chunk-type-name slot-name))
(defun chunk-type-slot-default-fct (chunk-type-name slot-name))
```

chunk-type-name a symbol that names a chunk-type
slot-name a symbol which names a slot in the chunk-type **chunk-type-name**

If **chunk-type-name** is the name of a chunk-type in the current model and **slot-name** is the name of a slot in that chunk-type then the default value for that slot is returned.

If either of the parameters is not valid or there is no current model then a warning is displayed and **nil** is returned.

chunk-type-documentation

```
(defmacro chunk-type-documentation (chunk-type-name))
(defun chunk-type-documentation-fct (chunk-type-name))
```

chunk-type-name a symbol that names a chunk-type

If **chunk-type-name** is the name of a chunk-type in the current model then the documentation string of that chunk-type is returned.

If **chunk-type-name** is not the name of a chunk-type in the current model or there is no current model then a warning is displayed and **nil** is returned.

3.10 Chunks

As with the other components of the framework, the internal representation of chunks is not part of the API, and thus they should only be created and accessed through the methods provided. All accessors will reference the chunks by name, and thus the modules will never access the actual chunk constructs. However, because the modules may need to augment the basic chunk definition in order to hold other information (in particular the declarative memory module will need to add a fair amount of parameters to the chunks) a mechanism is provided to allow a module to extend the definition of a chunk and create additional accessors.

define-chunks

```
(defmacro define-chunks (&rest chunks))
(defun define-chunks-fct (chunks-list))
```

chunks any number of chunk definition lists
chunks-list a list of chunk definition lists

A chunk definition list has the following syntax:

({<chunk name> | <chunk name> <doc string>} ISA <chunk-type> {<slot> <value>}*)

<chunk name> a symbol to name the new chunk
<doc string> a string to provide documentation for the chunk
<chunk-type> a symbol that names a chunk-type to use for this chunk
<slot> a symbol that names a slot in **<chunk-type>**
<value> a value to store in the slot

The chunk name is optional, and if not provided a new name will be generated for the chunk. If a chunk-name is provided, it must not name an existing chunk. The chunk-type must name a valid chunk type in the current model. Each slot of the chunk-type may only be set once in the definition.

If the syntax of the description list is correct and all of the components are valid then a new chunk is created with the slot values set as specified. Any slot not specified will have the value of the default for that chunk-type.

If a value for a slot is a symbol then it is assumed to be the name of a chunk. If there is not already a chunk by that name, then one is created of the chunk-type chunk.

However, within a call to define-chunks one can use the names of the chunks that are being defined in the other chunks without having them created as default chunks. Here is an example to clarify that:

```
(define-chunks (a ISA some-chunk-type slot b)
               (b ISA some-chunk-type slot a))
```

Because both **a** and **b** are being defined neither will need to be created automatically.

If the syntax is incorrect or any of the components are invalid in a description list then a warning is displayed and no chunk is created for that chunk description. If there is no current model then a warning is displayed, no chunks are created and **nil** is returned.

A list of the names of the chunks successfully created is returned.

chunk-p

```
(defmacro chunk-p (chunk-name?))
(defun chunk-p-fct (chunk-name?))
```

chunk-name? a symbol to be tested again the names of chunks

If **chunk-name?** is the name of a chunk in the current model then **t** is returned and if it is not the name of a chunk then **nil** is returned.

If there is no current model, then a warning is displayed and **nil** is returned.

chunk-chunk-type

```
(defmacro chunk-chunk-type (chunk-name))
(defun chunk-chunk-type-fct (chunk-name))
```

chunk-name a symbol which is the name of a chunk

If **chunk-name** is the name of a chunk in the current model then the name of the chunk-type for that chunk is returned.

If **chunk-name** is not the name of a chunk or there is no current model, then a warning is displayed and **nil** is returned.

chunk-documentation

```
(defmacro chunk-documentation (chunk-name))
(defun chunk-documentation-fct (chunk-name))
```

chunk-name a symbol which is the name of a chunk

If **chunk-name** is the name of a chunk in the current model then the documentation string for that chunk is returned.

If **chunk-name** is not the name of a chunk or there is no current model, then a warning is displayed and **nil** is returned.

chunk-slot-value

```
(defmacro chunk-slot-value (chunk-name slot-name))
(defun chunk-slot-value-fct (chunk-name slot-name))
```

chunk-name a symbol which is the name of a chunk
slot-name a symbol which names a slot in the chunk

If **chunk-name** is the name of a chunk in the current model and **slot-name** is the name of a slot in that chunk then the value of that slot of the chunk is returned.

If **chunk-name** is not the name of a chunk, **slot-name** is not the name of a slot in that chunk, or there is no current model, then a warning is displayed and **nil** is returned.

set-chunk-slot-value

```
(defmacro set-chunk-slot-value (chunk-name slot-name slot-value))
(defun set-chunk-slot-value-fct (chunk-name slot-name slot-value))
```

chunk-name a symbol which is the name of a chunk
slot-name a symbol which names a slot in the chunk
slot-value a new value for the slot

If **chunk-name** is the name of a chunk in the current model and **slot-name** is the name of a slot in that chunk then the value of that slot of the chunk is set to **slot-value** and **slot-value** is returned.

If **chunk-name** is not the name of a chunk, **slot-name** is not the name of a slot in that chunk, or there is no current model, then a warning is displayed and **nil** is returned.

mod-chunk

```
(defmacro mod-chunk (chunk-name &rest modifications))
(defun mod-chunk-fct (chunk-name modifications-list))
```

chunk-name a symbol that names a chunk
modifications any number of slot-name and value pairs
modifications-list a list of slot-name and value pairs

If **chunk-name** is the name of a chunk in the current model and **modifications** or **modifications-list** is an even length list then the items of the list are considered pair-wise to be the name of a slot in that chunk and a new value for that slot. All of those slots in the chunk are set to the new values specified. **chunk-name** is returned.

mod-chunk is essentially a short hand for multiple calls to **set-chunk-slot-value**.

If **chunk-name** does not name a chunk in the current model, there is no current model, there are an odd number of items in **modifications** or **modifications-list**, or any of the slot names are invalid then a warning is displayed, no changes are made, and **nil** is returned.

Examples:

```
(mod-chunk chunk1 slot1 10 slot2 goal33)
(mod-chunk-fct 'chunk5 '(slot6 "this" slot8 fact10))
```

copy-chunk

```
(defmacro copy-chunk (chunk-name))
(defun copy-chunk-fct (chunk-name))
```

chunk-name a symbol that names a chunk

If **chunk-name** is the name of a chunk in the current model then a new chunk is defined that has a new name, the same chunk-type as **chunk-name**, and the same values for all of the slots as **chunk-name**. The name of the new chunk is returned.

If **chunk-name** does not name a chunk in the current model or there is no current model then a warning is displayed and **nil** is returned.

pprint-chunk

```
(defmacro pprint-chunk (&rest chunk-names))
(defun pprint-chunk-fct (&optional chunk-names-list))
```

chunk-names any number of symbols that name chunks
chunk-names-list a list of symbols that name chunks

For each chunk name in **chunk-names** or **chunk-names-list** that is actually a chunk in the current model **pprint-chunk** will print all of the information about the chunk. That will include the chunk-name, its documentation string, chunk-type and all of the slot-value pairs for that chunk to ***standard-output***. If no names are provided, then all of the chunks in the current model will be printed.

pprint-chunk returns a list of the chunk names for the chunks printed.

If there is no current model then a warning is displayed and **nil** is returned.

chunks

```
(defun chunks ())
```

chunks returns a list of all the chunk names in the current model.

delete-chunk

```
(defmacro delete-chunk (chunk-name))
(defun delete-chunk-fct (chunk-name))
```

chunk-name a symbol that names a chunk

If **chunk-name** is the name of a chunk in the current model then that chunk is removed from the set of chunks in the current model. It returns **chunk-name**.

If **chunk-name** does not name a chunk in the current model or there is no current model then a warning is displayed and **nil** is returned.

delete-chunk *should be used rarely and carefully, because it does not attempt any clean up if that chunk is referenced in the slots of other chunks or if it is currently residing in a buffer, and thus available to other modules.*

merge-chunks

```
(defmacro merge-chunks (chunk-name1 chunk-name2))
(defun merge-chunks-fct (chunk-name1 chunk-name2))
```

chunk-name1 a symbol that names a chunk
chunk-name2 a symbol that names a chunk

If **chunk-name1** and **chunk-name2** are of the same chunk-type and have all of the same values for their slots then both chunks are replaced with a single underlying structure such that changes to either one will affect both chunks. The name of the "merged" chunk will be **chunk-name1**, but references to either name will still be valid.

If the chunks are merged, then any additional chunk parameters that have been added to the chunks will remain those that existed for **chunk-name-1**.

If either chunk is later deleted both of the chunks will become unavailable i.e. deleting any one of a set of merged chunks deletes all of those merged chunks.

If the chunks are successfully merged, then **chunk-name1** is returned.

If the chunks do not match on chunk-type or any slot value then no merging occurs and **nil** is returned.

If either name does not name a chunk in the current model or there is no current model then a warning is displayed and **nil** is returned.

eq-chunks

