Notes on Production Compilation

While the ultimate specification of production compilation is contained in the ACT-R source code the following notes cover most of the details. This is an updated description based off of the original production compilation prototype design documents written by John Anderson. Originally there were two documents. One described when it was safe to compose a pair of productions and the other detailed how to calculate the resulting production. Those have been combined and updated by Dan Bothell below. This document assumes that one is familiar with the ACT-R architecture and the details of ACT-R production syntax.

In addition, there is an Excel spreadsheet called “compilation.xls” found in the docs directory of the ACT-R distribution. Each page of that worksheet describes a type of buffer and indicates all of the valid buffer usage combinations for that buffer between a pair of productions that can be combined through production compilation. Those sheets are used to create the ACT-R code which determines whether or not productions can be combined. Details on how to read those spreadsheets is also included below.

Safe Productions

Production compilation is designed to produce safe productions. By safe productions we mean productions that won't cause problems that didn't exist in the original productions before compilation. Thus, assuming the original productions are safe, production compilation should not lead to productions which do things like: generate errors in the model by jamming a module, dropping any perceptual or motor actions from the model, or short circuiting (have a set of conditions which are always true after its own actions). To ensure that safety there are some fairly strict guidelines on which productions can be combined through production compilation. Here are the general rules for which productions can be combined.

1. If there are any eval conditions or actions they must be explicitly marked as safe using the `!safe-eval!` operator.

2. There must be no `!bind!`, `!mv-bind!` or `!safe-bind!` conditions, no `!bind!` or `!mv-bind!` actions, however explicitly indicated `!safe-bind!` actions are allowed.

3. The first production in the pair may not have a `!stop!` action in it.

4. There must be no direct buffer actions like:

```
+retrieval> =element
```
or
```
=goal> =new-goal
```

5. None of the conditions may contain slot modifiers that test for other than equality.

6. No buffer may be tested multiple times in the conditions of a production i.e. all slot tests for a given buffer must occur within a single buffer specification, and only a single action of each type (modification, request, or modification request) is allowed for a given buffer within a production.

7. The productions must have fired within a threshold time which defaults to 3 seconds, but may be changed with the `:tt` parameter.

8. Each buffer which is used in any way in either of the productions must have a safe usage between the two productions. To determine if the usage is safe the compilation type of the buffer is consulted. There are five types of buffers currently defined for compilation and here are their general descriptions and general set of acceptable usage:

a. Goal type buffers -- these are buffers distinguished by immediate effects (thus the module is never busy), requests create new chunks, the buffer is not subject to strict harvesting, and chunk modification is acceptable and dependable (what's specified in the production is guaranteed to occur). These buffers can be combined if there are no queries

in either production, no explicit clearing actions in either production and at most one of the productions makes a request for a new chunk. The goal buffer is the only buffer which defaults to the goal type.

b. Motor type buffers -- these are buffers with no result chunk from requests, take time to have their effects, and have the potential for jamming requests. These buffers cannot be combined if there are any references to a chunk in the buffer. If there are no such references then they can be combined if there is at most one request between the two productions and the queries are compatible. To consider if the queries are compatible the location of any request must be considered. If there is no request or the request occurs in the second production then the queries are compatible if only one production makes a query or if the queries are the same in both productions. If the first production has a request then the queries of the first production do not matter and the queries are compatible if the second production has no query or if the query of the second production is an explicit check for "state busy" (more generally all that matters is that the state must not be tested for free but for now the extreme stance of requiring a busy test is being used). The manual and vocal buffers are motor type buffers by default, and this is also the default type for a newly created buffer that doesn't have a compilation type set.

c. Perceptual type buffers -- these buffers can have chunks placed into them in response to a request, the request can take time, the buffers are strict harvested, the module may jam on requests, and the module may put a chunk into the buffer without a request (buffer stuffing). As a precaution we are going to consider direct modification to the chunks in these buffers by the productions to be unsafe. Thus, any explicit clearing or chunk modification action in either production will prevent their compilation. Like motor type buffers, we will not combine two productions which both make a request to the buffer. We will not combine productions where the first makes a request and the second tests the resulting chunk because the assumption is that the world is unpredictable and no assumptions should be made about the result of a perceptual request to be safe (this may be too restrictive in some cases and one may want to have such a compilation take effect for things like visual-locations where an item always occurs in the same location, but the default operation is to avoid that). We will not compile over a buffer stuffing occurrence which would be the case when both productions test the chunk in the buffer without a corresponding request in the first production. Thus, the only safe conditions for a perceptual buffer are when only one of the productions tests the contents of the buffer, only one production makes a request of the buffer, and the queries are compatible between the two productions. The compatibility of queries is the same as for a motor style buffer with the additional consideration that "state busy" or "buffer empty" may be tested in the second case. The visual, visual-location, aural, aural-location, and temporal buffers are considered perceptual by default.

