ACT-R 6 Proposals

Dan Bothell
John R. Anderson
Department of Psychology, Carnegie Mellon University
db30@andrew.cmu.edu
ja@cmu.edu

Michael D. Byrne
Department of Psychology, Rice University
byrne@acm.org

Christian Lebiere
Micro Analysis and Design
clebiere@maad.com

Niels A. Taatgen
Artificial Intelligence, University of Groningen
niels@ai.rug.nl

Preface by Dan

At last year's workshop, in the future of ACT-R session, I suggested that when we implement ACT-R 6 we should start fresh with a specification and go from there. Everybody seemed to like that idea, and developing that specification became my responsibility. There has still been some ongoing work with ACT-R 5, but it seems like it is about time to start the ball rolling on ACT-R 6.

This document is not that specification of the system, but more of a conceptual overview of what the pieces are that will need to be specified, along with some initial proposals for specifying things. In fact after some internal discussion of the proposals and initial work on a specification I chose to set the specification aside for now and instead implement a prototype of the proposals that were discussed in earlier versions of this document to better demonstrate the ideas for discussion before progressing with the full specification. Old habits die hard.

Additional Materials

On the ACT-R website there is now a page of material on ACT-R 6 at http://act-r.psy.cmu.edu/act-r6. There you will find a version of this document along with a few other items. There is a copy of the PowerPoint presentation from the Workshop, which covers the material from this document. You can also get the prototype system - ACT-R 6p, and a set of example models that demonstrate some of the proposals. There is also a Microsoft Excel spreadsheet with some very preliminary work on documenting the semantically meaningful uses of a buffer within a production and the framework for a detailed specification of production compilation.

ACT-R 6p

In the archive are lisp source files for implementing my initial prototype of ACT-R 6. This is by no means a full, robust, or optimized system. It is sufficient to demonstrate the issues to be discussed, and that is about it. It is likely that much of that code will be abandoned after a real specification is developed, but it will run some simple models for now.

To load it, all you need to do is load the "loader.cl" file. Then it is ready to run. It cannot be connected to the ACT-R Environment. It only has a few modules defined, and it provides only a small number of commands currently (which are documented in the included text file called "Command Reference"). It has not been tested very thoroughly. It has no real subsymbolic components. It is not particularly fast. It lacks quite a bit of error checking (in particular production parsing is not nearly as strict as the proposal indicates). Much of the code is sloppy and undocumented. The only file with significant documentation included is the "meta-process.cl" file which serves to demonstrate the amount of documentation that should exist for all the files of the full system. However, it will run the sample models to demonstrate certain issues.

The Models

There are several demonstration models available, and the current ACT-R 5 tutorial unit 1 models are included as a reference. Except for the ACT-R 5 models, all of them can be loaded and run in the provided ACT-R 6p system. They highlight several of the proposals described in this document. The instructions for running each, as well as what it is intending to show, are described in the comments at the top of the individual files.

The Productions Spreadsheet

The Excel spreadsheet contains information about the valid configuration of buffer conditions and actions which could be used within a single production. The proposal is that any of the warnings or errors indicated would result in a message and fail to create a production. It also contains a sheet which could be filled out (it is not currently filled) for a full specification of how the conditions and actions of two successive productions would be combined in production compilation.

Main Issues to consider for ACT-R 6

ACT-R 5 was an incorporation of the perceptual and motor systems of ACT-R/PM into the main architecture and to simplify their use a new concept was introduced into the theory – buffers. The buffers have become central to the operating of the system and most people have found they make it easier to model and easier to teach modeling with ACT-R. The one problem however is that there is neither a specified interface for how a buffer should operate nor a unified mechanism for implementing one. There is a "modular buffer" mechanism that Christian has for extending the system, and it is currently incorporated into the main distribution. However, it is not documented nor do the currently implemented buffers use it. That is probably *the* main objective of specifying ACT-R 6 – taking the buffer concept introduced in ACT-R 5 and really making clear what a buffer is and how it must operate.