```
(defmacro eq-chunks (chunk-name1 chunk-name2))
(defun eq-chunks-fct (chunk-name1 chunk-name2))
```

chunk-name1 a symbol that names a chunk
chunk-name2 a symbol that names a chunk

If the chunks named by **chunk-name1** and **chunk-name2** both reference the same underlying chunk then **eq-chunks** returns **t**. Basically, if the names are the same or if the chunks have been merged then **eq-chunks** returns **t**.

Otherwise **nil** is returned, which includes the cases where either **chunk-name1** or **chunk-name2** is not the name of a chunk in the current model or there is no current model.

equal-chunks

```
(defmacro equal-chunks (chunk-name1 chunk-name2))
(defun equal-chunks-fct (chunk-name1 chunk-name2))
```

chunk-name1 a symbol that names a chunk
chunk-name2 a symbol that names a chunk

If the chunks named by **chunk-name1** and **chunk-name2** both reference the same underlying chunk or if they are of the same chunk-type and all of the slots in both chunks are the same (satisfying eq-chunk if chunk names, string= if strings, or equal if something else) then **equal-chunks** returns **t**. Basically, if the chunks "look" the same, then **equal-chunks** returns **t**.

Otherwise **nil** is returned, which includes the cases where either **chunk-name1** or **chunk-name2** is not the name of a chunk in the current model or there is no current model.

3.11 Extending the Chunk Representation

When the chunk structure needs to be augmented for use by a module the following function should be used to add the parameters that the module will use and then the automatically defined accessors can be used to set and get those parameters.

extend-chunks

```
(defmacro extend-chunks (parameter-name &optional (default-value nil)))
(defun extend-chunks-fct (parameter-name &optional (default-value nil)))
```

parameter-name is a symbol which will be the name of the new parameter for the chunk
default-value is the initial value for that parameter and the value all chunks for which the parameter has not been set will return

extend-chunks is used to add a new parameter to the internal chunk structure. If the chunk does not already have a parameter by that name, then an accessor called chunk-*parameter-name* (where *parameter-name* is the **parameter-name** symbol provided) will be created which takes one parameter

that is a chunk name. That accessor can be used to get the corresponding parameter of a chunk or used with setf to change the value of that parameter for the chunk.

If a new accessor is created, then the name of that accessor is returned.

If a parameter by that name already exists, then the keyword :duplicate-parameter will be returned.

If **parameter-name** is not a valid symbol or any other problems occur **nil** is returned.

This has a global impact, and will affect all chunks in all models and all meta-processes.

3.12 Chunk Specifications

Chunk specifications will be the things that are passed to modules through the buffers as requests. As with most of the other framework components, the internal representation of a chunk-spec will not be part of the API and the provided accessors are the only way one should access their components. A module that accepts requests does not have to support all possible chunk specifications that could be passed to it e.g. the motor module will probably not accept modifiers on the slots of the specification. However, it is up to the module writer to parse the chunk-specs and report a warning if there something invalid with the request.

define-chunk-spec

```
(defmacro define-chunk-spec (&rest specifications))
(defun define-chunk-spec-fct (specifications-list))
```

specifications a specification of a chunk as described below
specifications-list a list containing a specification of a chunk as described below

define-chunk-spec is used to create a chunk specification that can be passed to a module as a request. Here is the syntax for a specification:

<chunk name> | ISA <chunk-type> {{<modifier>} <slot name> <slot value>}*

<chunk-name> is the name of a chunk
<chunk-type> is the name of a chunk-type
<modifier> is one of =, -, >, <, <=, or >= and if it is omitted then = is assumed
<slot name> is the name of a slot in the specified chunk-type
<slot value> is any value which is to be tested against in that slot and may be a variable (a symbol which starts with an =) if the test is an = test

A slot name may be used any number of times with any of the modifiers.

The semantics of the modifiers are:

=: the slot contains the specified value
-: the chunk does not contain the specified value
>: the value of the slot is greater-than the specified value
<: the value of the slot is less-than the specified value
<=: the value of the slot is the same as or less-than the specified value
>=: the value of the slot is greater-than or the same as the specified value

If a chunk-name is passed in then the chunk-spec that is created will be one that specifies the chunk-type of the named chunk and then the = test for the current slot value of each of the slots of that chunk. Otherwise, only the components that are provided in the specification will be part of the chunk-spec.

If the syntax of the specification is correct and the components are valid then a chunk specification is returned.

If the syntax is incorrect, any of the components are invalid, or there is not a current model then a warning is displayed and **nil** is returned.

The chunk-spec that is returned can then be passed to a module as a buffer request.

Example, assuming this chunk-type and chunk have been created:

```
(chunk-type goal-type state slot)
(define-chunks (goal10 isa goal-type state start))
```

Then the following three calls to **define-chunk-spec** would return equivalent chunk specifications:

```
(define-chunk-spec goal10)
(define-chunk-spec isa goal-type = state start = slot nil)
(define-chunk-spec isa goal-type state start slot nil)
```

match-chunk-spec-p

```
(defun match-chunk-spec-p (chunk-name chunk-spec
                           &key (=test #'chunk-slot-equal)
                                (-test #'chunk-slot-not-equal)
                                (>test #'>) (>=test #'>=)
                                (<test #'<) (<=test #'<=)))
```

chunk-name a symbol that names a chunk
chunk-spec a chunk specification
chunks a list of chunks or the keyword :all
=test a function or the name of a function that takes two parameters and returns **t** or **nil**
-test a function or the name of a function that takes two parameters and returns **t** or **nil**
>test a function or the name of a function that takes two parameters and returns **t** or **nil**
>=test a function or the name of a function that takes two parameters and returns **t** or **nil**
<test a function or the name of a function that takes two parameters and returns **t** or **nil**
<=test a function or the name of a function that takes two parameters and returns **t** or **nil**

match-chunk-spec-p is used to determine if a chunk named by **chunk-name** matches the **chunk-spec**. If it matches then **t** is returned and if it does not **nil** is returned.

A chunk matches the **chunk-spec** if it is of the chunk-type specified or a subtype of the chunk-type in the specification and the test of every slot in the specification with the corresponding slot value in the chunk returns **t**. To test the slot values the test function provided which corresponds to the modifier on that slot will be called with the slot value as the first parameter and the specification's value as the second parameter unless the slot value specified is a variable. A variable in the **chunk-spec** means that the slot of the chunk can contain any value which is not nil. If more than one slot contains the same variable, then the chunk matches the chunk-spec only if those slots have the same value as tested with the **=test**.

If **chunk-name** or **chunk-spec** is invalid or there is no current model then a warning is displayed and **nil** is returned.

The default test functions for the equal and not-equal tests will operate like this:

```
(defun chunk-slot-equal (arg1 arg2)
  (cond ((and (numberp arg1) (numberp arg2))
         (= arg1 arg2))
        ((and (stringp arg1) (stringp arg2))
         (string-equal arg1 arg2))
        ((and (chunk-p arg1) (chunk-p arg2))
         (eq-chunk-fct arg1 arg2))
        (t
         (equal arg1 arg2))))

(defun chunk-slot-not-equal (arg1 arg2)
 (not (chunk-slot-equal arg1 arg2)))
```

find-matching-chunks

```
(defun find-matching-chunks (chunk-spec
                             &key (chunks :all)
                                  (=test #'chunk-slot-equal)
                                  (-test #'chunk-slot-not-equal)
                                  (>test #'>) (>=test #'>=)
                                  (<test #'<) (<=test #'<=)))
```

chunk-spec a chunk specification
chunks a list of chunk names or the keyword :all
=test a function or the name of a function that takes two parameters and returns **t** or **nil**
-test a function or the name of a function that takes two parameters and returns **t** or **nil**
>test a function or the name of a function that takes two parameters and returns **t** or **nil**
>=test a function or the name of a function that takes two parameters and returns **t** or **nil**
<test a function or the name of a function that takes two parameters and returns **t** or **nil**
<=test a function or the name of a function that takes two parameters and returns **t** or **nil**

find-matching-chunks returns the list of chunk names from **chunks** (or all the chunks in the current model if **chunks** is :all) which match **chunk-spec** as described in **match-chunk-spec-p**.

If **chunk-spec** or **chunks** is invalid or there is no current model then a warning is displayed and **nil** is returned.

chunk-spec-chunk-type

```
(defun chunk-spec-chunk-type (chunk-spec))
```

chunk-spec a chunk specification

chunk-spec-chunk-type returns the name of the chunk-type from the specification.

If **chunk-spec** is not a chunk specification then a warning is displayed and **nil** is returned.

chunk-spec-slots

