

## **ACT-R Model Writing**

This text and the corresponding texts in other units of the tutorial are included to help introduce cognitive modelers to the process of writing, testing, and debugging ACT-R models. Unlike the main tutorial units which cover the theory and use of ACT-R, these documents will cover issues related to using ACT-R from a software development perspective. They will focus mostly on how to use the tools provided to build and debug models, but will also describe some of the typical problems one may encounter in various situations and provide suggestions for how to deal with those.

### **Models are Programs**

The important thing to note up front is that an ACT-R model is a program – it is a set of instructions which will be executed by the ACT-R software. There are many different methodologies which one can use when writing programs as well as different approaches to software testing which one can employ. These guides are not going to promote any specific approaches to either task. Instead, they will attempt to describe general techniques and the tools which one can use when working with ACT-R regardless of the programming and testing methods being used.

Learning to write ACT-R models is similar to learning a new programming language. However, ACT-R as a programming language differs significantly from most other languages and the objectives of writing a model are typically not the same things one tries to achieve in other programming tasks. Because of that, one of the difficulties that many beginning ACT-R modelers have is trying to treat writing an ACT-R model just like a programming task in any other programming language instead of taking the details of ACT-R into account. Some of the important differences to keep in mind while modeling with ACT-R will be described in this section.

From a language perspective, probably the biggest difference between ACT-R and other programming languages is what will be running the program. The model is not being written as commands for a computer to execute, but as commands for a cognitive processor (essentially a simulated human mind) to perform. In addition to that, the operators available for use in writing the model are very low-level actions, much like assembly language in a computer programming language. Thus ACT-R is basically the opposite of most programming languages. It is a very low-level language written to run on a “processor” with many high-level capabilities built into it whereas most languages are a high-level set of operators targeting a very general low-level processor for execution.

Another difference with ACT-R is in how the sequence of actions is determined. In many programming languages the programmer specifies the commands to perform as a specific sequence of instructions with each one happening after the previous as written in the program. For ACT-R however the order of the productions themselves in the code does not matter, nor does the order of the conditions within an individual production matter. The next action to perform, i.e. which production to fire, is based on which one currently matches the current state of the buffers and modules, and that requires satisfying all of the conditions on the LHS of a production. Thus, the modeler is responsible for explicitly building the sequence of actions to take into the model because there is no automatic way to have the system iterate through them “in order”.

Finally, perhaps the biggest difference between writing a cognitive model and most other programming tasks is that for cognitive modeling one is typically attempting to simulate or predict human behavior and performance, and human performance is often not optimal or efficient from a computer programming perspective. Thus, optimizations and efficient design metrics which are important in normal programming tasks, like efficient algorithms, code reuse, minimal number of steps, etc, are not always good design choices for creating an ACT-R model because such models will not perform “like a person”. Instead, one has to consider the task from a human perspective and rely on psychological research and performance data to guide the design of the model.

## **ACT-R and Lisp**

While ACT-R is its own modeling language, it is itself written in Lisp. ACT-R models are also valid Lisp programs and much of the modeler’s interaction with ACT-R will occur through the standard interactive Lisp prompt (often referred to as the REPL for read-eval-print loop). Because of that, some familiarity with Lisp programming is recommended before learning to model in ACT-R, but it is not absolutely required. Similarly, one should also be familiar with the tools provided by the Lisp which is being used so that if unexpected things, like a Lisp error, happen you know how to deal with them. For the most part, this guide will assume the modeler is familiar with using Lisp and will not discuss general Lisp concepts, but the next section will describe how to differentiate warnings generated by issues in the ACT-R model from those which are the result of a more general Lisp problem.

## **Errors and Warnings**

When writing a model one is likely to encounter various errors and warnings from ACT-R and Lisp. This section will provide some information on generally how to deal with those and how to determine whether the problem was reported by ACT-R or the underlying Lisp.

## Lisp Errors

An error is a serious condition that has occurred in the Lisp and it will cause things to stop until it is dealt with. Typical things that will cause a Lisp error are missing or unbalanced parenthesis that result in invalid Lisp code, trying to execute functions which do not exist, or calling functions with invalid or an incorrect number of arguments. When an error occurs it will display some information describing the error which occurred, and then it will have to be dealt with before continuing. How one does that is going to vary from Lisp to Lisp. In some Lisps a dialog window may appear to display the error and provide options for dealing with it, whereas other Lisps may just print the information at the prompt and wait for an explicit command to be entered to deal with the error. Often when a Lisp stops because of an error it will make multiple options available that allow for debugging or continuing on while ignoring the error, but unless you are comfortable with Lisp programming the recommendation is to always choose the option which “aborts” the error. After aborting the error you will usually have to fix whatever caused the error and the error message should provide enough of a description to tell you what happened and thus what needs to be fixed.

Here are two examples of the same error, calling the ACT-R run command without the required time argument, displayed at the prompt in different Lisps along with the action needed to abort the error:

```
CG-USER(95): (run)
Error: RUN got 0 args, wanted at least 1 arg.
[condition type: PROGRAM-ERROR]
Restart actions (select using :continue):
  0: Return to Top Level (an "abort" restart).
  1: Exit the CG event-handling loop (event-loop)
  2: Unwind to the top-level event-handling loop.
  3: Exit this IDE listener (Listener 1).
  4: Abort entirely from this (lisp) process.

[1] CG-USER(96): :continue 0
CG-USER(97):

? (run )
> Error: Too few arguments in call to #<Compiled-function RUN #x4C2E4CE>:
>      0 arguments provided, at least 1 required.
> While executing: RUN, in process listener(1).
> Type :POP to abort, :R for a list of available restarts.
> Type :? for other options.
1 > :pop

?
```

The first thing to look at is the description of the error. In this case it is indicating that there were not enough arguments provided to the run function. The specific error description is going to vary among Lisps, but should be sufficient to describe what happened in all of them. After that, the error message indicates some way to “abort” the error, and again it is not the same in the

different Lisps. In the first Lisp there are several options listed and it indicates that :continue must be used to pick one, with 0 being the one that aborts the error. In the second one it indicates that typing :pop will abort the error. Something else to notice is that when the Lisp is in an error state it will indicate how many errors are currently pending at the beginning of the command prompt (the 1s in these cases). Then, after the error has been aborted, that error indicator goes away. Thus, whenever you see a number at the beginning of the REPL prompt you should attempt to clear those pending errors before doing anything else.