There are also still some issues with the incorporation of ACT-R/PM that have not been resolved well yet. The primary one of those is the handling of time and the sequencing of events. ACT-R and ACT-R/PM each have a mechanism for advancing the model through time and right now in ACT-R 5 they both operate together through a somewhat awkward mechanism. Also, RPM is organized around modules (which makes the "modular buffer" name a little confusing), but the current situation is that all of the ACT-R cognition (procedural, declarative, and goals) is considered a single module. When that is considered in conjunction with the buffers it seems that there should be more of a distinction there as well.

Another issue has to do with sources of activation. Currently, only the goal buffer is a source, but there has been some interest in making other buffers sources as well. One view is that the buffers should be treated equally as much as possible. So, indeed all of them should be sources, and perhaps every buffer would have a parameter (similar to the current :ga parameter) which specified how much activation it spread. That is based on a parsimony of implementation and there are some theoretical aspects to consider, but if the default value for those parameters is 0, then by setting :ga to 1 it would be equivalent to ACT-R 5. Thus, other than introducing a bunch of new parameters, it does not seem to break anything. Some other issues with that then are the learning of the Sji values as described in equations 4.3 of "Atomic Components of Thought". It seems that in general that needs to be revisited. The issues are if it applies to buffers other than retrieval for i, the chunk that is "needed", and would j be over the slots of all chunks in buffers that had non-zero activation spread or still just the goal.

Recently there have been several requests from people wanting to implement multiple models in ACT-R. However, the current system does not lend itself to doing that easily. One has to essentially work up a mechanism from scratch for doing so depending on how you want the models to operate (there was a tool provided at one time by Christian called the multi-model extension, but it does not work with ACT-R 5). This seems like the perfect time to build support for multiple models into the system. Adding both synchronous (all models are progressing along the same time line) and asynchronous

(each agent operates in its own time independent of the others) mechanisms into the system now will make it more useful for current and future users.

There are also some loose ends in the current implementation that might as well be cleaned up now. One of those is the claim of the theory that past goals become chunks in memory when they are completed. However, that is not how the system currently operates. Currently, all chunks enter declarative memory upon creation, so as soon as one appears in a buffer it is also available for retrieval from declarative memory. With the separation of the cognitive system into separate modules, that seems like something which should be fixed and generalized across buffers.

One of those that has been a problem for a while is how clear-all operates. The default behavior has an impact on the ability to compile model files for speed i.e. without specifying an optional parameter you cannot compile the model file. If you do specify nil for the optional parameter to clear-all you can then compile the file, but then reset operates as a reload which introduces other potential pitfalls. For any Lisp without an incremental compiler (and MCL is possibly the only full Common Lisp with one) that makes things difficult because one has to separate the model code from any support code in order to compile the critical components.

Another one is the use of gentemp for naming dynamically created objects. It has caused quite a bit of trouble in two ways. First, for any Lisp that doesn't take liberties with the CL specification (MCL actually removes interned symbols after a while, which is quite useful, but not kosher) there is a serious problem with long running models because eventually the memory required for all of the new symbols themselves overwhelms the machine. Also, even for a deterministic model, there is no easy correspondence between the names of new chunks or productions generated on different runs, and that can make debugging a model difficult. ACT-R 5 contains a crude mechanism to help alleviate the first issue, but does nothing to address the second one.

Given the new focus on buffers and modules something that is necessary now is to provide a framework for the easy addition of new components. Not only should we provide such a framework, but the basic components provided with the system should be built using those same mechanisms.

Finally, production compilation is becoming an increasingly important piece of the architecture. Given the new syntax for productions, it will need to be specified very explicitly to be understandable and useable. Also, with the ability of users to add new buffers dynamically, it will have to be generalized so that it can operate with those buffers as well as the existing ones, or at least safely ignore any new buffer.