```
(defun chunk-spec-slots (chunk-spec))
```

chunk-spec a chunk specification

If **chunk-spec-slots** returns a list of the names of the slots that are specified. Each slot will occur only once in the list no matter how many times it may be tested in **chunk-spec**.

If **chunk-spec** is not a chunk specification then a warning is displayed and **nil** is returned.

chunk-spec-slot-spec

```
(defun chunk-spec-slot-spec (chunk-spec
                             &optional (slot nil)))
```

chunk-spec a chunk specification
slot a slot name or **nil**

chunk-spec-slot-specs returns a list of the slot specifications for the slot named **slot** in **chunk-spec** or for all of the slots if **slot** is **nil**. Each slot specification will be a list of three elements. The first element will be the symbol of the modifier for the test. The second will be the symbol that is the slot name and the third will be the value.

If **chunk-spec** is not a chunk specification or **slot** does not name a slot in **chunk-spec** then a warning is displayed and **nil** is returned.

chunk-spec-variable-p

```
(defun chunk-spec-variable-p (chunk-spec-slot-value))
```

chunk-spec-slot-value the value from a slot specification of a chunk-spec

chunk-spec-variable-p returns **t** if **chunk-spec-slot-value** is a variable in a specification i.e. a symbol who's first character of the symbol-name is "=".

Otherwise it returns **nil**.

3.13 Devices

The device is being copied directly from the current ACT-R/PM device interface for now with two additions for setting and retrieving the current device. The docs for this section are mostly copied from the ACT-R/PM documentation at http://chil.rice.edu/byrne/RPM/docs/device_interface.html.

Some of this may need to be adjusted or amended in the context of the framework, particularly in the interest of supporting communication between multiple models, but for now its current definition will be used as is.

3.13.1 Introduction

One of the key differences between "vanilla" ACT-R and ACT-R/PM is that ACT-R/PM has the ability to interact with a simulated device (e.g. a computer). In order for this to work, there must be a simulated device that can communicate with ACT-R/PM.

The simulated device must be a Lisp object, but it can be *any* Lisp object. This object need not be an MCL window. Some things will be handled automatically if the device happens to be an MCL window, but there is no requirement that the device be one.

In order to communicate with ACT-R/PM, the device Lisp object must have certain methods defined for it, which will be called automatically by ACT-R/PM at the appropriate times. The device has to understand different kinds of messages from ACT-R/PM, such as keystrokes and mouse movements, and it has to be able to tell ACT-R/PM what is currently visible.

3.13.2 Setup

These two functions are not part of the current device interface and are new for ACT-R 6.

install-device

```
(defun install-device (device))
```

device is something that is to be used as a device for the current model. It need not be a CLOS object, but must have the appropriate methods defined for its type.

The device of the current-model is set to be **device** and **device** is returned.

If there is no current model, then a warning is displayed and **nil** is returned.

*This replaces **pm-install-device** from ACT-R/PM and ACT-R 5*.

current-device

```
(defun current-device ())
```

current-device returns the currently installed device for the current model.

If there is no current model, then a warning is displayed and **nil** is returned.

3.13.3 Basic Methods

For a Lisp object to be installed as an "RPM device," certain methods need to be defined. Here's what they are, the arguments they take, and what they need to do:

build-features-for

```
build-features-for (device vis-mod)
```
Called by RPM whenever it needs to know what's on the screen. The vis-mod parameter is ACT-R/PM's Vision Module, because sometimes a pointer to that will come in handy when building features. This method should return a list of icon features representing what is currently visible. Details on that are provided in the "Icon Construction" section below.

device-handle-click

```
device-handle-click (device)
```
Called whenever RPM clicks the mouse. Note that it is the device's responsibility to know where the cursor is at all times.

device-handle-keypress

`device-handle-keypress (device key)`
This will be called by RPM whenever the model presses a key. The identity of the key is passed as the `key` parameter.

device-move-cursor-to

`device-move-cursor-to (device xyloc)`
Whenever RPM moves the cursor, this method is called. The location to which RPM is moving the cursor is passed as a vector #(X Y) format.

device-speak-string

`device-speak-string (device string)`
Whenever RPM outputs speech, this method will be called. The string spoken is passed in the `string` parameter.

get-mouse-coordinates

`get-mouse-coordinates (device)`
Should return the current location of the mouse cursor, in (X Y) format.

All of the above methods are already implemented for MCL windows in the file "mcl-interface.lisp," though you can redefine them to suit your needs if you want. The following methods are not defined on MCL windows because they are optional:

device-update

`device-update (device time)`
This is optional, and will be called frequently by RPM. Essentially, whenever RPM is ready to have the device do something, it will call this method with the current time (in seconds) passed in the `time` parameter. This is useful if there are moving objects that need to have their position updated, for example. Because it will be called often, though, this method should do as little computation as possible.

device-update-attended-loc

`device-update-attended-loc (device xyloc)`
Called periodically by RPM to let the device know where RPM is attending, the where being in #(X Y) format in the `xyloc` parameter. This is not really intended as an interaction channel, but more of a debugging aid, and is optional.

3.13.4 Icon Construction

The icon, as described in the section on the Vision Module, is a list of `icon-feature` objects which represents what is currently visible. The only way that the Vision Module can know what is currently visible is for the device to tell it, which is handled with the `build-features-for` method. RPM will call this method whenever it needs a new icon. Thus, to be an RPM device, you must define one of these methods for your device.

This already works somewhat for MCL windows. The `build-features-for` method for MCL windows calls a method called `get-sub-objects` which returns a list of the window's subviews, and then `build-features-for` is called on each of the subviews. Method dispatching figures out what kind of object each subview is and the appropriate method is called for each subview. Methods are currently defined for `editable-text-dialog-item`s, `radio-button-dialog-item`s, `button-dialog-item`s, `check-box-dialog-item`s, and `static-text-dialog-item`s. Thus, to extend processing of an MCL window to new types of views, all that is necessary is a `build-features-for` method for that class.

The recommended way of handling this is to define a new subclass of `VIEW` in which all drawing for your custom object is done. Then, you specialize a method of `BUILD-FEATURES-FOR` for this class of view. This method should return a list of `icon-feature`s. This class is defined in the file "vision-module.lisp," and has several subclasses: `oval-feature`, `rect-feature`, and `line-feature`. Buttons, for example, generate `oval-feature`s. Your object can be constructed out of these basic features, or, if necessary, you can define more subclasses of icon-feature. One thing you should not have to do, however, is write any code for generating features for text. To do that, simply call the `build-string-feats` method, also found in "vision-module.lisp."

An example might help. First, the feature in question are arrows, which can either point left or right. You'll need to define a class for the view which draws arrows (you'll also have to define a `view-draw-contents` method to actually have the arrow drawn on the screen. That's an exercise for the reader.):

```
(defclass arrow-view (simple-view)
  ((direction :accessor direction)))
```

You'll normally want a subtype of `icon-feature` to be put into the icon when the screen is processed:

```
(defclass arrow-feature (icon-feature)
  ((direction :accessor direction :initarg :direction))
  (:default-initargs
    :isa 'ARROW
    :value 'ARROW
    :dmo-id (gentemp "ARROW")))
```

Now, to actually have the system build such a feature when an arrow view is encountered, you need to define a `build-features-for` method, which generates a feature of that class:

```
(defmethod build-features-for ((self arrow-view) (vis-mod vision-module))
  (let ((my-position (view-loc self)))
    (make-instance 'arrow-feature
      :direction (direction self)
      :x (point-h my-position)
      :y (point-v my-position)
      :attended-p nil
      :screen-obj self)))
```

The `:screen-obj` keyword specifies which real object on the screen corresponds to the icon feature. RPM uses this information for things like change detection and tracking.

Of course, we could have made the arrow more complicated and constructed it out of, say, three lines (rather than making an arrow a basic visual feature). If you go that route, one common operation in drawing things on the screen is the use of MCL's `line-to` function. Rather than building `line-features` for all those lines yourself, you can call the function `rpm-line-to` instead. This will both draw the line and create a matching `line-feature`, adding that feature to the icon. Be sure that the RPM device has been set before calling this, or results can potentially be flaky.

Of course, your device might not be an MCL window, which is the point of the Device Interface. That's OK, it just means that there's a little more work you'll need to do. You'll have to figure out how to carve up what RPM can see into `icon-features` (or objects that are instances of classes that are subclasses of `icon-feature`) and return a list of them when `build-features-for` is called on your device.

One other thing: When the icon features are generated by views and they're in MCL windows, the `width` and `height` properties will be filled in automatically by RPM. If your objects are not views in MCL windows, then you'll have to make sure you fill these in so that RPM can compute geometry for Fitts' law stuff. Alternately, if you've subclassed on `icon-feature`, you can provide an `approach-width` method for your class which returns the effective Fitts' width of your object (in degrees of visual angle) given an approach angle (in radians, with zero being an approach on the x-axis from zero). You may need to provide that for non-rectangular objects anyway.

3.14 Buffers

The following functions provide the commands for accessing and manipulating the buffers. There are several things which one can do with a buffer. They are: get the chunk that it holds, test if the buffer is empty, request the state of the buffer's module, request an action of the buffer's module (either a request that it update the buffer or that it modify the chunk currently in the buffer), modify the chunk in the buffer, clear the chunk from the buffer, or place a new chunk into the buffer. An immediate version and a scheduled version of each operation can be used. In general, the scheduled versions of the requests and modifications are preferable to the non-scheduled ones because they get recorded in the trace and allow things to be handled in the time line of the simulation. However, sometimes it may be necessary to use the immediate version. In particular, the reading and testing functions would not make much sense to use in a scheduled fashion, but scheduling a notification of having done so may be important.

An example of using the buffer components can probably be best described in the context of the procedural module for ACT-R. One can consider that are three main activities that are done by the procedural system: conflict resolution, production selection, and production firing.

Conflict resolution would need to check most if not all of the buffers and request the states of possibly many modules. Those buffer reads and module tests would not run through the scheduler, and would need to be direct checks.

If a production were selected, then during the production selection it would schedule a notice of the tests that were performed in that production's matching so that the information was available in the trace. In

fact, there is actually a need to have that information available for modeling fMRI data because the buffer "uses" need to be recorded and the only way to do that will be by watching the event stream.

When a production fires it would schedule all of the buffer actions and requests that occur on the right hand side of that production.

buffer-chunk