## Lisp Warnings

Lisp warnings are less serious than errors and will not bring things to a halt nor will they require explicit action to clear them. They are an indication that something unexpected or unusual was encountered which you may need to correct. The types of things for which Lisp warnings may be displayed are going to vary from Lisp to Lisp, however here are some typical things that most Lisps will indicate a warning for: defining functions that use undefined variables, defining variables in functions and then not using them, loading a file which redefines a function defined elsewhere, and defining functions that reference other functions which do not yet exist. A Lisp warning will typically be displayed after the prompt as a Lisp comment which starts with a semicolon. Because they do not cause the system to halt they are often easy to ignore, but the recommendation is to read and understand every warning is displayed when you load a model instead of just ignoring them because they are likely to indicate something that could lead to an error later. Here are some warnings displayed in different Lisps when defining a function called test that creates a variable called x, but does not use it:

```
;Compiler warnings :  
;   In TEST: Unused lexical variable X  
TEST  
?  
  
; While compiling TEST:  
Warning: Variable X is never used.  
TEST  
T  
NIL  
CG-USER(100):
```

Notice how in both cases after the warning the prompt does not indicate an error condition after displaying the warning.

## ACT-R Warnings

Warnings from ACT-R are very similar to the warnings from Lisp. They are an indication that there is a potentially problematic situation in the ACT-R model or accompanying Lisp code. An ACT-R warning may occur when the model is loaded and also while the model is being run. An ACT-R warning can be distinguished from a Lisp warning because the ACT-R warnings will

always be printed inside of the Lisp “block comment” character sequence `#|` and `|#` and start with the word “Warning” followed by a colon. Here are some examples of ACT-R warnings:

```
#|Warning: Creating chunk STARTING of default type chunk |#  
#|Warning: A retrieval event has been aborted by a new request |#  
#|Warning: Production TEST already exists and it is being redefined. |#
```

As with Lisp warnings, the recommendation is to make sure that you read all ACT-R warnings displayed and make sure to correct the issue or understand why it is not a serious problem. One thing to be careful of however is that many ACT-R warnings are only displayed when the ACT-R trace is enabled. If you set the `:v` parameter to `nil` you will not see potentially serious warnings. Thus, until you are certain that a model is performing correctly the recommendation is to leave the `:v` parameter enabled, and if you encounter any problems while the model is running with the trace turned off, turning the trace back on can often lead to seeing the warnings that indicate the issue.

Some of the most common ACT-R warnings will be described in more detail in this and later units of the model writing texts. If you do not understand what a particular ACT-R warning means, then one thing you can do to find out more information is search the ACT-R reference manual to find an example with the same or similar warning (things specific to the model like chunk or production names found in the warning would of course have to be omitted in the search). That should indicate which ACT-R command generated the warning and provide more details about it.

## Debugging Example

To show the tools one can use to debug an ACT-R model and describe some of the issues one may encounter when working with ACT-R models we will work through the process of debugging a model which is included with the unit materials. We will start with a model for the task that does not work and then through testing and debugging determine the problems and fix them, showing the ACT-R tools which one can use along the way. This task is going to start the testing and debugging with essentially the whole model written. When writing your own models you may find it easier to perform incremental testing as you go instead of waiting until you have written everything, and the same tools and processes would be applicable then too.

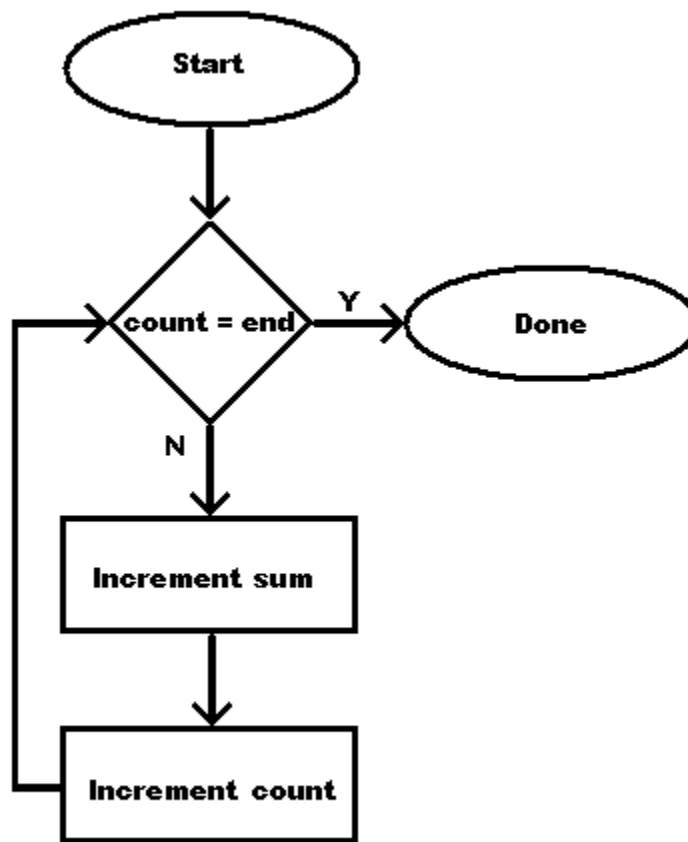
For this unit we will work through a non-working version of the addition by counting task which is in the `broken-addition.lisp` file. Before starting the debugging process however, we will first look at the design of this model because without knowing what it should do we cannot really determine if it is or is not doing the correct thing.