So, to summarize, here are the issues which need to be worked out in the development of ACT-R 6:

- Clearly specify the buffer mechanism
- Define a unified scheduler

- Split cognition into multiple modules
- Resolve the sources of activation
- Add support for multiple models
- General system cleanup (clear-all and gentemp)
- Incorporate a mechanism for adding new buffers and modules
- Ensure production compilation works well for existing and new buffers

One Big Change for ACT-R 6

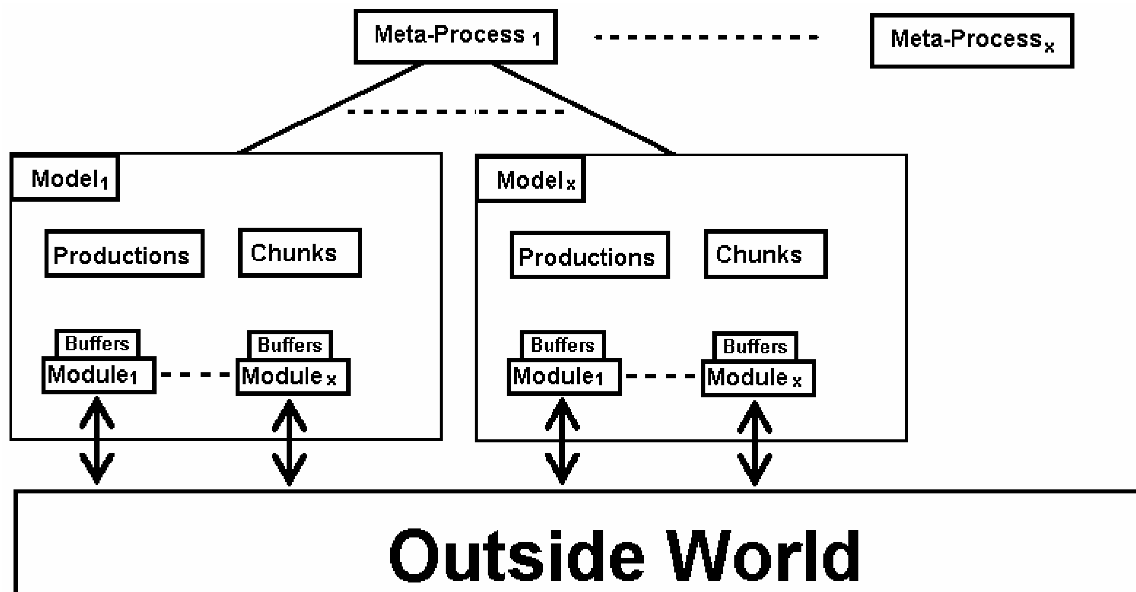
There is one thing that is proposed for the final ACT-R 6 that the previous systems have had (at least to some extent), and that is backward compatibility. There is not a lot of benefit in having ACT-R 6 run models that were written for any of the previous versions. That is already partially the case for ACT-R 5. It will run many ACT-R 4 models, but it will not run all of them without some changes to the model. Moreover, because it tries to be backward compatible leads to some really nasty and subtle "bugs" when writing an ACT-R 5 model, which are extremely difficult to detect especially for a novice who does not know about how ACT-R 4 operates. Probably the most problematic is the misspelling of retrieval on the LHS of a production. That will then be treated as an ACT-R 4 style LHS retrieval, which *most* of the time will be the same chunk as in the retrieval buffer, but it has the additional time cost for the retrieval and when it does not match it is extremely frustrating and difficult to detect.

The proposal is not that we totally abandon the current models. The system would allow for ACT-R 5 models to be converted to ACT-R 6 models fairly easily in most cases, and perhaps some automated tools could be developed for doing that conversion. However, that will require a deliberate conversion on the modeler's part. ACT-R 6 will not contain any of the mechanisms that do the conversion automatically because that would just lead to a perpetuation of the subtle bug type problems. Finally, it is not unprecedented for this to happen when ACT-R makes a big change and thus should not be a large concern, but something of which to be aware.

Proposed system overview

The overall structure of the system as proposed is based heavily upon the current structure of ACT-R/PM, but there are some differences, in particular, the addition of the model abstraction. Another change from the current implementation is the splitting of the current cognitive system into separate modules and defining buffers as actual "things".