d. Retrieval type buffers -- these are like perceptual buffers except for a difference in the assumptions about requests. We assume that the effect of a request to a retrieval type buffer can be predicted reliably because it does not depend on the external world, the declarative memory does not modify or delete chunks, and the module does not jam (a second request just replaces the first). The same set of constraints as for perceptual

buffers apply to retrieval buffers, except for one important difference. If a production with a request is followed by a production which harvests that resulting chunk we allow for the combination of the productions. The tests on the chunk from the second production in that situation are eliminated from the combined production by a proceduralization procedure described below. Only the retrieval buffer is of the retrieval type by default.

e. Imaginal type buffers -- these buffers are like the goal type buffers, except that there may be some time which passes between the request and the resulting chunk being placed into the buffer. Because the state of the module may be important while waiting for the result of a request the imaginal type buffers allow for the querying of the buffer whereas the goal type buffers do not allow for queries. Thus, the imaginal type buffers allow for request and modification actions like the goal type and also allow for queries in a manner similar to the perceptual buffers. That combination results in a lot of additional cases which are allowed, and because of that, there is one new restriction which is imposed that doesn't apply to either of the other two types. If both productions test the buffer contents, the first production makes a request, and the second production also makes a query of the buffer, then that query in the second production must be for "state free" to ensure that the situation is safe. The imaginal buffer is the only buffer of the imaginal type by default.

Calculating the New Production

Compiling two productions involves three steps – calculating their variable unification, substituting that unification into the rules, and then collapsing the two rules into one. Here is a description of the process along with a demonstration of how the process occurs for a pair of productions. [Note however that this description is not an exact description of the operation of the actual Lisp code which implements the mechanism in the ACT-R system itself. That code is more general so that it may handle all of the default buffer types as well as being able to handle user defined types and also combines some of the later steps for efficiency.]

A. Unification

The first task in production compilation is to produce a unification of the buffer descriptions. I call this unification because of its similarity to unification in first-order logic (see Russell & Norvig) but it is not exactly serving the same purpose. To consider the need for unification consider the production pair in the following model:

```
(define-model demo

  (sgp :epl t)

  (chunk-type goal arg1 arg2 arg3 arg4)
  (chunk-type fact arg1 arg2 arg3 arg4)

  (add-dm (goal isa goal arg1 3 arg2 1 arg3 1 arg4 6)
          (fact isa fact arg1 3 arg2 3 arg3 5 arg4 1))

  (goal-focus goal)

  (p first
     =goal>
       isa goal
       arg1 =v1
       arg2 =v2
       arg3 =v2
       arg4 =v3
     ==>
     =goal>
       arg1 =v2
       arg2 =v3
       arg3 =v1
     +retrieval>
       isa fact
       arg1 3
       arg2 =v1)

  (p second
     =goal>
       isa goal
       arg1 =v0
       arg2 =v1
```

```
      arg3 3
    =retrieval>
      isa fact
      arg3 =v2
    ==>
    =goal>
      arg2 =v2
      arg3 =v1
      arg4 =v0))
```

When that model is run it will generate this production through compilation:

```
(P PRODUCTION0
  "FIRST & SECOND - FACT"
   =GOAL>
       ISA GOAL
       ARG1 3
       ARG2 =V2
       ARG3 =V2
       ARG4 =V3
 ==>
   =GOAL>
       ARG1 =V2
       ARG2 5
       ARG3 =V3
       ARG4 =V2
)
```

To generate that production requires tracking the different identities of the variables `=v1` and `=v2` in the two productions and determining the final state of the goal chunk.

As a step to calculating this rule we need to calculate the most-general unifier. Here are the steps by which this is done.