```
(defmacro buffer-chunk (&rest buffer-names))
(defun buffer-chunk-fct (buffer-names-list))
```

buffer-names any number of symbols which name buffers
buffer-names-list a list of symbols which name buffers

buffer-chunk returns a list of the names of the chunks in the requested buffers from the current model in the same order as they were requested. If a requested buffer is empty the corresponding value is **nil** and if a buffer name is invalid the corresponding value will be :error.

If no names are provided, then all of the current model's buffers will be printed with their corresponding contents to ***standard-output*** and a list containing a cons of the buffer name and the buffer's current chunk name or nil for each buffer will be returned.

If there is no current model then a warning is printed and **nil** is returned.

buffer-chunk is intended to be a command available for modelers to use.

buffer-read

```
(defun buffer-read (buffer-name))
```

buffer-name a symbol which names a buffer

buffer-read returns the name of the chunk in the named buffer of the current model.

If the buffer is empty the return value is **nil.**

If the buffer name is invalid or there is no current model then a warning is printed and **nil** is returned.

schedule-buffer-read

```
(defun schedule-buffer-read (buffer-name time-delta
                             &key (module-name :none) (priority 0)
                             (output t))
```

buffer-name a symbol that names a buffer
time-delta a time in seconds and should be a non-negative number
module-name a symbol that names the module which is scheduling the event

priority a number or the keyword :max or :min
output either **t** or **nil**

schedule-buffer-read is used to record a buffer reference in the trace of the model. It schedules an event to occur as if the following were executed:

```
(schedule-event-relative time-delta #'buffer-read
                         :module module-name
                         :priority priority
                         :parameters (list buffer-name)
                         :output output)
```

The **buffer-read** action essentially does nothing and is only there to record a reference for the trace and so that hook functions could record such information if necessary.

If any of the parameters are not valid or there is no current model or meta-process then **nil** is returned and nothing is scheduled.

Otherwise **schedule-buffer-read** will return the event generated by scheduling the **buffer-read** action.

test-buffer-state

```
(defun test-buffer-state (buffer-name chunk-spec)
```

buffer-name a symbol that names a buffer
chunk-spec a chunk specification

test-buffer-state sends **chunk-spec** to the module for which **buffer-name** is the interface.

The result returned by the module is returned from **test-buffer-state**.

If the module does not support a state test, **buffer-name** is invalid, or there is not a current model then a warning will be displayed and **test-buffer-state** will return **nil** without consulting the module.

schedule-test-buffer-state

```
(defun schedule-test-buffer-state (buffer-name chunk-spec time-delta
                                   &key (module-name :none) (priority 0)
                                   (output t))
```

buffer-name a symbol that names a buffer
chunk-spec a chunk specification
time-delta a time in seconds and should be a non-negative number
module-name a symbol that names the module which is scheduling the event
priority a number or the keyword :max or :min
output either **t** or **nil**

schedule-buffer-state-test is used to record a test of a module's state in the trace of the model. It schedules an event to occur as if the following were executed:

```
(schedule-event-relative time-delta #'buffer-state-test
                         :module module-name
                         :priority priority
                         :parameters (list buffer-name chunk-spec)
                         :output output)
```

The **buffer-state-test** action is a dummy function which does nothing and is only there to record a reference for the trace and so that hook functions could record such information if necessary. It does not make a request of the module because it is assumed that such a request would already have been performed.

If any of the parameters are not valid or there is no current model or meta-process then **nil** is returned and nothing is scheduled.

Otherwise **schedule-test-buffer-state** will return the event generated by scheduling the **buffer-state-test** action.

empty-buffer-p

```
(defun empty-buffer-p (buffer-name))
```

buffer-name a symbol that names a buffer

empty-buffer-p returns **t** if there is no chunk in the named buffer of the current model or **nil** if there is a chunk in that buffer.

If **buffer-name** is invalid or there is no current model then a warning will be displayed and **nil** will be returned.

schedule-empty-buffer-p

```
(defun schedule-empty-buffer-p (buffer-name time-delta
                                &key (module-name :none) (priority 0)
                                (output t))
```

buffer-name a symbol that names a buffer
time-delta a time in seconds and should be a non-negative number
module-name a symbol that names the module which is scheduling the event
priority a number or the keyword :max or :min
output either **t** or **nil**

schedule-empty-buffer-p is used to record a test of a buffer's emptiness in the trace of the model. It schedules an event to occur as if the following were executed:

```
(schedule-event-relative time-delta #'empty-buffer-p
                         :module module-name
                         :priority priority
                         :parameters (list buffer-name)
                         :output output)
```

The empty-buffer-p action essentially does nothing and is only there to record a reference for the trace and so that hook functions could record such information if necessary.

If any of the parameters are not valid or there is no current model or meta-process then **nil** is returned and nothing is scheduled.

Otherwise **schedule-empty-buffer-p** will return the event generated by scheduling the **empty-buffer-p** action.

clear-buffer

```
(defun clear-buffer (buffer-name))
```

buffer-name a symbol that names a buffer

clear-buffer removes the chunk from the named buffer of the current model.

All modules which registered to be notified when a buffer is cleared are sent that chunk name.

clear-buffer returns the name of the chunk removed if there was one, or **nil** if there was no chunk in the buffer.

If **buffer-name** is not valid or there is no current model, then a warning is printed and **nil** is returned.

schedule-clear-buffer

```
(defun schedule-clear-buffer (buffer-name time-delta
                              &key (module-name :none) (priority 0)
                              (output t))
```

buffer-name a symbol that names a buffer
time-delta a time in seconds and should be a non-negative number
module-name a symbol that names the module which is scheduling the event
priority a number or the keyword :max or :min
output either **t** or **nil**

schedule-clear-buffer is used to schedule that a buffer be cleared instead of directly clearing it. It schedules an event to occur as if the following were executed:

```
(schedule-event-relative time-delta #'clear-buffer
                         :module module-name
                         :priority priority
                         :parameters (list buffer-name)
                         :output output)
```

If any of the parameters are not valid or there is no current model or meta-process then **nil** is returned and nothing is scheduled.

Otherwise **schedule-clear-buffer** will return the event generated by scheduling the **clear-buffer** action.

set-buffer-chunk

```
(defun set-buffer-chunk (buffer-name chunk-name)
```

buffer-name a symbol that names a buffer
chunk-name a symbol that names a chunk

set-buffer-chunk places the chunk **chunk-name** into the buffer **buffer-name**. If the buffer already has a chunk in it at the time of this call then that buffer is cleared before the new chunk is set as the current content using **clear-buffer**. Thus, all registered modules will be informed of the clearing, but no event will be recorded in the trace.

If either **buffer-name** or **chunk-name** is invalid or there is no current model, then a warning will be printed, no change will be made, and **nil** will be returned.

Otherwise, **set-buffer-chunk** returns **chunk-name**.

schedule-set-buffer-chunk

```
(defun schedule-set-buffer-chunk (buffer-name chunk-name time-delta
                                  &key (module-name :none) (priority 0)
                                  (output t))
```

buffer-name a symbol that names a buffer
chunk-name a symbol that names a chunk
time-delta a time in seconds and should be a non-negative number
module-name a symbol that names the module which is scheduling the event
priority a number or the keyword :max or :min
output either **t** or **nil**

schedule-set-buffer-chunk is used to schedule that a buffer be set to hold a chunk instead of directly setting the buffer's contents. It schedules an event to occur as if the following were executed:

```
(schedule-event-relative time-delta #'set-buffer-chunk
                         :module module-name
                         :priority priority
                         :parameters (list buffer-name chunk-name)
                         :output output)
```

If any of the parameters are not valid or there is no current model or meta-process then **nil** is returned and nothing is scheduled.

Otherwise **schedule-set-buffer-chunk** will return the event generated by scheduling the **set-buffer-chunk** action.

module-request

```
(defun module-request (buffer-name chunk-spec)
```

buffer-name is a symbol that names a buffer
chunk-spec a chunk specification

module-request sends **chunk-spec** to the module for which **buffer-name** is the interface in the current model.

If the module does not handle requests, **buffer-name** is invalid, there is not a current model, or the **chunk-spec** is not valid then a warning will be displayed and this call will return **nil** without consulting the module.

Otherwise **module-request** returns **t**.

schedule-module-request

```
(defun schedule-module-request (buffer-name chunk-spec time-delta
                                &key (module-name :none) (priority 0)
                                (output t))
```

buffer-name a symbol that names a buffer
chunk-spec a chunk specification
time-delta a time in seconds and should be a non-negative number
module-name a symbol that names the module which is scheduling the event
priority a number or the keyword :max or :min
output either **t** or **nil**

schedule-module-request is used to schedule the sending of **chunk-spec** to the module for which **buffer-name** is the interface buffer instead of directly sending that request. It schedules an event to occur as if the following were executed:

```
(schedule-event-relative time-delta #'module-request
                         :module module-name
                         :priority priority
                         :parameters (list buffer-name chunk-spec)
                         :output output)
```

If any of the parameters are not valid or there is no current model or meta-process then **nil** is returned and nothing is scheduled.

Otherwise **schedule-module-request** will return the event generated by scheduling the **module-request** action.

module-mod-request

```
(defun module-mod-request (buffer-name modification)
```

buffer-name is a symbol that names a buffer
modifications a list of chunk modifications

module-mod-request sends the chunk modifications to the module for which **buffer-name** is the interface in the current model.

If the module does not handle chunk modification requests, **buffer-name** is invalid, there is not a current model, or the modifications list is not valid then a warning will be displayed and this call will return **nil** without consulting the module.

Otherwise **module-mod-request** returns **t**.

schedule-module-mod-request

```
(defun schedule-module-mod-request (buffer-name modification time-delta
                                    &key (module-name :none) (priority 0)
                                    (output t))
```

buffer-name a symbol that names a buffer
modifications a list of chunk modifications
time-delta a time in seconds and should be a non-negative number
module-name a symbol that names the module which is scheduling the event
priority a number or the keyword :max or :min
output either **t** or **nil**

schedule-module-mod-request is used to schedule the sending of a buffer modification request to the module for which **buffer-name** is the interface buffer instead of directly sending that modification request. It schedules an event to occur as if the following were executed:

```
(schedule-event-relative time-delta #'module-mod-request
                         :module module-name
                         :priority priority
                         :parameters (list buffer-name modification)
                         :output output)
```

If any of the parameters are not valid or there is no current model or meta-process then **nil** is returned and nothing is scheduled.

Otherwise **schedule-module-mod-request** will return the event generated by scheduling the **module-mod-request** action.

mod-buffer-chunk

```
(defun mod-buffer-chunk (buffer-name modifications)
```

buffer-name a symbol that names a buffer
modifications a list of chunk modifications

modify-buffer-chunk is equivalent to calling **mod-chunk-fct** on the chunk in the named buffer with the modifications specified. If the parameters are valid then the modifications are made and **t** is returned.

If the parameters are not valid, the buffer is empty, or there is no current model then a warning is printed and **nil** is returned.

schedule-mod-buffer-chunk

```
(defun schedule-modify-buffer-chunk (buffer-name modifications time-delta
                                     &key (module-name :none) (priority 0)
                                     (output t))
```

buffer-name a symbol that names a buffer
modifications a list of chunk modifications
time-delta a time in seconds and should be a non-negative number
module-name a symbol that names the module which is scheduling the event
priority a number or the keyword :max or :min
output either **t** or **nil**

schedule-modify-buffer-chunk is used to schedule that **modifications** be made to the chunk in **buffer-name** instead of directly modifying that chunk. It schedules an event to occur as if the following were executed:

```
(schedule-event-relative time-delta #'mod-buffer-chunk
                         :module module-name
                         :priority priority
                         :parameters (list buffer-name modifications)
                         :output output)
```

If any of the parameters are not valid or there is no current model or meta-process then **nil** is returned and nothing is scheduled.

Otherwise **schedule-mod-buffer-chunk** will return the event generated by scheduling the **mod-buffer-chunk** action.

3.14.1 Buffer Parameters

At this point each buffer has one parameter associated with it that specifies its source spread (basically an extension of the :ga parameter in ACT-R 5 for each buffer). The user-level parameter for each buffer will be specified when the module is defined, but it may be necessary for another module to get at that parameter value without knowing what the user-level parameter is. For now, an accessor is created

specifically for that purpose, but if it is determined that there should be other parameters associated with buffers in the future this may need to be generalized.

buffer-spread

```
(defun buffer-spread (buffer-name))
```

buffer-name a symbol which names a buffer

buffer-spread returns the value of the source spread parameter for the named buffer in the current model.

If the buffer name is invalid or there is no current model then a warning is printed and **nil** is returned.

3.15 Modules

The modules are where the real work of the system will take place. Each model will have its own "instance" of each module. What an instance is will be up to the writer of the module. The module writing guideline is that it somehow encapsulate all of the information specific to a model in such a way that the instances in each model are independent from the modeler's perspective, or that any dependence between different model's instances of a particular module is well documented. When a module is defined, one specifies the functions which are to be used by the framework for interfacing it to the rest of the system. Outside of those hooks, a module may be implemented however one wants, and can provide any other functions for user interaction that it wants.

define-module