An overview of the implementation level for ACT-R 6 looks like this:



There are one or more meta-processes. Each of those can have any number of models associated with it and every model is associated with exactly one meta-process. Each model uses a set of modules from all of the currently defined modules. Those modules may each have some buffers through which they can interact with cognition. The chunks of the model do not have to be elements of declarative memory. The modules may interact with the "Outside World", and it would be possible for multiple models to interact through the "Outside World". However, the implementation of the "Outside World" is not currently discussed, but will likely take a form similar to the current device interface.

Details of the components

Here is some more detail of the specific pieces introduced above.

The meta-process

The meta-process is essentially the system's scheduler. It can control any number of models, and holds the current simulated time and the sequence of actions to perform. The name meta-process was chosen because it is not a part of the theory of ACT-R, but a necessary mechanism for the operation of the system. It is basically a stripped down version of the master-process in ACT-R/PM. The proposal is that everything which a buffer or module does must be scheduled through the meta-process. That way there is a complete trace of every event of the system and there is a clean interface between the components of the system.

This abstraction allows for the implementation of multiple asynchronous models. By creating multiple meta-processes one could have multiple models running in independent time frames. Implementation wise, the proposal is that as with the current ACT-R 5 implementation there would be a default meta-process which is assumed if one is not specified for particular operators. Thus, when working with a single meta-process this abstraction can basically be ignored.

The model

The model is a new abstraction for ACT-R 6 at the code level. It essentially represents a single ACT-R model as one would think of in the previous versions. It contains its own copy of the modules that define what the model can do as well as its own set of chunks and productions. The addition of this abstraction allows for the smooth creation of multiple models within a single meta-process i.e. synchronous models who operate on the same time line.

The chunks

As before, chunks are the storage units of declarative memory, and the proposal is that now they will also be the data structure through which productions, buffers, and modules interact. The important thing to note about chunks in ACT-R 6 is that they do not have any direct association with the declarative memory module. Unlike the previous systems, chunks would no longer "automatically" go into declarative memory upon creation.

They will only enter through the operations on the buffers described below or when explicitly placed there by the modeler.

The buffers

Buffers were introduced in ACT-R 5 to serve as the interface between the productions and modules. That will continue with ACT-R 6, and the specification of exactly how they work is one of the major objectives of this document and covered in detail below.

The module

The module is a generalization of the module class which is currently in the ACT-R/PM implementation. It defines a subsystem of the architecture and specifies the interactions and operations available to a model. A module may interact with the procedural module (productions) through its buffer(s).

There are two open issues relating to modules. The first is whether a module should be able to have more than one buffer. And the second is whether modules should be allowed to interact directly with each other, other than through the productions.

In essence, one cannot have both of those things at the same time and the current (ACT-R 5) visual system is the prime example. It has two components - a "what" and a "where" essentially. Each of those systems places chunks in a buffer, but there is a strong relationship between those chunks that exists outside of the productions. Either it is one module with two buffers, or it is two modules which share a common representation which would basically be interacting directly. Given that it is not really possible to enforce the non-interaction of user defined modules the current inclination is to set guidelines which we recommend, but to have no restraints on either mechanism built in. [A quick note - the prototype system does not enforce either restriction].

Details of some specific modules for ACT-R 6

Here are some of the main modules that will be available in ACT-R 6. These are basically the separation of the current cognitive system into individual modules. This is only really a high level description and not a detailed accounting of their functionality.

Procedural module

The procedural module is the production selection and execution system. It is a separation and encapsulation of functionality currently performed by the existing ACT-R 5 cognitive module. It is where the productions are implemented and is a critical component of the system because the selection and firing of productions coordinates the interactions of the other modules. There would be no buffer associated with the procedural module.

Declarative module

The declarative module is the long term memory system. It is also a separation and encapsulation of functionality currently performed by the existing ACT-R 5 cognitive module. It is another critical module because the operations of the buffers (as defined later) will depend on it. There is one buffer called retrieval associated with the declarative module.

Goal module

The goal module implements a very simple intentional system. In ACT-R 5 there is not really a module behind the goal – it is “only a buffer”, but that is not an option in the ACT-R 6 framework. Conceptually, there must be a module to perform the actions. There is one buffer called goal associated with the goal module.