## **Addition Model design**

Before starting to write a model it is useful to start with some design for how you intend the model to work. It does not have to be a complete specification of every step the model will take, but should at least provide a plan for where it starts, the general process it will follow, and what the end condition and results are. As you write the model you may also find it useful to update the design with more details as you go. In that way you will always have a record of how the model works and what it is supposed to do. Below is design information for the addition model provided at increasing levels of detail.

Here is the very general description of the model. This model will add two numbers together by counting up from the first number (incrementally adding one) a number of times indicated by the second number. It does this by retrieving chunks from declarative memory that indicate the ordering of numbers from 0 to 10 and maintaining running totals for the sum and current count in slots of the goal buffer.

Based on that description we can expand upon that a little and create a simple flow chart to indicate the basic process the model will follow:



Another important thing to specify is the way that the information will be encoded for the model, and generally that will involve creating new chunk-types for the task. Here are the new chunk-types which we will use for this model:

The count-order chunk-type will be used to create chunks which encode the sequencing of numbers by indicating the order for a pair of numbers with first preceding second:

```
(chunk-type count-order first second)
```

The add chunk-type will be used to create chunks indicating the goal of adding two numbers and it contains slots for holding the two numbers, the final sum, and the running count as we progress through the additions:

```
(chunk-type add arg1 arg2 sum count)
```

The design of a model should also indicate detailed information about the starting conditions and the expected end state for the model. Here are some detailed descriptions for those aspects of this task:

Start: the model will have a chunk of type add in the goal buffer. That chunk will have the starting number in the arg1 slot, the number to add to it will be in the arg2 slot, and all other slots will be empty.

End: when the model finishes, the value of the sum slot of the chunk in the goal buffer will be the result of adding the number in the arg1 slot to the number in the arg2 slot.

For the model we've created for this task, we will also indicate specific details for what each of the productions we've written is supposed to do. Our model consist of four productions, each corresponding to a state in the flow chart above with the branching test encoded as conditions within the productions for the states that are being branched to.

initialize-addition: (the start state) If the goal is to add two numbers and we do not yet have a sum then set the sum to be the first number, set the count to 0, and make a request to retrieve a chunk to find the next number after that first number.

terminate-addition: (the done state) If the goal is to add two numbers and our count is equal to the second number then stop the model (which we do here by clearing the count slot of the goal).

increment-sum: If the goal is to add two numbers and our current count is not equal to the second number and we have retrieved a count-order chunk for incrementing our current sum then update our sum slot to be the value from the second slot of that count-order and retrieve a count-order chunk to increment the current count.

increment-count: If the goal is to add two numbers and we have retrieved a count-order chunk for incrementing our current count then update our count slot to be the value from the second slot of that count-order and retrieve a count-order chunk to increment the current sum.

Now that we know what the model is supposed to do in detail, we can start testing what has been implemented thus far.

## **Loading the model**

The first step is of course to load the model. However, when we do so we get a Lisp error indicating an end of file (or "eof") occurred which will look something like this:

```
> Error: Unexpected end of file on #<BASIC-FILE-CHARACTER-INPUT-STREAM
```



```
("broken-addition-start.lisp" /904 ISO-8859-1) #xC3AC5DE>, near position 1638
```

An eof error almost always means that there is a missing right parenthesis somewhere in the file, but some other possible causes could be an extra left parenthesis, a missing double-quote character, or an extra double-quote character. To fix this we will have to look at the file, find what is missing or doesn't belong and correct it. If you are using an editor that has built in support for Lisp code, for instance the editor in a Lisp with an IDE, Emacs, or the edit window in the standalone ACT-R Environment, then it shouldn't be too difficult to match parentheses or otherwise locate the issue, but if your editor does not have such capabilities then unfortunately it may be a difficult process to track down the problem. In this case, what we find is that the closing right parenthesis of the define-model call is missing at the very end of the file. After adding that into the file and saving it we should then clear the error from the Lisp and load it again. The load should be successful now, but there are some ACT-R warnings which we should investigate next before trying to run it.

## Initial ACT-R Warnings

Here are the warnings displayed when the model is loaded:

```
#|Warning: No production defined for (INITIALIZE-ADDITION =GOAL> ADD ARG1 =NUM1 ARG2
=NUM2 SUM NIL ==> =GOAL SUM =NUM1 COUNT 0 +RETRIEVAL> ISA COUNT-ORDER FIRST =NUM1). |#
#|Warning: Invalid syntax in =GOAL> condition. |#
#|Warning: First element to define-chunk-spec isn't the symbol ISA. (ADD ARG1 =NUM1
                                ARG2 =NUM2 SUM
                                NIL) |#
#|Warning: --- end of warnings for undefined production INITIALIZE-ADDITION --- |#
#|Warning: No production defined for (TERMINATE-ADDITION =GOAL> ISA ADD COUNT =NUM
ARG2 =NUM2 SUM =ANSWER ==> =GOAL> ISA ADD COUNT NIL). |#
#|Warning: Invalid buffer modification (=GOAL> ISA ADD COUNT NIL). |#
#|Warning: --- end of warnings for undefined production TERMINATE-ADDITION --- |#
#|Warning: Production INCREMENT-SUM already exists and it is being redefined. |#
```

Whenever a model generates ACT-R warnings upon being loaded the next step one takes should be to understand why the model generated those warnings because there is no point in trying to run it unless you know what problems it may have right from the start. Sometimes the warnings indicate a situation that is acceptable to ignore, like the default chunk creation warnings shown in the unit 1 text for the semantic model, but often they indicate something more serious which must be corrected in the model before it can be run.

To determine what the warnings mean one should start reading them from the first warning down because sometimes there may be multiple warnings generated for a single issue. Productions in particular often generate several warnings when there is a problem with creating one. For this model, all of the warnings are related to production issues and we will look at them in detail here to help explain what they mean.