```
(defmacro define-module (module-name buffer-list
                         &key (version "Unspecified") (documentation "")
                              (creation nil) (reset nil) (state nil)
                              (request nil) (buffer-mod nil) (params nil)
                              (delete nil) (notify-on-clear nil)))
```

module-name a symbol that will name the module
buffer-list a list of the buffer name specifications that this module will use
version a string that can be used for version control and bug tracking
documentation a string that describes the module
creation a function, function name or **nil**
reset a function, function name or **nil**
state a function, function name or **nil**
request a function, function name or **nil**
buffer-mod a function, function name or **nil**
params a function, function name or **nil**
delete a function, function name or **nil**
notify-on-clear a function, function name or **nil**

define-module creates a new module for the system which will be referenced by **module-name** and provides the buffers named in **buffer-list** as an interface. The operation of the module is controlled by the remaining parameters which are described in detail below. If a module is successfully defined, then **module-name** is returned.

If there is already a module named **module-name** or buffers with any of the names in **buffer-list** then a warning is displayed, no module is created and **nil** is returned.
An element of the **buffer-list** can be a symbol that names the buffer, a list of a symbol that names the buffer and a keyword that specifies the parameter to use in setting the activation spreading from the buffer (the default parameter name will be :*buffer*-activation where *buffer* is the name of the buffer) or a list of a symbol that names the buffer the keyword that specifies the parameter and a default value for the activation parameter (the default default value is 0 and should be left there).

version should be set to a string that indicates in some fashion a version for this module. Every module should have a version and every update to the module should change that version. This will be displayed when the **mp-print-versions** command is called.

documentation should be a string that contains some brief documentation about the purpose of the module. This will also be displayed when the **mp-print-versions** command is called.

The remainder of the parameters (**creation**, **reset**, **state**, **request**, **buffer-mod**, **params, delete,** and **notify-on-clear**) are for specifying the functions that interface the module to the framework. Those functions will be called by the framework as necessary. The situations in which they will be called and the parameters that will be passed to them are described below. All of the functions are optional, and by leaving the corresponding parameter as nil, the module will not be called for that particular situation.

creation

The creation function will be called only once per instantiation of the module, when a model is first created. The creation function will be passed one parameter which will be the name of the model in which the module is currently being instantiated.

It should return something that identifies this instance of the module for use by the other functions of the module as described below. The return value of the creation function is the "instance" of this module. If there is no create function for a module the instance will be nil for all instantiations of that module.

reset

The reset function will be called after the creation function is called and every time a model containing an instance of the module is reset. The reset function will be passed one parameter which will be the instance of the module for that model.

The reset function should be used to reinitialize the module and typical tasks would be to define chunk-types and chunks that are used by the module.

The return value from the reset function is ignored.

state

The state function is used to report on the state of the module. It will be called in response to any state request being made to any of the buffers of the module. The state function will be passed three parameters. The first will be the instance of the module for the model in which the buffer request was made. The second will be the name of the buffer to which the request was made, and the third will be the chunk specification that was sent to the buffer.

The return value from the state function should be either **t** or **nil**. If the current state of the module is consistent with the specification provided it should return **t** otherwise it should return **nil**.

Stylistically, this function should not have any persistent or time delayed effects nor should it schedule any actions because that is what the "true" requests are for and the distinction should be maintained for consistency.

request

The request function is used to respond to requests made to the buffers of the module. It will be called in response to requests being made to any of the buffers of the module. The request function will be passed three parameters. The first will be the instance of the module for the model in which the buffer request was made. The second will be the name of the buffer to which the request was made, and the third will be the chunk specification that was sent to the buffer.

The return value from the request function is ignored.

Stylistically, this function should schedule events for buffer changes or other actions that it does as a response to the request, particularly if those events are to occur at a future time.

buffer-mod

The buffer-mod function is used to respond to requests made to the module to modify the chunk in any of the module's buffers. The buffer-mod function will be passed three parameters. The first will be the instance of the module for the model in which the buffer-mod request was made. The second will be the name of the buffer to which the request was made, and the third will be a list of chunk modifications indicating how to modify the chunk in the buffer.

The return value from the buffer-mod function is ignored.

Stylistically, this function should schedule the buffer modification or any other actions that it does as a response to the request, particularly if those events are to occur at a future time.

params

The params function is used to report and control the parameters of a module. All of the parameter values are to be maintained by the module, and the params function is how the framework will get those parameters from the module and how it will pass user requests to set them back to the module. The

params function is called in three situations. The first is a request from the framework for the parameters of the module and will occur after the reset function of the module is called. The other times it will be called are in response to the **sgp** command when it will be either a request for the current value of a parameter of the module or to set a new value for a parameter of the module.

The params function will be called with two parameters. The first will be the instance of the module in the model in which the request is being made. The second parameter will be one of three things depending on the situation for which it is being called.

One possibility is that the second parameter is **nil**. That signifies that this is a request from the framework for the module to report all of its parameters. In this case, the params function should return a list of parameters which have been generated by calling **define-parameter**. The parameters of that list do not need to be unique i.e. every instance of the module could return the same list of parameters that was built once when the module code was loaded.

The second possibility is that the second parameter is the name of a parameter for the module. In that case, it is a request for the current value of that parameter, and the params function should return that value.

The third possibility is that the second parameter is a cons where the car is a parameter name and the cdr is a value. In that case, this is a request to set the value of the named parameter if it is owned by the module, or a notification that a non-owned parameter has been changed. If it is a parameter that the module owns then the value is what was passed to **sgp** and it has already passed the valid test if one was provided when creating the parameter. The function should handle the request to change the parameter however the module needs that to be done, and then return the current value of that parameter. Note that how a module "sets" parameters is entirely up to the module implementer, and there is nothing that requires it to return the same value as the one requested for the setting. If the parameter is one which the module does not own, then the value is the return value from the owning module which may or may not be the same as the value which was passed to **sgp**. For a non-owned parameter the return value of the params function is ignored.

delete

The delete function will be called once when a model with an instance of the module is deleted. The delete function will be passed one parameter which will be the instance of the module in the model which is being deleted.

The return value from the delete function is ignored.

notify-on-clear

The notify-on-clear function will be called when any buffer in the model is cleared, regardless of which module defined that buffer. The notify-on-clear function will be passed three parameters. The first will be the instance of the module in that model. The second will be the name of the buffer which is being cleared and the third will be the name of the chunk that is being cleared from that buffer.

The return value of this function is ignored.

The reason for such a test is to enable the functioning of the declarative memory module to have chunks enter declarative memory from the buffers without having to "build that in" and could allow for the creation of alternate declarative memory systems that would not require modifying the internals of the framework. I don't foresee any module other than declarative memory using it at this point, but perhaps once it is there maybe other uses will be found.

get-module

```
(defmacro get-module (module-name))
(defun get-module-fct (module-name))
```

module-name a symbol which is the name of a module

If **module-name** is the name of a module in the current model then the instantiation of that module in the current model is returned.

If **module-name** does not name a module in the current model or there is no current model then a warning is printed and **nil** is returned.

This exists so that if a module provides functions that are called other than through the buffer it can get the correct instantiation of itself to use. It is not really for general purpose use because the instantiations of a module are really only meaningful within the code of the module.

3.16 Parameters

The only function relevant for parameters is one to define them as needed by the params function of a module. The coordination of parameter setting between users and the modules is handled in the framework by **sgp** so there is no need for anyone to decode a parameter definition.

define-parameter

```
(defun define-parameter (param-name
                         &key (owner t) (valid-test nil)
                              (default-value nil) (warning "")
                              (documentation "")))
```

param-name a keyword that will be the name of the parameter
owner either **t** or **nil**
valid-test a function, function name or **nil**
default-value any non-keyword value
warning a string
documentation a string

define-parameter is used to specify the details of a parameter which a module will use. **param-name** will be the name of the parameter, and it must be a keyword. The **owner** value determines whether this

is a parameter that the module is making available to the model for use by **sgp** or whether this is a parameter that the module would like to be informed about but which is owned by another module. If **owner** is **t**, then the module is the one which is making the parameter available and will be responsible for maintaining the value, and if **owner** is **nil** then the parameter must be owned by a different module. If after a model is defined there are parameters without owners then a warning will be displayed indicating which parameters are unowned and which modules requested to be informed about them. [An example when one might be interested in monitoring a parameter without owning it would be the current ACT-R :esc parameter. That is going to be owned by one of the core ACT-R modules, but is potentially important to multiple modules so that they know whether to operate symbolically or to utilize their subsymbolic components. Being able to know when it changes instead of always having to check for its current value could be a lot more efficient.]

The remainder of the parameters are only meaningful if **owner** is **t**, otherwise they are ignored.

valid-test should be a function that will be called every time **sgp** is called to set this parameter. It will be passed one value which will be the potential new value for the parameter. That function should return **t** if the new value is acceptable for the parameter and **nil** if it is not. If the **valid-test** function returns **nil** then a warning will be displayed along with the **warning** string, and the parameter value will not be passed to the module.

default-value is a way to provide a default value for the parameter. When a model is reset, all of the parameters are set to their default values or nil if no default value is specified.

documentation should be set to a string that describes the parameter. It will be used when **sgp** displays all of the parameters. The **documentation** string will be displayed along with the name of the parameter and its current value.

The return value from **define-parameter** is a parameter definition which can be used to build the list that is to be returned by a modules params function.

4. Framework Modules

In addition to the main construct already specified there are three support modules which will be part of the framework. A printing module will exist to coordinate the output for a model, including its trace. A naming module will provide support in generating names for things, like chunks, which are unique within a model and which can then be "cleaned up" when no longer needed. Finally, a pseudo-random number generator module will be included to provide a consistent source of pseudo-random numbers independent of the environment (Lisp implementation and operating system) in which the system is run.

4.1 Printing module

No buffer
one parameter **:v** (verbose)

The print module exists to coordinate the output of a model and may be used by the other modules. It is used by the meta-process for sending output to the model including the event trace.

The v parameter (verbose) operates like a combination of the v parameter and the other tracing parameters from previous versions of ACT-R. It specifies whether any output from the printing module will be displayed and additionally where it will go.

The possible values are:

nil which turns off any output from the printing module for the model
t which then sends the output to ***standard-output***
an open stream in which case output is sent to that stream
a pathname in which case the file specified is opened and all output appended to it
a string denoting a file name that file name is converted to a true pathname and used as above

For the file options, the file will be opened for writing when the parameter is set and then closed when either the parameter is changed, or the module is deleted.

The default value is **t**.

There are three commands provided by the printing module which may be useful for modelers and module writers.

format-trace

```
(defun format-trace (event))
```

event is an event which was either returned by one of the scheduling functions or which was passed to a hook function.

format-trace returns a string that contains the text that will be displayed in the trace for **event** if it is executed and has its output set to **t**.

If **event** is not an event, then **nil** is returned.

A modeler or module writer would not normally need to call this command because the trace is generated automatically by the scheduler, but it could be useful for debugging or recording information during the hook functions.

model-output

```
(defmacro model-output (control-string &rest args))
```

control-string is a control-string as would be passed to the format function
args are the arguments to use in that control string

The **control-string** and **args** are passed to format with the stream set to the one specified by the v parameter and that is followed by a newline.

The result of that format call will be returned.

model-warning

```
(defmacro model-warning (control-string &rest args))
```

control-string is a control-string as would be passed to the format command
args are the arguments to use in that control string

The **control-string** and **args** are passed to format with the stream set to ***error-output***. If the v parameter is not **nil**, and the stream it specifies is not the same as ***error-output***, then the message is also sent to that stream.

It always returns **nil**.

4.2 Naming module

no buffer
no parameters

Because names for things (models, chunks, meta-processes, etc) are symbols there are some issues related to the need to unintern them when they are no longer needed. Modules or model code may need to create new names on the fly, and this module provides a way to generate a new name which is unique within the model and which will be properly removed when no longer needed.

It is recommended that this mechanism be used instead of **gentemp** for any module which requires new symbols, particularly for the names of chunks. It has a couple of additional features which should also make it more user friendly than **gentemp** for modeling.

The names generated by the naming module consist of a prefix and a number. The numbers for each prefix used for a name generated by the naming module will start at 0 within each instance of the naming module and will be reset to 0 when the module is reset.

That means that a deterministic model will always end up with the same names for any newly generated items if they are all created using the naming module. That should make debugging much easier, and when a model's trace is displayed after running it (as in the tutorial for example) the same item names will appear when it is run again.

There are two commands provided by this module.

new-name