Eval module

The eval module is designed to take the place of !eval!, !bind!, and !output! in ACT-R 6. This module will execute arbitrary Lisp code passed in as requests. It exists because it seems desirable to make the production syntax only buffer based without giving up any of the flexibility available. There will be one buffer called eval associated with the eval module.

Buffers

First, there needs to be an explicit definition of a buffer in ACT-R 6 and here is the proposal: “A buffer is the interface through which the procedural module interacts with other modules in the system.” Of course how it operates still needs to be specified, and how buffers operate is intimately tied to how productions work. So, before going on to how buffers work, let us first look at productions.

Productions

Productions are the unit of procedural memory. They consist of conditions and actions. The conditions are tests upon the contents of the buffers and modules and the actions are requests made to the buffers and modules. For ACT-R 6, a new production syntax is represented below. It is very similar to the ACT-R 5 syntax, but does have some important differences.

With the introduction of buffers in ACT-R 5 the production syntax was modified slightly to accommodate them, but the system was still capable of using ACT-R 4 style LHS retrievals intermixed with the buffer tests. As noted above, that has resulted in some very difficult to detect bugs. That, along with the desire to clarify and unify how the buffers will operate in ACT-R 6 has led to the development of a new syntax for productions in ACT-R 6. The new syntax looks very much like ACT-R 5 with a few additions and some omissions. Below is a BNF diagram of the new syntax. [Some notational conventions that are used are putting literals into double quotes and using square brackets to represent optional items.]

Production ::= "(" "p" <name> [<doc-string>] <condition>* "==>" <action>* ")"

<name> ::= <symbol>

<doc-string> ::= <string>

<condition> ::= {"="<buffer-name>">" {<chunk> | <chunk-spec>}} |
{"+"<buffer-name>">" {<chunk> | <chunk-spec>}} |
{"-"<buffer-name>">"}

<action> ::= {"="<buffer-name>">" {<chunk> | <slot-pair>*}} |
{"+"<buffer-name>">" {<chunk> | <chunk-spec>}} |
{"-"<buffer-name>">" }

<buffer-name> ::= a <symbol> which is one of the valid buffer names for the model

<chunk> ::= {<chunk-name> | <variable>}

<chunk-name> :: a <symbol> which is the name of a chunk

<variable> ::= {"="<used-buffer-name> | "="<symbol>}

<used-buffer-name> ::= a <buffer-name> that is used in an "=" condition

<chunk-spec> ::= "isa" <chunk-type> <slot-spec>*

<chunk-type> ::= a <symbol> which is the name of a chunk-type defined for the model

<slot-spec> ::= [<modifier>] <slot-pair>

<slot-pair> ::= <slot-name> <slot-value>

<modifier> ::= "-" | "<" | "<=" | ">" | ">="

<slot-name> ::= a <symbol> which is the name of a slot for the appropriate <chunk-type>

<slot-value> ::= <chunk-name> | <variable> | <number> | <string> | <slot-list> | "nil"

<slot-list> ::= "("{<slot-list> | <number> | <variable> | <symbol> | <string>}+ ")"

<symbol> ::= a valid Lisp symbol

<number> ::= a valid Lisp number

<string> ::= a valid Lisp string

Now, what does that all mean? First, the big change is that the only thing that can occur on the LHS is a condition (a buffer test) and the only thing that can occur on the RHS is a buffer action. There is also a generalization of the shorthand notation that is available on the RHS in ACT-R 5 that allows for a specific chunk to be requested. Here are the general semantics of the condition and action specifications:

=buffer> tests or modifies the chunk in the buffer (tests/affects the **chunk**)
-buffer> tests that the buffer is empty or empties the buffer (tests/affects the **buffer**)
+buffer> makes a request of the buffer's module (tests/affects the **module**)

The generalization of the shorthand notation that allows a chunk name or variable to be used instead of a full specification is that this can be thought of as fully expanding that chunk in place (all of the slot bindings) and then performing the match or request as normal. There is one situation where that does not quite hold however which is noted below, and that is something that may mean the generalization needs to be reexamined.