Step 1. Renaming

We rename any of the variables that are the same (and thus potentially clash) in the second production. So, in our example above it would become something like this:

```
(p second-a
    =goal>
      isa goal
      arg1 =v0
      arg2 =v1-a
      arg3 3
    =retrieval>
      isa fact
      arg3 =v2-a
    ==>
    =goal>
      arg2 =v2-a
      arg3 =v1-a
      arg4 =v0)
```

Step 2: Extracting buffer descriptions to map

We now map the buffer descriptions in the two productions. There are three cases to consider depending on whether there is a request in the first production (note we assume that we are dealing with legal cases and so it is not the case that these two productions are blocked by a request as they might well be for a perceptual or motor buffer):

Step 2a: No request in first production

In this case, we want to make consistent the description of the buffer at the end of the first production and its description at the beginning of the second production. For the first production, this is the description of the buffer in the condition plus any updates produced by the action. In the case of our goal buffer in the first production we had

```
=goal>
  isa goal
  arg1 =v1
  arg2 =v2
  arg3 =v2
  arg4 =v3
==>
=goal>
  arg1 =v2
  arg2 =v3
  arg3 =v1
```

which implies that the final state of the goal buffer was

```
=goal>
  isa goal
  arg1 =v2
  arg2 =v3
  arg3 =v1
  arg4 =v3
```

The condition of the second production specifies the state at the beginning of the second production:

```
=goal>
  isa goal
  arg1 =v0
  arg2 =v1-a
  arg3 3
```

Step 2b: A non-retrieval type request in first production

Suppose our goal description from the first production had been

```
=goal>
  isa goal
  arg1 =v1
  arg2 =v2
  arg3 =v2
  arg4 =v3
==>
+goal>
  isa goal
  arg1 =v2
  arg2 =v3
  arg3 =v1
```

Then the condition would have been irrelevant to the buffer state at the end of the first production and our buffer description would have been:

```
+goal>
  isa goal
  arg1 =v2
  arg2 =v3
  arg3 =v1
```

This would have been paired with the same condition from the second description.

Step 2c: A retrieval-type request in first production

In this case there are three descriptions to relate. The first is the buffer description at the end of the first production:

```
+retrieval>
  isa fact
  arg1 3
  arg2 =v1
```

The second is the actual chunk placed in the buffer:

```
FACT
  ISA FACT
   ARG1  3
   ARG2  3
   ARG3  5
   ARG4  1
```

And the third is the condition of the second production:

```
=retrieval>
  isa fact
  arg3 =v2-a
```

Step 3: Extract Mappings from buffer descriptions

We now take the buffer descriptions and map the elements that fill comparable slots. In the case of the goal descriptions we would get the mappings:

=v0 → =v2
=v1-a → =v3
=v1 → 3

In the first two mappings it really does not matter whether we map production 1 variables onto production 2 variables or vice versa. However, because of our renaming of production 2 clashes, it is better to map production 2 variables onto production 1. However, if a variable is mapped onto a constant then we have no choice but to map from that variable name. In the case above this is a production 1 variable.

(Note that in general it is possible that we might have two constants in the same slot and these would conflict. However, this will not happen with productions that follow each other under normal circumstances.)

In the case of the retrieval extraction what we want to do is to map the variables from both production descriptions onto the constants of the chunk:

=v1 → 3
=v2-a → 5

In addition to the buffer mappings, if there is a safe-bind in the actions of the first production we will map that variable onto the value returned by the Lisp expression.

Step 4: Merging Mappings to be Consistent

The mappings from different buffers have to be combined. In the case above the combination is just the union of the two mappings:

=v0 → =v2
=v1-a → =v3
=v1 → 3
=v2-a → 5

However, in other cases it can become more complicated. For instance, suppose the retrieval condition in the second production had been:

```
=retrieval>
  isa fact
  arg3 =v2-a
  arg4 =v0
```

Then we would have also added the mapping:

=v0 → 1

That has to be combined with =v0 → =v2 from the mapping of the goal buffer. Depending on order in unification we can either get the resulting pair:

=v2→1 and =v0→1

or

=v0 → =v2 and =v2→ 1

For the second pair, if they are applied in the given sequence that results in =v0 and =v2 both being replaced by 1, but the sequence =v2→1 and =v0→=v2 would not achieve this.

The unification algorithm I coded is based on the general unification algorithm as described in Figure 9.1 of Russell & Norvig. However, it is simpler because we are not mapping compound expressions. In the Russell&Norvig algorithm there is the possibility for unification to fail but this actually cannot occur when two productions have fired one after another.

B. Substitution

Having now obtained a set of substitutions from the unification process, one needs to go throught the original productions performing the substitution on them. Given our set of substitutions

=v0 → =v2
=v1-a → =v3
=v1 → 3
=v2-a → 5

This transforms our original productions into:

```
(p first*
     =goal>
       isa goal
       arg1 3
       arg2 =v2
       arg3 =v2
       arg4 =v3
     ==>
     =goal>
       arg1 =v2
       arg2 =v3
       arg3 3
     +retrieval>
       isa fact
       arg1 3
```

```
      arg2 3)

(p second*
   =goal>
     isa goal
     arg1 =v2
     arg2 =v3
     arg3 3
   =retrieval>
     isa fact
     arg3 5
   ==>
   =goal>
     arg2 5
     arg3 =v3
     arg4 =v2)
```

which provides a consistent labelling from which we can begin the merging step. The example above does not have any safe-evals, safe-binds, or queries but the substituions should apply to these as well.