```
(defmacro new-name (&optional (prefix "CHUNK")))
(defun new-name-fct (&optional (prefix "CHUNK")))
```

prefix is a string or symbol

It returns a symbol which is generated by appending the current count for that **prefix** to the **prefix**.

Unlike gentemp, it does not guarantee that the symbol does not already exist, however it does guarantee that it is not currently used to name a chunk in the current model. So, if that symbol would match an already used name, the count is incremented until an unused symbol is found.

When the module is deleted or reset it clears its name space and uninterns any symbols it has generated which are no longer "necessary". By necessary, it means that no instance of the naming module has generated such a name, nor was that symbol interned prior to **new-name** generating it for the first time.

new-symbol

```
(defmacro new-symbol (&optional (prefix "CHUNK")))
(defun new-symbol-fct (&optional (prefix "CHUNK")))
```

prefix is a string or symbol

This command is to be used if completely new symbols are required. This should not be used in a module for things that are either throw-away names or user visible names. Those items are better generated with the **new-name** command.

new-symbol operate like **new-name**, except that the symbol returned is guaranteed to not already be interned at the time **new-symbol** is called. When the module in which such a symbol is generated is reset all of those new symbols are uninterned.

There is no guarantee on the numbering of such items, and it will not be restored upon a reset. Because it uninterns the symbols, one should be careful about using this for things that exist outside of the context of a model.

4.3 Random module

no buffers
one parameter **:seed**

This module will provide a consistent pseudo-random number generator across Lisp implementations and/or operating systems and provide a uniform mechanism for specifying a "starting state" for a model to allow for repeatability and debugging. If all of the modules of a model use the provided random functions and there are no other sources of randomness in the model, then resetting the seed to the same point will result in identical performance regardless of implementation. Also, by including a strong pseudo-random number generator it protects the user from issues that could arise due to a weak pseudo-random number generator in any particular Lisp implementation.

The choice of the generator is something to research and discuss, but the interface can be specified now and following the specification for ***random-state*** and **random** in Lisp should be sufficient.

The value of the **seed** parameter is set to a "random value" when the module is created or reset.

It can be read using **sgp**, and the representation will be such that it can be printed and that printed representation will be Lisp readable such that it could be read and passed back to **sgp** as the value of the **seed** parameter to regenerate that state.

It does not require an operator like **make-random-state** because the random module will always return a copy of the seed to **sgp** when requested for its value.

Two functions will be made available to users.

act-r-random

```
(defun act-r-random (limit))
```

limit is a positive integer or a positive float

act-r-random will operate like **random** as defined in the ANSI Lisp specification, without the optional parameter for a random-state.

act-r-random returns a pseudo-random number that is a non-negative number less than **limit** and of the same type as **limit**. An approximately uniform choice distribution is used. If **limit** is an integer, then each of the possible results occurs with (approximate) probability 1/**limit**.

act-r-noise

```
(defun act-r-noise (s))
```

s a non-negative number

act-r-noise will operate like the noise function in ACT-R 5 except it will use the **act-r-random** function to generate the value. It approximates a random sample from a normal distribution with a mean of zero and the provided s value.

Appendix

For now, there is only one example module in the API, but more will be added in future revisions. The example modules are not designed to be efficient or bullet proof, but to demonstrate the components of the API in a concrete example of something that should be somewhat familiar to ACT-R users.

A1. Demonstration Declarative Module

One buffer called **retrieval**
One parameter called **:latency**

The demonstration declarative memory module collects the chunks that have been cleared from the buffers and also provides a function for directly adding chunks to the declarative memory. If there is already a chunk in the declarative memory that is the same (under chunk-equal) as one being cleared then those two chunks are merged. The first time a chunk is stored in the declarative memory module the time of that event is recorded as the creation time of the chunk. The number of times that a chunk is added to declarative memory is recorded in the references of the chunk. It accepts requests through the retrieval buffer to find a chunk in that set. When a request to update the retrieval buffer is made the declarative module will be busy for :latency seconds (the latency parameter is rounded to the tenth of a second). If there is a chunk that matches the chunk-spec requested then that chunk will be placed into the retrieval buffer after :latency seconds have passed. If there is not a chunk that matches the chunk-spec requested then the module will report an error after :latency seconds. If the module is busy when it receives a request it will terminate that pending request, report a warning, and initiate a new request cycle that will take :latency seconds. It does not respond to requests to modify the chunk in the buffer. The state requests that it will respond to are "ISA busy" which will return t if the module is currently busy, "ISA free" which will return t if the module is not currently busy, "ISA error" which will return t if the last request made was not terminated prematurely but did not result in a chunk being placed into the buffer, and "ISA module-state" which will respond to only the modality slot specification. If the modality slot test is free then the state request will return t if the module is not busy. If the modality slot test is busy then the state request will return t if the module is busy. The state request can also test the negation of the modality slot i.e. testing that it is not free will return t if the module is currently busy and vice versa. Any other state requests will result in a warning being displayed and return nil.

A1.1 Declarative Module Source Code

```
;;; Start by defining a structure to hold the instance of the module

(defstruct dm "an instance of the declarative memory module"
  (chunks nil)  ; the list of chunks that are in declarative memory
  (latency 0)   ; the value of the latency parameter
  (busy nil)    ; record whether the module is busy
  (failed nil)) ; record whether the last request failed

;;; Call define-module to hook the module into the framework.

;;; Indicate that it is to be named declarative and that it
;;; has a buffer called retrieval.

(define-module declarative (retrieval)
  :version "0.1b"
  :documentation "An example of the declarative memory module"

  ;; The creation function returns a new dm structure
  ;; that doesn't require knowing the current model's name

  :creation (lambda (x) (declare ignore x) (make-dm))

  :reset reset-dm-module
  :state dm-state-request
  :request dm-request
  :params dm-params
  :notify-on-clear merge-chunk-into-dm

  ;; The buffer-mod and delete functions are not used by this
  ;; module and could be left off since nil is the default value.

  :buffer-mod nil
  :delete nil
  )

;;; Reset-dm-module
;;;
;;; This function will be called each time a model is
;;; reset and after the initial creation of an instance of the
;;; module.  The parameter that is passed is an instance of
;;; the module as returned by the creation function.

(defun reset-dm-module (dm)

  ;; For demonstration it checks if the chunk-types
  ;; used in its state requests exist and if not defines them.

  (unless (chunk-type-p busy)
    (chunk-type busy))
  (unless (chunk-type-p free)
    (chunk-type free))
  (unless (chunk-type-p error)
    (chunk-type error))
```

```
  ;; Adds two parameters to chunks to record information about when
  ;; they are added to declarative memory and how many times they are added.

  (extend-chunks creation-time 0.0)
  (extend-chunks references 0)

  ;; Set all of the slots of this instance to their initial values.

  (setf (dm-chunks dm) nil)
  (setf (dm-latency dm) 0)
  (setf (dm-busy dm) nil)
  (setf (dm-failed dm) nil))


;;; Dm-state-request
;;;
;;; This function will be called each time a request for the module's state is
;;; made of the retrieval buffer.
;;;
;;; The parameters are an instance of the module, the name of the buffer through
;;; which the request was made (which in this case will always be retrieval), and
;;; a chunk-spec that describes the request.
;;;
;;; This module only responds to requests that are of the chunk-types busy, free,
;;; error, or module-state and for module-state requests only = and - tests of the
;;; modality slot are meaningful.

(defun dm-state-request (dm buffer request)
  (case (chunk-spec-chunk-type request)

    ;; For the simple chunk-type tests respond based on value of the
    ;; appropriate slot in the module.

    (busy
     (not (null (dm-busy dm)))) ;; must respond t or nil
    (free
     (null (dm-busy dm)))
    (error
     (dm-failed dm))

    (module-state
     (if (not (and (< 0 (length (chunk-spec-slots request)) 2)
                   (equal (car (chunk-spec-slots request)) 'modality)))

         ;; If there are no slots provided or more than just the modality slot
         ;; report a warning and return nil

         (progn
           (model-warning "Invalid state request of type module-state made of the ~
                           ~s buffer" buffer)
           nil)

       ;; Otherwise, test against the modality slot using functions from the API
       ;; for demonstration.
```

```
       ;; First, create a new chunk that represents the current state of the module

       (let* ((current-state (car (define-chunk-fct
                                   (list
                                    (list 'isa 'module-state
                                          'modality (if (dm-busy dm)
                                                        'busy
                                                      'free))))))

              ;; Then use the provided chunk to chunk-spec tester to determine
              ;; if they match and fail if a modifier other than - or = is
              ;; given as part of the request.