Examples

Here are examples that describe in detail using all the possible condition and action options. Textual is used as the buffer name to avoid any confusion with current semantics (keeping with the informal 'buffers should end in "al"' rule), and the proposal is that all buffers will be treated equally:

LHS

=textual> isa <chunk-type> <slot-spec>*

This is a test of the chunk in the textual buffer against the specified slots. It is the same as the =textual> test would be now if there were a textual buffer.

=textual> <chunk>

This tests that the chunk in the textual buffer matches exactly with the chunk specified, but does not require that they be the exact same chunk i.e. the names are not compared.

-textual>

This tests whether the textual buffer is empty (contains no chunk). This is a new test, but not something which should be controversial. Its purpose is to allow for a more robust testing of state which can help to alleviate the necessity for explicit markers in the goals and elsewhere.

+textual> isa <chunk-type> <slot-spec>*

This is the one test that differs significantly from the current system because such a mechanism does not really exist. It is designed as the replacement for the necessity of the <buffer>-state buffers. It is in effect a LHS request of a module. These requests are explicitly for the module's state and they are "in place" requests – one does not have to wait to harvest the result nor does it affect the current contents of the buffer. The <chunk-type> must be either module-state or a subtype of module-state, which allows for module writers to implement modules with more complex states. Thus, specifically, this would request the state of the textual buffer's module and test the result of that state request.

+textual> <chunk>

Same as above, but instead of specifying the module-state chunk directly it is compared to a chunk specified using the shorthand notation.

RHS

=textual> <slot-pair>*

This is a modification of the chunk currently in the textual buffer setting the slots as specified. It is the same as an =textual> action would be now. The one interesting thing is that there is no requirement to specify any slot-pairs (the * means 0 or more), and there is a reason that such an action might be necessary which will be described later.

=textual> <chunk>

This action has two possible interpretations to consider. The one that is being proposed is the one that the generalization would suggest. This would modify the chunk currently in the textual buffer so that all its slots matched the ones of the specified chunk. The one issue there is that the ISA slot is not modifiable, so the generalization is not quite accurate and there is the possibility for “run time” errors of that action when the chunk-types do not match, but that seems likely to be a rare occurrence and thus using the generalization seems appropriate.

[The other possibility has to do with suggestions that the productions should be able to place specific chunks directly into the buffers independent of the buffer’s module and this would seem to be one way to do so. That seems to provide some extra flexibility into the system, but it is a new flexibility which does bring up some interesting issues that would need to be considered if it were to be the mechanism chosen. If such a mechanism is desirable, then perhaps a new operator should be introduced for it instead of modifying this one, and *buffer> seems like a reasonable option if such a thing is to be investigated. If such a mechanism is implemented, then it would likely be the “!eval!” of ACT-R 6 - something that is there because people need it for ease of modeling, but not something strongly encouraged. It will violate one of the constraints on buffers proposed below. However, if the buffer system is also changed from the proposal below in the future, then perhaps that is not as unusual a command to add as it seems to be now.]

-textual>

This clears the chunk from the textual buffer. This differs from ACT-R 5 in that currently for some buffers the “–” action also sends an implicit clear request to the buffer’s module, but now that would have to be sent explicitly as a request if it is desired.

+textual> isa <chunk-type> <slot-spec>*

This would send the chunk specification as a request to the textual buffer’s module. This isn’t any different from what a +textual> request would do in ACT-R 5 (at least conceptually).

+textual> <chunk>

This would send that chunk explicitly as a request to the textual buffer's module. [That does not necessarily feel right, because the generalization is supposed to be the expansion in place of the chunk which seems like a copy would be better. However, as some modules are currently implemented in ACT-R 5 having the explicit chunk is necessary. So, for now the proposal is that the explicit chunk gets sent, but that may be reevaluated as things progress.] This differs from the current RHS "direct requests" because currently they are buffer specific (for the goal buffer it places it in the buffer directly, and for a retrieval it is a special retrieval request of that chunk only. The proposal is that the mechanisms not differentiate this on a buffer by buffer basis - all this does is send that chunk to the appropriate module. What the corresponding module does with such a request of course can differ from module to module, but at the level of sending the request they are all equal.