C. Collapsing

Now we will describe the general mechanisms used to collapse the conditions and actions from the two productions into one. The buffer tests, buffer modifications, and requests for a given buffer are considered together using the following rules as appropriate for the buffer type (for example motor style buffers do not allow for conditions testing a chunk in the buffer). The other components: evals, binds, and queries will be considered separately.

1. If there is no request in the first production

The condition of the new production will be formed by determining C1 + (C2 – A1) where C1 is the buffer condition from the first production, C2 - A1 is the conditions in the second production which were not created by the modification actions of the first production, and + denotes the union of the two sets of conditions.

The modification action of the new production will be the union of the modification actions from the second production with modification actions from the first production which are not also changed by the second production.

If there is a request in the second production then that request is added to the new production.

2. A request in the first production of a non-retrieval type buffer

The condition of the new production will just be the condition of the first production. This is essentially the same as the previous case since with the request C2 – A1 would be the empty set since A1 generates a new chunk and thus is responsible for all the conditions tested in the second.

If there is a modification action in the first production then that is added to the new production.

The request in the new production will be formed by taking the request from the first production and appying any modification actions from the second production to that request. This is similar to the way the modification action is determined from the previous case.

3. A request in the first production of a retrieval type buffer

The condition of the new production will be the condition of the first if there are any. This is effectively the same as both the cases above.

A retrieval type buffer does not allow for modifications, thus there will be none to consider.

The request from the first production is not used since it has already been used in the mapping process above.

If there is a requests in the second production then that will be added to the actions of the new production.

Safe-Evals

The union of the safe-evals from the conditions of both productions will be used in the conidtions of the new production and the union of the safe-eval actions from both productions will be added to the new production's actions. The union is used so as to remove any duplicates between the productions.

Safe-Binds

Safe-binds are only allowed in the actions of the production as described under the conditions for composition. Only the safe-binds from the second production will be used in the new production because any from the first production were already considered as constants in the mapping step.

Queries

With respect to queries, only some buffer types allow for queries and there are different rules based on the type as to what sorts of queries are allowed, but assuming the

productions are valid for compilation the process will be the same. First, a query from the first production is always used if there is one. How a query from the second is handled, when there is one, depends on whether or not there is also a request in the first production. If there is no request in the first production then the query from the second is unioned with any query from the first to determine the query for the new production. If there is a request in the first, then the query from the second production is not used since that query was only appropriate for having the second production safely follow the first because of that request.

The Buffer Type Spreadsheets

The spreadsheets found in the “compilation.xls” file provide the specific conditions for determining whether or not a particular buffer type allows a pair of productions to be composed. Each separate sheet describes a specific type of buffer, and the content of those sheets is used directly to generate the code which specifies a compilation buffer type. The cells at the top of the sheet contain details necessary for generating the code and are probably not all that informative, but will be described later for those that are interested in possibly adding new types.

The important part of the sheet is the table in rows 5-45. The cells of that table specify the conditions under which the usage of this buffer type in the two productions can be composed. The rows indicate the buffer’s usage in the first production and the columns are for the buffer's usage in p2. The numbers along the outside of the table are a reference used internally in the code for representing the particular condition and that condition is also displayed symbolically in the table. There are 6 ways in which a buffer may be used in a production and the following symbols are used to denote them:

LHS test: =
LHS query: ?
RHS modification: =
RHS explicit clear: -
RHS request: +
RHS modification request: *

In the symbolic representation the “>” is used to separate the LHS and RHS usage, a "." separates items on the same side, and empty brackets {} indicates no usage of the buffer on that side. Order doesn't matter since the production testing is considered in parallel and actions have a fixed priority regardless of the order in the production. Thus, =>=.+.* is a production with a lhs test of the buffer and then a modification, a request, and a modification request on the RHS. Not all 64 possible combinations are included in the table because only those combinations which can occur in a valid production are shown e.g. since one can’t modify a buffer that isn’t tested on the LHS of a production there is no {}>= case in the table.