This first warning indicates that the definition of the initialize-addition production, which it shows in the warning, is not valid and thus it could not create that production:

```
#|Warning: No production defined for (INITIALIZE-ADDITION =GOAL> ADD ARG1 =NUM1 ARG2
=NUM2 SUM NIL ==> =GOAL SUM =NUM1 COUNT 0 +RETRIEVAL> ISA COUNT-ORDER FIRST =NUM1). |#
```

Whenever there is a “No production defined” warning there will be more warnings after that which will provide the details about what specifically was wrong with the production. In this case this is the next warning:

```
#|Warning: Invalid syntax in =GOAL> condition. |#
```

It’s telling us that there is something wrong with the =goal> test on the LHS, which may be sufficient to help us fix it, but in fact the next warning is even more specific about what is wrong:

```
#|Warning: First element to define-chunk-spec isn't the symbol ISA. (ADD ARG1 =NUM1
                                                                ARG2 =NUM2 SUM
                                                                NIL) |#
```

That is telling us that the ISA test is missing from the =goal> condition. Often there will be references to low-level ACT-R functions in the most specific warnings, in this case define-chunk-spec, which you may not yet understand. Generally, the meaning of the issue should still be understandable without knowing what that command really does, but if you want more details then you can consult the ACT-R reference manual to find out more information on the low-level functions which are referenced.

The next warning displayed is just an indication that there are no more warnings about the problem in the initialize-addition production:

```
#|Warning: --- end of warnings for undefined production INITIALIZE-ADDITION --- |#
```

At this point one can now either go fix that problem and try loading it again or continue reading through the warnings. For the purposes of this text we are going to continue through all of the warnings first and then fix them afterwards, but some people prefer to fix problems one at a time and would instead stop here and go fix the initialize-addition production before continuing.

The next warning is another indication of a badly formed production, this time it is the terminate-addition production:

```
#|Warning: No production defined for (TERMINATE-ADDITION =GOAL> ISA ADD COUNT =NUM
ARG2 =NUM2 SUM =ANSWER ==> =GOAL> ISA ADD COUNT NIL). |#
```

and again the next warning indicates more information about what exactly is wrong:

```
#|Warning: Invalid buffer modification (=GOAL> ISA ADD COUNT NIL). |#
```

This is telling us that there is something wrong with the =goal> action on the RHS of the production, and then the next warning indicates that there are no more details available for this issue:

```
#|Warning: --- end of warnings for undefined production TERMINATE-ADDITION --- |#
```

The final warning that is displayed is telling us that there are multiple productions with the name increment-sum, and thus the earlier one is being overwritten by a later one:

```
#|Warning: Production INCREMENT-SUM already exists and it is being redefined. |#
```

Now that we've looked over the warnings, all of them are things which need to be fixed before we can run the model and we will address them one at a time in the next section.

## Fixing initialize-addition

Here is the initialize-addition production from the model:

```
(P initialize-addition
  =goal>
    add
    arg1      =num1
    arg2      =num2
    sum       nil
==>
  =goal
    sum       =num1
    count     0
  +retrieval>
    isa       count-order
    first     =num1
)
```

and here are the warnings again:

```
#|Warning: No production defined for (INITIALIZE-ADDITION =GOAL> ADD ARG1 =NUM1 ARG2
=NUM2 SUM NIL ==> =GOAL SUM =NUM1 COUNT 0 +RETRIEVAL> ISA COUNT-ORDER FIRST =NUM1). |#
#|Warning: Invalid syntax in =GOAL> condition. |#
#|Warning: First element to define-chunk-spec isn't the symbol ISA. (ADD ARG1 =NUM1
ARG2 =NUM2 SUM
NIL) |#
#|Warning: --- end of warnings for undefined production INITIALIZE-ADDITION --- |#
```

This is a fairly easy problem to correct and all we need to do is add the missing isa to the goal condition like this:

```
(P initialize-addition
  =goal>
    isa      add
    arg1      =num1
    arg2      =num2
    sum       nil
==>
  =goal
    sum       =num1
    count     0
  +retrieval>
    isa      count-order
    first    =num1
)
```

## Fixing terminate-addition

Here is the text of the terminate-addition production from the model:

```
(P terminate-addition
  =goal>
    ISA      add
    count    =num
    arg2      =num2
    sum       =answer
==>
  =goal>
    isa      add
    count    nil
)
```

and here are the warnings for it again:

```
#|Warning: No production defined for (TERMINATE-ADDITION =GOAL> ISA ADD COUNT =NUM
ARG2 =NUM2 SUM =ANSWER ==> =GOAL> ISA ADD COUNT NIL). |#
#|Warning: Invalid buffer modification (=GOAL> ISA ADD COUNT NIL). |#
#|Warning: --- end of warnings for undefined production TERMINATE-ADDITION --- |#
```

In this case the problem is a common mistake often made when writing modification actions in a production. A buffer modification action in a production specifies the slots of the chunk in the buffer that are to be changed. The chunk-type of that chunk, the isa value, cannot be changed, thus it is not valid to include that in a buffer modification action. If this were a request to a buffer, a + action instead of an = action, then usually we would want to include the isa to indicate the type of action to perform (though there are times when a request can be made without the isa just like a modification). Thus, to fix these warnings all we need to do is remove the incorrect isa specification from the goal buffer action:

```

(P terminate-addition
  =goal>
    ISA      add
    count    =num
    arg2      =num2
    sum       =answer
==>
  =goal>
    count    nil
)

```

## Fixing increment-sum

Here is the final warning to be addressed:

```
#|Warning: Production INCREMENT-SUM already exists and it is being redefined. |#
```

In this case the problem is two productions with the same name. A simple approach would be to just change the name of one of them to clear the warning, but it is better to understand why we have two productions with the same name and if both are indeed valid productions to name them correctly.

Comparing the productions in the model to the design of the task that we created it appears that the second instance of increment-sum is the correct version and that the first one should be increment-count. Something like that may have come about simply as a typo or perhaps by copying-and-pasting increment-sum, since the two productions are very similar, and then failing to change the name on that new one after making other changes to it. Whatever the cause, after changing that production name we are now ready to save the model and try again.