Valid Productions

On sheet1 of the production spreadsheet available all possible combinations of buffer tests and actions that can occur for a particular buffer within a single production are represented, and it is copied below as well. Indicated in green are what are consider valid productions, the orange cells are productions which have some problem for which a warning should be generated, and the red cells contain an error. The proposal is that either a warning or an error should prevent the creation of the production so that there is no possibility of run-time issues, particularly in the context of production compilation. On the spreadsheet, there is also an indication of the state of the buffer when a particular production is selected and after it fires, which will be important for production compilation.

General Case

		LHS							
		{}	{=}	{+}	{-}	{=,+}	{=,-}	{+,-}	{=,+,-}
	{}						W1		W1
	{=}	E1		E1	E1		W1	E1	W1
	{+}	**	**		**		W1		W1
	{-}		*		*	*	W1	*	W1
	{=,+}	E1	**	E1	E1		W1	E1	W1
RHS	{=,-}	E1		E1	E1		W1	E1	W1
	{+,-}	W2	W2	W2	W2	W2	W1,W2	W2	W1,W2
	{=,+,-}	E1,W2	W2	E1,W2	E1,W2	W2	W1,W2	E1,W2	W1,W2

Open Production Issue

There is one issues that has come up a couple of times in discussions of ACT-R 5 as well as previously with ACT-R 4 and it has some implications at the theory level which should also be worked out before continuing further with ACT-R 6. That is whether <chunk-type> and <slot-name> (as used in the BNF above) should allow for the use of variables. This provides some extra flexibility to the productions and can get around issues of fixed slot names and rigid type hierarchies, but it also comes with a lot of issues that would need to be worked out.

Buffers (continued)

Now that how productions work has been described, it seems that buffers are really a very passive construct which need to do the following:

- hold at most 1 chunk
- return that chunk
- modify that chunk
- remove that chunk
- pass a provided chunk to a module
- get and return the state chunk from a module

Given that, the proposal is that all buffers share a common implementation and defining a new buffer amounts to essentially providing a name by which to reference it and the module to which it is the interface. In fact, because it is tied directly to a module the proposal is that it would be implicitly constructed with the definition of a module. Thus, users will never create new buffers directly. Users will create new modules which can have buffers associated with them, but the implementation of the buffers is not modifiable.

The next issues then are how a chunk goes from a buffer into declarative memory, how cognition and the interfaced module affect the buffer, and how references are attributed when dealing with base-level learning.

Buffers, Chunks and Declarative Memory

As for chunks entering declarative memory the proposal is that whenever the buffer is cleared, the chunk is merged into declarative memory. By merging, it means the mechanism that exists for cleared goal chunks now. That is, if an identical chunk (one that matches everywhere except possibly name) already exists in declarative memory no new chunk is added. Otherwise, the new chunk becomes a member of declarative memory. In fact the proposal is that declarative memory will never have "duplicate" chunks. Whenever a chunk is added to declarative memory it will be merged. That might have some impact on existing models, but it should not be a critical issue because from the model's perspective identical chunks are indistinguishable now anyway and it seems to really only be an issue when using the name explicitly "outside" of the model.

How does a buffer change

What operations can affect a buffer? The buffer's module may place a chunk into the buffer either as the result of a request or through buffer stuffing. This happens in the current system and is the primary use of the buffer – to hold a chunk.
The proposal for ACT-R 6 is that except for the declarative memory module (the retrieval buffer), modules must create new chunks to place into buffers and not reuse chunks which exist in declarative memory. The merging mechanism will then add those chunks to declarative memory when necessary. [The possible exception would be the RHS "+buffer> <chunk>" requests which pass a chunk directly to a module which it could then place it into the buffer. Again, that does not feel quite right, but may be a necessary thing.]

The actions of a production will also affect the buffer.

For a –buffer request, the buffer is cleared and the chunk is merged with declarative memory.

For an =buffer request, the chunk currently in the buffer is modified, and remains there.