                (result (match-chunk-spec-p current-state request
                                            :>test #'(lambda (x y) nil)
                                            :>=test #'(lambda (x y) nil)
                                            :<test #'(lambda (x y) nil)
                                            :<=test #'(lambda (x y) nil))))

         ;; clean up the temporary chunk used and return the result

         (delete-chunk current-state)
         result)))

    ;; If it is not a valid chunk-type in the request print a warning and
    ;; respond nil

    (t (model-warning "Invalid state request of type ~S made of the ~S buffer"
                      (chunk-spec-chunk-type request)
                      buffer)
       nil)))


;;; Dm-request
;;;
;;; This function will be called each time a request for the module to update the
;;; retrieval buffer is made.
;;;
;;; The parameters are an instance of the module, the name of the buffer through
;;; which the request was made (which in this case will always be retrieval), and
;;; a chunk-spec that describes the request.
;;;
;;; This module responds to all requests by attempting to find a chunk in the set
;;; of chunks it has collected that matches the chunk-spec provided and if one is
;;; found it places it into the buffer.

(defun dm-request (dm buffer request)
  (declare ignore buffer) ;; It is always going to be retrieval

  ;; If the module has not completed the last request

  (when (dm-busy dm)

    ;; Report a warning about that and remove the unexecuted event from the queue.

    (model-warning "Retrieval event ~S has been aborted by a new request"
                   (evt-action (dm-busy dm)))
```

```
    (delete-event (dm-busy dm)))

  ;; Clear the failed attempt flag of the module

  (setf (dm-failed dm) nil)

  ;; Schedule an event to start the retrieval at the current time
  ;; but with a priority of :min and save that as the busy flag
  ;; instead of immediately attempting the retrieval.

  ;; Not important for this demonstration, but in the context of
  ;; a request being made from the RHS of a production this would be
  ;; important to ensure that any buffer modifications have had a chance
  ;; to occur so that the "correct" sources are used for activation spreading.

  (setf (dm-busy dm)
    (schedule-event-relative 0 #'start-retrieval
                             :module 'declarative      ;; declarative is making the
                                                       ;; request
                             :destination 'declarative ;; and is also the module
                                                       ;; that gets the request.
                                                       ;;
                                                       ;; Alternatively, the first
                                                       ;; parameter below could
                                                       ;; have been dm and the
                                                       ;; :destination left at nil.
                                                       ;;
                                                       ;; One would likely use the
                                                       ;; :destination keyword when
                                                       ;; the action is something
                                                       ;; provided by a module for
                                                       ;; which you don't have an
                                                       ;; instance but do know the
                                                       ;; name.
                             :priority :min
                             :params (list request))))


;;; Start-retrieval
;;;
;;; This function is called to actually attempt a retrieval.
;;;
;;; The parameters it receives are an instance of the module and the chunk-spec of
;;; the request.
;;;
;;; It schedules the completion of the retrieval for :latency seconds in the future
;;; and the action it takes depends on whether or not it finds a chunk that
;;; matches.

(defun start-retrieval (dm request)

  ;; get the list of all chunks which match the request

  (let ((chunks (find-matching-chunks request :chunks (dm-chunks dm))))

    ;; Schedule the event for :latency seconds in the future saving
    ;; the event as the busy flag.
```

```
    (setf (dm-busy dm)
      (if chunks

          ;; Just place the first matching chunk in the buffer

          (schedule-event-relative (dm-latency dm) #'retrieved-chunk
                                   :module 'declarative
                                   :params (list dm (car chunks)))

        (schedule-event-relative (dm-latency dm) #'retrieval-failure
                                 :module 'declarative
                                 :destination 'declarative)))))

;;; Retrieved-chunk
;;;
;;; Called as an event when a chunk has been retrieved and is ready to be placed
;;; into the buffer.
;;;
;;; The parameters are an instance of the module and the name of the chunk to put
;;; into the buffer.

(defun retrieved-chunk (dm chunk)

  ;; Clear the busy flag

  (setf (dm-busy dm) nil)

  ;; Schedule an event to put the chunk into the buffer right now instead of
  ;; placing it there directly to comply with the guideline that buffer changes
  ;; should be scheduled.

  (schedule-set-buffer-chunk retrieval chunk 0
                             :module 'declarative
                             :priority :max))

;;; Retrieval-failure
;;;
;;; Called as an event when a chunk failed to be found in response to a request.
;;;
;;; The parameter is an instance of the module.

(defun retrieval-failure (dm)

  ;; Clear the busy flag and set the failure flag.

  (setf (dm-busy dm) nil)
  (setf (dm-failure dm) t))


;;; Dm-params
;;;
;;; This function will be called under three conditions:
;;;
;;;  - the framework wants to find out what parameters the
;;;    module has and any that it wants to "watch"
```

```
;;;  - one of the module's parameter values is requested
;;;  - a change to a parameter is being made.
;;;
;;; The first parameter is an instance of the module and the second indicates under
;;; what condition this call is occurring and the details of that condition if
;;; appropriate.

(defun dm-params (dm param)
  (cond ((null param)  ;; When the second parameter is nil the framework is asking
                       ;; for a list of parameters for the module.

         (list

          ;; This module controls a parameter called :latency
          ;; that can only be set to a number >= 0

          (define-parameter :latency
            :valid-test #'(lambda (x) (and (numberp x) (>= x 0.0)))
            :default-value 0.0
            :warning "Latency must be a number greater than or equal to zero"
            :documentation "Time to complete a retrieval in seconds")

          ;; This module would also like to know if the :esc parameter is changed
          ;; but does not control that parameter.

          (define-parameter :esc
            :owner nil)))

        ((keywordp param)  ;; When the second parameter is a keyword
                           ;; it is a request for that parameter's value

         ;; Since the only parameter for this module is the latency,
         ;; just return that automatically.

         (dm-latency dm))

        ((consp param)  ;; When the second parameter is a cons it is an indication
                        ;; that a parameter change is being made.

         (case (car param)

           ;; For the parameter that the module controls, store the properly
           ;; rounded value and return it.
           ;; Because the module controls the parameter it can
           ;; override a parameter value request.

           (:latency
            (setf (dm-latency dm)
              (* 0.1 (round (cdr param) .1))))

           ;; For the parameter we are watching just note the change.

           (:esc
            (model-output "I now know that :esc is ~S" (cdr param)))))))
```

```
;;; Merge-chunk-into-dm
;;;
;;; This function will be called automatically each time a buffer is cleared.
;;;
;;; The parameters are an instance of the module, the name of the buffer that was
;;; cleared, and the name of the chunk that was in the buffer.
;;;
;;; This module adds that chunk to declarative memory and increments its reference
;;; count.  If a matching chunk already exists in declarative memory, then those
;;; chunks are merged together.  If this is the first occurrence of the chunk, then
;;; its creation time is set to the current time.

(defun merge-chunk-into-dm (dm buffer chunk)
  (declare ignore buffer)  ;; don't care which buffer it came from

  ;; Find any existing matching chunk

  (let ((existing (find chunk (dm-chunks dm) :test #'equal-chunks-fct)))

    (if existing

        ;; if there is a match, merge them making sure to preserve
        ;; the one that was there previously - order matters.

        (merge-chunks-fct existing chunk)

      ;; otherwise set its creation time and add it to the list

      (progn
        (setf (chunk-creation-time chunk) (mp-time))
        (push chunk (dm-chunks dm))))

    ;; then increment the reference count

    (incf (chunk-references chunk))))

;;; Add-dm
;;; Add-dm-fct
;;;
;;; User level function for creating chunks and placing them directly into the
;;; declarative memory list of the declarative memory module of the current model.
;;;
;;; It takes a parameter which is a chunk definition list like define-chunk-fct
;;; takes.  Those chunks are created and then added to the declarative memory
;;; list with the current creation time and 1 reference.

(defmacro add-dm (&rest chunk-list)
  `(add-dm-fct ',chunk-list))

(defun add-dm-fct (chunk-definitions)

  ;; Need to find the current instance of the declarative module

  (let ((dm (get-module declarative)))
```

```
;; if there is one, create the chunks and set the parameters

(if (dm-p dm)

    ;; just pass the list of chunk defs off to define-chunks to do the creation

    (let ((chunks (define-chunks-fct chunk-definitions)))

      ;; Then iterate over those chunks and add them to the module

      (dolist (chunk chunks chunks)
        (push chunk (dm-chunks dm))
        (setf (chunk-creation-time chunk) (mp-time))
        (setf (chunk-references chunk) 1)))

   ;; otherwise report a warning to the meta-process because there may not
   ;; be a current model

  (mp-warning "Could not create chunks because no instance of the declarative ~
               module was found"))))
```

Acknowledgements

I would like to thank John and Mike for their feedback on the earlier (and much rougher) version and Scott and Jon for ideas and suggestions as I worked on the revision.