## More Warnings

When we load the model now we see another set of warnings:

```

#|Warning: No production defined for (INITIALIZE-ADDITION =GOAL> ISA ADD ARG1 =NUM1
ARG2 =NUM2 SUM NIL ==> =GOAL SUM =NUM1 COUNT 0 +RETRIEVAL> ISA COUNT-ORDER FIRST
=NUM1). |#
#|Warning: First item on RHS is not a valid command |#
#|Warning: --- end of warnings for undefined production INITIALIZE-ADDITION --- |#

```

Again it is indicating problems with the initialize-addition production. This is an important thing to note about production warnings. No matter how many problems may exist in a production, only one of them will generate warnings at a time because once a problem is detected no further processing of that production will occur. Thus it may take several iterations of addressing the warnings, fixing the production issues, and then reloading before all of the productions are syntactically correct.

This time the warning for initialize-addition indicates that there is a problem with the first action on the RHS of the production, and here again is our updated version of the production:

```
(P initialize-addition
  =goal>
    isa add
      arg1      =num1
      arg2      =num2
      sum       nil
==>
  =goal
    sum      =num1
    count    0
  +retrieval>
    isa      count-order
    first    =num1
)
```

Looking at that closely, we can see that there is a missing “>” symbol at the end of the goal modification request above. We will add that, save the model, and load it yet again. This time the model loads without any warnings. So, now we are ready to start testing the model’s operation.

## Testing

When testing a model one of the important issues is generating meaningful tests, and the design of the model is useful in determining what sorts of things to test. The tests should cover a variety of possible input values to make sure the model is capable of handling all the types of input it is expected to be able to handle. Similarly, tests should be done to make sure that all of the components of the model operate as intended. Thus, if the model has different strategies or choices it can make there should be enough tests to make sure that all of those strategies operate successfully. Similarly, if the model is designed to be capable of detecting and/or correcting for invalid values or unexpected situations one will also want to test a variety of those as well. While it is typically not feasible to test all possible situations, one should test enough of them to feel confident that the model is capable of performing correctly.

Because this model does not have any different strategies or choices nor is it designed to be able to deal with unexpected situations we only really need to generate tests for valid inputs, which are addition problems of non-negative numbers with sums between 0 and 10. Because that is not an extremely large set of options (only 66 possible problems) one could conceivably test all of them, particularly if some Lisp code was written to generate and verify them automatically, but that is usually not the case. Thus, we will treat this as one would a more general task and generate some meaningful test cases to run explicitly.

One way to generate tests would be to just randomly pick a bunch of different addition problems, but a more systematic approach is usually much more useful. When dealing with a known range of possible values, a good place to start is to test values at the beginning and end of the possible range, and starting testing with what seem to be the easiest case is usually a good start. Thus, our first test will be to see if the model can correctly add 0+0.

## The First Run

To do that, we need to create a chunk to place into the goal buffer with those values in it. The model as given already has such a chunk created called test-goal found along with the other chunks created for declarative memory. So, at this point it might seem like a good time to try to run the model, and here is what we get when we do:

```
> (run 10)
0.000    PROCEDURAL          CONFLICT-RESOLUTION
0.000    -----          Stopped because no events left to process
```

Nothing happened. While it may be obvious to you why this model did not do anything at this point, we are still going to walk through the steps that one can take to figure that out. The first step in figuring that out is to determine what you expect should have happened, and having a thorough design can be helpful with that. In this case what should have happened is that the initialize-addition production should fire to start the model along the task.

When one expects a production to fire and it does not, the ACT-R tool that can be used to determine the reason is the whynot command because that will explain why a production did not match the current context. That tool is accessible either by calling the command at the Lisp prompt, or through the procedural viewer in the ACT-R Environment. When using the whynot command one can provide any number of production names along with it (including none). For each of the productions provided it will print out a line indicating whether the production matches or not, and then either the current instantiation of the production if it does match the current context or the production itself along with a reason why it does not match. If no production names are provided then the whynot information will be reported for all productions. To use the tool in the procedural viewer one must highlight a production in the list of productions on the left of the dialog and then press the button labeled “Why not?” on the top left. That will open another window which will contain the same information as displayed by the whynot command.

Because the model stopped at the time when we expected that production to be selected we can use the whynot tool now and find out why it did not fire. If the model had done something else instead or continued on and done other things then we would have to employ some other tool to stop the model at the appropriate time to be able to investigate. We will use that other tool when

solving other problems that arise later. Here are the results of calling the `whynot` command for `initialize-addition`:

```
> (whynot initialize-addition)

Production INITIALIZE-ADDITION does NOT match.
(P INITIALIZE-ADDITION
 =GOAL>
   ISA ADD
   ARG1 =NUM1
   ARG2 =NUM2
   SUM NIL
==>
 =GOAL>
   SUM =NUM1
   COUNT 0
+RETRIEVAL>
   ISA COUNT-ORDER
   FIRST =NUM1
)
It fails because:
The GOAL buffer is empty.
```

It did not fire because the goal buffer is empty. Looking at our model we can see that the goal buffer is empty because we do not call the `goal-focus` command to put the test-goal chunk into the buffer. We need to add this:

```
(goal-focus test-goal)
```

to the model definition. Now, we need to save the model and load it again. There are still no warnings being reported so we should try to run it again.

## The Second Run

Here is what happens when we run it again:

```
CG-USER(5): (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.100 DECLARATIVE RETRIEVED-CHUNK A
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.100 PROCEDURAL PRODUCTION-FIRED TERMINATE-ADDITION
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 ----- Stopped because no events left to process
```

Looking at that trace, it has fired the productions we would expect from our design. First it initializes the addition process, and then it terminates because we have counted all the numbers



that it needed to (which is zero). The next thing to check is to make sure that it performed the changes to the goal buffer chunk as intended to create the appropriate result.

To check the chunk in the goal buffer we can use either the buffer-chunk command from the prompt or the buffer viewer tool in the Environment. For the command, any number of buffer names can be provided (including none). For each buffer provided it will print out the buffer name, the name of the chunk in the buffer, and then all of the slot contents for that chunk. If no buffer names are provided then for every buffer in ACT-R it will print the name of the buffer along with the name of the chunk currently in that buffer. To use the buffer viewer tool one can select a buffer from the list on the left of the dialog and then the details as would be printed by the buffer-chunk command for that buffer will be shown on the right. One may open multiple buffer viewer dialogs if desired, which can be useful when comparing the contents of different buffers.