For a +buffer request, the buffer will automatically be cleared, as if by –buffer, in addition to sending the specified chunk to the module. Then the module may respond by placing some chunk into the buffer when the action completes. The implicit –buffer action is there to insure that a chunk could not be "lost" in processing a new request, because the old one would be properly merged with declarative memory first.

An open issue with that is what happens if the chunk that gets cleared was never harvested? The current proposal is that it would be merged into declarative memory. However, perhaps there would be good reasons for not doing that, and it is something that will need to be investigated.

Those are very similar to how things operate in ACT-R 5, but there is a new proposal for ACT-R 6 that can potentially help with ensuring the production compilation mechanism operates cleanly, and would potentially alleviate some of the tedium of buffer management within productions. This mechanism is speculative at this time and discussion is welcome.

Strict Harvesting

The idea is that we take the harvesting metaphor more literally and have a LHS test against a buffer also clear the buffer unless there is a RHS action to “reseed” that chunk. Any =buffer request on the RHS would be sufficient to maintain the chunk in the buffer. That is the reason for the empty =buffer> action in the production syntax. It allows for the situation where one wants to test a buffer and keep the current chunk in that buffer without changing it.

Looking at the productions from the tutorial models, there is a constant need for clearing the visual-location buffer after using it. That seems to be something that confuses students. With this new mechanism, the issue is changed so that now one only needs to worry about keeping a buffer around if it is needed, and that seems like an easier system to use.

Also, this avoids issues of “uninitiated” buffer testing that can arise in production compilation when multiple successive productions test against the results of a single request. As an example of the problem, consider three productions p1, p2 and p3 which fire in that order. p1 requests a retrieval and both p2 and p3 test against the retrieval buffer with no actions upon it. Now, p1 and p2 will be compiled into a new production, call it p1’. The problem is that because the retrieval request is removed in the compilation process, if p1’ is chosen at some point in the future p3 will not be able to successfully fire next because there will not be a chunk in the retrieval buffer. However, if p2 had to maintain that chunk in the buffer so that it was available for p3 that could be considered in the compilation process and it would be possible to avoid that problem. It still might not be an easy issue to resolve, but at least it would be possible.

This change to the operation does not seem like it has much in the way of consequences for existing models, but it is definitely still open for discussion. A quick look at available ACT-R 5 models indicates that multiple accesses like that are fairly rare, so it should not be encountered often, and since the system is not backward compatible anyway that would be one of the minor changes necessary to update a model to ACT-R 6 i.e. adding an empty modification to unchanged but further used buffers. The only real issue seems to be that it makes the initial description of things a little more complicated.

Buffers and Base-Level Learning

As for base-level learning, the proposal is that a generalization of the current mechanism from ACT-R 5 should be used. However, there are still some unresolved issues. A reference would be credited when either:

a) a LHS reference is made to a chunk

In ACT-R 5 only references to the retrieval buffer are counted, but in keeping with the treating of buffers equally, the proposal would be that all LHS references would count. That is however something that should be discussed and investigated further.

b) The chunk is merged with an existing chunk

The references of the merged chunks would be added together.

The biggest potential problem with that is that the chunk in the goal buffer has the potential to get a lot of references, but perhaps that is not a bad thing overall. The kosher implementation of subgoaling requires retrieval of old goals from declarative memory. Thus having those chunks receive lots of references to keep their activations pretty high seems like a good thing.

Another issue with it is that because the chunks do not enter declarative memory until they are cleared from the buffers it seems a little strange to talk about references for base-level learning of chunks that are not a part of declarative memory. However, that is probably just something which will take a little getting used to, and those that learn the mechanism without knowing how the "old" systems work probably will not find it strange.

Summary

This document describes the proposed changes and remaining open issues at a pretty high level. The available prototype system, its command reference, and the example models get down into more details of implementation. These items should provide a good vehicle for discussion of the issues still at hand.

The timeline that is currently envisioned is to come to a resolution on most of these issues in the next couple of months. Then have the full specification of the system ready for discussion by next summer. That would be followed by an alpha version of ACT-R 6 being taught as a class locally in the spring of 2005, and then the official release by the Workshop of 2005.