The cells of the table indicate whether or not productions with those particular usages can be composed. If it is nil then they cannot be composed and the cell will be color coded with a color that indicates a reason for that being blocked located in a key below the table. A cell could be nil for multiple reasons, but only one of the colors is represented. If the cell is t then that combination can be composed without restrictions. If it is any other value then that value names a function to call to test some additional constraint which must be met to determine if those productions can be composed. A simple description of those functions' operations is located below the table along with some additional comments about this type of buffer.

The additional information at the top provides the details necessary for the compilation module to create and use this compilation buffer type. The A1 cell holds the name of the compilation buffer type being specified. B1 holds the name of a function to call for performing the mapping of constants to variables for that buffer between the two productions' or nil if no such mapping is necessary. C1 holds the name of a function to call for creating the conditions and actions for that buffer in the new production. Cell D1 holds the name of a function to call for performing any extra compatibility testing necessary after composability has been determined for the productions, or the value nil if there is no additional test necessary. Cell E1 specifies the name of a function to call to determine if this buffer type requires instantiating any variable slot usage the buffer may have in a dynamic production or nil if no such check is required. If that function returns a true value then the buffer will have all of the variablized slots instantiated before any of the other production compilation steps are performed. If cell F1 is non-nil then this type is to be treated like retrieval for purposes of mapping i.e. the result chunk is used to create constant bindings and the request is expected to be dropped from the resulting production. If cell F1 is nil then this type will not be treated in that manner. Cell G1 can be the name of a function or nil. If it names a function then when there is a failure to compose two productions because of a condition for this buffer that function will be called to produce a string to provide details about the failure for the production compilation trace. Cell A2 is a list of the buffers which will be set to this type, or the value :default if this should be the
default type for any buffers not explicitly specified in some type.

Procedural Partial Matching

When both procedural partial matching and production compilation are enabled there are some complications to the above mechanisms which at this time are not completely resolved. If one uses both of those mechanisms together then I would appreciate any feedback you have on what you experience and what you would like the results to be.

All of the above descriptions still basically hold with respect to the composition process when :ppm is enabled, but because there's the possibility for different constants to have been matched in the parent productions with :ppm enabled the mechanism will attempt to compose productions which would not be composable without :ppm enabled. The complications occur when there are differing constants between the productions and/or differences between the constants and the variable bindings. Generally, the mechanism should always do something reasonable, but there are some situations which can result in different outcomes for seemingly similar starting situations. A good example of that occurs in the ppm-compilation test model and the critical situation can be summarized with a model like this:

```
(define-model foo
                (sgp :v t :esc t :epl t :pct t :ppm 1)
               (chunk-type goal step s1 s2)
               (define-chunks (a isa chunk) (b isa chunk)
                 (g isa goal step 1 s1 b s2 a))
               (set-similarities (a b -.5))

               (p p1
                  =goal>
                  isa goal
                  step 1
                  s1 A
                  ==>
                  =goal>
                  step 2)
               (p p2
                  =goal>
                  isa goal
                  step 2
                  s2 =x
                  s1 =x
                  ==>
                  =goal>
                  step 3)

               (goal-focus g))
```

The question is what should the resulting composed production look like? One candidate would be that it should be the same as one would get with :ppm disabled:

```
(P PRODUCTION0
  "P1 & P2"
```

```
   =GOAL>
        ISA GOAL
        STEP 1
        S1 A
        S2 A
 ==>
   =GOAL>
        STEP 3
)
```

Another candidate takes the approach that the composed production should have two utility penalties since each of the parents had a penalty so that the composed production doesn't get the same utility as a pair where only one of the parents has mismatched. That would look something like this:

```
(P PRODUCTION0
  "P1 & P2"
   =GOAL>
        ISA GOAL
        S1 A
        STEP 1
        S1 B
        S2 B
 ==>
   =GOAL>
        STEP 3
)
```

As it turns out, right now the ACT-R software can produce both of those depending on which value was bound to =x in the instantiation of p2. How it gets instantiated at this time is a function of the order of the slot tests in the production definition, so swapping the slot tests changes the resulting composed production. That seems problematic, but for now that "feature" is being allowed so that people can get it to do what they want if they use it.

As a further complication, if the variable from p2 is actually used in the actions of that production then things seem even more confusing since even without compilation turned on that production could do different things based on which value gets bound. So with composition turned on what should the resulting production do? Presumably it should do the "same thing" as the parents, but is that same thing the exact action which took place or the same potential for "different" things to happen?

There are some potentially interesting questions to answer here and for now they're being left somewhat unanswered. The software will try to do something "reasonable" but that may or may not be "right". Until somebody really applies these mechanisms together in modeling real data they will remain in that unanswered state.