Here is the output from the buffer-chunk command for the goal buffer:

```
> (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
  ISA ADD
  ARG1  0
  ARG2  0
  SUM   0
  COUNT NIL
```

There we see that the sum slot has the value 0 which is what we expect for 0+0. The model has worked successfully for this test. However, that was a very simple case and we do not yet know if it will actually work when there is counting required, or in fact if it can add zero to other numbers correctly. Thus, we need to perform more tests before we can consider the model to be finished.

## Next Test

For the next test it seems reasonable to verify that it can also add 0 to some other number since that does not involve any more productions than the last test and would be good to know before trying any more involved tasks. To do that we will try the problem 1+0 and to do so we need to change the arg1 value of test-goal from 0 to 1 like this in the model:

```
(test-goal ISA add arg1 1 arg2 0)
```

We need to then save that change and reload the model. Now we will run it again and here is the result of the run and the chunk from the goal buffer:

```

> (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.100 DECLARATIVE RETRIEVED-CHUNK B
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
0.100 PROCEDURAL PRODUCTION-FIRED TERMINATE-ADDITION
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 ----- Stopped because no events left to process

> (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
ISA ADD
ARG1 1
ARG2 0
SUM 1
COUNT NIL

```

Everything looks as we would expect so now it seems reasonable to move on to a test which requires actually adding numbers.

## Test with Addition

Since this is the first test of performing an addition we should again create a simple test, and adding 1+1 seems like a good first step since we know the model can add 0+0 and 1+0 correctly. To do that we again change the chunk test-goal, save and load the model.

Before running it, it would be a good idea to make sure we know what to expect. Given the model design above, we expect to see four productions fire in this order: initialize-addition to get things started, increment-sum to add the first number, increment-count to update the count value, and then terminate-addition since our count will then be equal to 1.

Here is what we get when we run it:

```

> (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.100 DECLARATIVE RETRIEVED-CHUNK B
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
0.100 PROCEDURAL PRODUCTION-FIRED TERMINATE-ADDITION
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 ----- Stopped because no events left to process

```

It does not do what we expected it to. The first production fired as we would expect, but then instead of the increment-sum production firing the terminate-addition production fired and stopped the process just as it did when the model was adding 0. Now we have to determine what caused that problem, and the first step towards doing that is determining when in the model run the first problem occurred.

Typically, the first thing to do is to look at the trace and compare it to the actions we would expect to happen. When doing that it is often helpful to have more detail in the trace so that we see all of the actions that occur in the model. Thus, we would want to set the :trace-detail parameter to high in the model, save it, load it, and then run it again. Here is the trace with the detail level set to high:

```
> (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED INITIALIZE-ADDITION
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.050 PROCEDURAL PRODUCTION-SELECTED TERMINATE-ADDITION
0.050 PROCEDURAL BUFFER-READ-ACTION GOAL
0.100 DECLARATIVE RETRIEVED-CHUNK B
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
0.100 PROCEDURAL PRODUCTION-FIRED TERMINATE-ADDITION
0.100 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 ----- Stopped because no events left to process
```

Reading through that trace the first thing that seems wrong is the selection of terminate-addition at time 0.050 (which doesn't show up in the trace with the default trace-detail level). So, that is where we will investigate further to determine why the problem occurred. For more complicated models reading through the trace may not provide quite as definitive an answer, because there could be situations where everything appears to go as expected but the model still generates a wrong result. In those cases, it may be necessary to add even more detail to the trace by putting !output! actions into the productions to display additional information or to walk through the model one event at a time using the stepper tool of the ACT-R Environment (as we will discuss later) and inspect the buffer contents and module states along the way.

Now that we know the problem seems to be at time 0.050 with the selection of the terminate-addition production the next step is figuring out why it fires at that time. One could start by just looking at the model code and trying to determine why that may have happened, but for the purposes of this exercise we will do a more thorough investigation using the stepper tool because

often one will need to see more information about the current state of the system at that time to determine the problem. [If one does not want to use the stepper tool in the ACT-R Environment there is also a run-step command which can be called instead of run to step through things at the prompt, but we will not describe the use of that here and you should consult the ACT-R 6.0 reference manual for details on using that command instead.] To use the stepper tool it should be opened before running the model, and then when the model is run the stepper will stop the system before every event that will be displayed in the trace (thus the trace-detail setting also controls how detailed the stepping is with the stepper tool). While the stepper has the model paused, it will show the action that will happen next near the top of the stepper display and for some actions it will also show additional details in the windows below that. When the stepper has the system paused, all of the other Environment tools can still be used to inspect the components of the system. Now that we have an idea where the problem occurs we want to get the model to that point and investigate further. So, we should reset the model, open the stepper tool, and then run the model.

To get to the event we are interested in, the production selection at time 0.050 seconds, one could just continually hit the step button until that action is the next one. For this model, since there are not that many actions, that would not be difficult. However, if the problem occurs much later into a run, that may not be a feasible solution. In those situations one will want to take advantage of the “Run Until:” button in the stepper. That can be used to run the model until a particular time, to when a specific production is selected or fired, or an event occurs for a specified module. To select what type of action to run until one must select it using the menu button to the right of the “Run Until:” button, and then one must provide the details of when to stop (a time, production name, or module name) in the entry to the right of that button. For this task, since we are interested in the selection of a production we can use the run until button to make that easier. Thus, we should select production from the menu button, type terminate-addition in the entry box, and then press the “Run Until:” button. Doing that we see the trace printed out up to that point and the stepper now shows that selection event along with details of the production. Our design for this production is that it should stop the model when there is a sum and the count is equal to the second argument, or specifically when the count slot of the chunk in the goal buffer is the same as the arg2 slot of the chunk in the goal buffer. If we look at the chunk in the goal buffer at this time we see that those values are not the same:

```
GOAL: TEST-GOAL-0
TEST-GOAL-0
  ISA  ADD
  ARG1  1
  ARG2  1
  SUM   1
  COUNT 0
```

Thus there is likely something wrong with the terminate-addition production. If we look at the details in the stepper (or the production in the procedural viewer or the model file) then we see that it is binding three different variables to the slots being tested and it is not actually comparing any of them:

```
(P terminate-addition
  =goal>
    ISA      add
    count    =num
    arg2     =num2
    sum      =answer
==>
  =goal>
    count    nil
)
```

So we need to change that so it does the comparison correctly, which means using the same variable for both the count and arg2 tests. If we change the arg2 test to also use =num that will fix the problem. So we should close the stepper, make that change to the model file, save it, reload it, and they try running it again. Of course, we did not necessarily need to go through all of those steps to locate and determine what was wrong because we may have been able to figure that out just from reading the model file, but that is not always the case, particularly for larger and more complex models, so knowing how to work through that process is an important skill to learn.

Before reloading the model, we might also want to change the trace detail level down from high so that it is easier to check if the model does what we expect. Setting it to low will give us a minimal trace, but that should still be sufficient since it will show all the productions that fire. After making that change as well and then reloading here is what the model does when we run it:

```
CG-USER(19): (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.200 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.250 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.300 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.350 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.400 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.450 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.500 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
. . .
9.950 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
10.000 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
10.000 ----- Stopped because time limit reached
```

Now we have a problem where the increment-sum production fires repeatedly. Again, one could go straight to looking at the model code to try to determine what is wrong, but here we will work through a more rigorous process of stepping through the task and using the diagnostic tools that are available.

As before, the first step should be to turn the trace detail back to high so that we can see all of the details. We can run it now and look at the trace, but we don't need all 10 seconds worth since the problem occurs well before the first second is over. So, we will only run the model up to time 0.300 since that is after the first repeat of increment-sum which we know to be a problem:

```
CG-USER(22): (run 0.3)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED INITIALIZE-ADDITION
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.100 DECLARATIVE RETRIEVED-CHUNK B
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 PROCEDURAL PRODUCTION-SELECTED INCREMENT-SUM
0.100 PROCEDURAL BUFFER-READ-ACTION GOAL
0.100 PROCEDURAL BUFFER-READ-ACTION RETRIEVAL
0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.150 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.150 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.150 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.150 DECLARATIVE START-RETRIEVAL
0.150 PROCEDURAL CONFLICT-RESOLUTION
0.200 DECLARATIVE RETRIEVED-CHUNK A
0.200 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.200 PROCEDURAL CONFLICT-RESOLUTION
0.200 PROCEDURAL PRODUCTION-SELECTED INCREMENT-SUM
0.200 PROCEDURAL BUFFER-READ-ACTION GOAL
0.200 PROCEDURAL BUFFER-READ-ACTION RETRIEVAL
0.250 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.250 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.250 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.250 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.250 DECLARATIVE START-RETRIEVAL
0.250 PROCEDURAL CONFLICT-RESOLUTION
0.300 DECLARATIVE RETRIEVED-CHUNK A
0.300 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.300 PROCEDURAL CONFLICT-RESOLUTION
0.300 PROCEDURAL PRODUCTION-SELECTED INCREMENT-SUM
0.300 PROCEDURAL BUFFER-READ-ACTION GOAL
0.300 PROCEDURAL BUFFER-READ-ACTION RETRIEVAL
0.300 ----- Stopped because time limit reached
```

As with the last problem, here again the issue looks to be an incorrect production selection since we expect increment-count to follow increment-sum. Thus, it is the selection at time 0.200 which seems to be in error. That is where we will investigate further using the stepper.

To do so we need to reset the model, open the stepper, and then run it again for at least 0.200 seconds. Again, we could use step to advance to where the problem is, but here again the run until button provides us with shortcuts because we can either advance to the selection of increment-sum or directly to the time we are interested in. This time, we will use the time option to skip ahead to the time at which we notice the problem.

Select time as the run until option using the menu button and enter 0.2 in the entry box. Then press “Run Until:” to skip to the first event which occurs at that time. The event we are interested in is not that first event at that time, so we now need to hit the step button three times to advance to the specific event at time 0.200 we are interested in. Looking at the details of the production-selected event in the stepper there are actually two things worth noting. The first is of course that increment-sum is selected which we do not want, and the other is that increment-count is not listed under the “Possible Productions” section which lists all of the productions which match the current state and could possibly be selected. Thus, while we would expect it to be firing now it does not actually match the current state. Both of those issues will need to be fixed, but first we will correct the issue with increment-sum since that seems more important – there is no point in trying to fix increment-count if increment-sum is still going to fire continuously.

Again, here is where having a thorough design for the model will help us figure out what the problem is since we can compare the production as written to what we intend it to do, but sometimes, particularly while learning how to model with ACT-R, you may not have considered all the possible details in the initial design. Thus, you may have to figure out why the production does not work and adjust your design as well when encountering a problem. Here we will look more at the production itself along with the high-level design instead of just looking at our detailed design specification. The first thing to realize is that since the production is firing again after itself, that means that either its actions are not changing the state of the buffers and/or modules thus it will continue to match or that its conditions are not sensitive to the changes which it makes thus allowing it to continuously match (and of course it is also possible that both of those are true). Here is the production from the model for reference:

```
(P increment-sum
  =goal>
    ISA      add
    sum      =sum
    count    =count
    - arg2   =count
  =retrieval>
```

```

        ISA      count-order
        second   =newsum
==>
=goal>
  sum          =newsum
+retrieval>
  isa          count-order
  first        =count
)

```

We will start by looking at the actions of the production. It modifies the sum slot of the goal to be the next value based on the retrieved chunk, and it requests a retrieval for the chunk corresponding to the current count so that it can be incremented. Those seem to be the correct actions to take for this production and do result in a change to the state of the buffers. Those actions show up in the high detail trace when the model runs, and if we are really concerned we could also step through those actions with the stepper and inspect the buffer contents, but that does not seem necessary at this point. So, now we should look at the conditions of the production, keeping in mind the actions which it makes because testing those appropriately is what the production is apparently missing. Looking at the conditions of this production we see that it tests the sum slot, which is what gets changed in the actions, but it is not actually using that value for anything. Thus, as long as there is any value in that slot this production will fire. Similarly, in the retrieval buffer test of this production there are no constraints on what the chunk in the buffer should look like, only that it be of type-count order and have a value in the second slot. The only real constraint specified in the conditions of this production is that the count slot's value does not match the arg2 slot's value. Thus, we will have to change something in the conditions of this production so that it does not fire again after itself.

Considering our high-level design, it is supposed to fire to increment the sum. Thus, it should only fire when we have retrieved a fact which relates to the sum, but it does not have such a constraint currently. So, we need to add something to it so that it only fires when the retrieved chunk is relevant to the current sum. Given the way the count-order chunks are set up, what we need to test is that the value in the first slot of the chunk in the retrieval buffer matches the value in the sum slot of the chunk in the goal buffer. Adding that constraint to the production like this:

```

(P increment-sum
=goal>
  ISA      add
  sum      =sum
  count    =count
  - arg2    =count
=retrieval>
  ISA      count-order
  first     =sum
  second   =newsum
==>
=goal>
  sum      =newsum
+retrieval>

```



```

        isa      count-order
        first    =count
    )

```

seems like the right thing to do, and we can now save, load, and retest the model.

Here is the trace we get now:

```

CG-USER(38): (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.000 PROCEDURAL CONFLICT-RESOLUTION
0.000 PROCEDURAL PRODUCTION-SELECTED INITIALIZE-ADDITION
0.000 PROCEDURAL BUFFER-READ-ACTION GOAL
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.050 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.050 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.050 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.050 DECLARATIVE START-RETRIEVAL
0.050 PROCEDURAL CONFLICT-RESOLUTION
0.100 DECLARATIVE RETRIEVED-CHUNK B
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
0.100 PROCEDURAL CONFLICT-RESOLUTION
0.100 PROCEDURAL PRODUCTION-SELECTED INCREMENT-SUM
0.100 PROCEDURAL BUFFER-READ-ACTION GOAL
0.100 PROCEDURAL BUFFER-READ-ACTION RETRIEVAL
0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.150 PROCEDURAL MOD-BUFFER-CHUNK GOAL
0.150 PROCEDURAL MODULE-REQUEST RETRIEVAL
0.150 PROCEDURAL CLEAR-BUFFER RETRIEVAL
0.150 DECLARATIVE START-RETRIEVAL
0.150 PROCEDURAL CONFLICT-RESOLUTION
0.200 DECLARATIVE RETRIEVED-CHUNK A
0.200 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.200 PROCEDURAL CONFLICT-RESOLUTION
0.200 ----- Stopped because no events left to process

```

which does not have increment-sum selected and firing again after the first time. So, now we need to determine why increment-count, which we expect to be selected now, is not. Since the model has already stopped where we expect increment-count to be selected we do not need the stepper to get us to that point. All we need to do now is determine why it is not being selected, and to do that we will use the whynot tool, either from the command line or in the procedural viewer. Here is what we get from calling the whynot command for increment-count:

```

> (whynot increment-count)

Production INCREMENT-COUNT does NOT match.
(P INCREMENT-COUNT
 =GOAL>
   ISA ADD
   SUM =SUM
   COUNT =COUNT
 =RETRIEVAL>
   ISA COUNT-ORDER

```

```

        FIRST =SUM
        SECOND =NEWCOUNT
==>
=GOAL>
    COUNT =NEWCOUNT
+RETRIEVAL>
    ISA COUNT-ORDER
    FIRST =SUM
)

```

It fails because:

The value in the FIRST slot of the chunk in the RETRIEVAL buffer does not satisfy the constraints.

It tells us that it does not match and that one reason for that is because of a mismatch on the first slot of the chunk in the retrieval buffer. We can verify that by looking at the production and the contents of the goal and retrieval buffers. The production's constraint on the first slot is that its value must match the value of the sum slot of the chunk in the goal buffer:

```

(P INCREMENT-COUNT
=GOAL>
    ISA ADD
    SUM =SUM
    COUNT =COUNT
=RETRIEVAL>
    ISA COUNT-ORDER
    FIRST =SUM
    SECOND =NEWCOUNT
==>
=GOAL>
    COUNT =NEWCOUNT
+RETRIEVAL>
    ISA COUNT-ORDER
    FIRST =SUM
)

```

here are the chunks in the goal and retrieval buffers:

```

> (buffer-chunk goal retrieval)
GOAL: TEST-GOAL-0
TEST-GOAL-0
  ISA ADD
  ARG1  1
  ARG2  1
  SUM   2
  COUNT 0

RETRIEVAL: A-0 [A]
A-0
  ISA COUNT-ORDER
  FIRST  0
  SECOND 1

```

Looking at that it is indeed true that they do not match. Notice however that the first slot's value from the retrieval buffer does match the count slot's value in the goal buffer. Given that this production is trying to increment the count, that is probably what we should be checking instead

in this production i.e. that we have retrieved a chunk relevant to the current count. Thus, if we change the production to test the count slot's value instead it might fix the problem:

```
(P increment-count
  =goal>
    ISA      add
    sum      =sum
    count    =count
  =retrieval>
    ISA      count-order
    first    =count
    second   =newcount
==>
  =goal>
    count    =newcount
  +retrieval>
    isa      count-order
    first    =sum
)
```

Along with that change we should probably also change the trace-detail back down to low before saving, loading, and running the next test. Here is what we see when running the model again along with the chunk in the goal buffer at the end:

```
> (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.200 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.250 PROCEDURAL PRODUCTION-FIRED INCREMENT-COUNT
0.300 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL C
0.300 PROCEDURAL PRODUCTION-FIRED TERMINATE-ADDITION
0.300 ----- Stopped because no events left to process
```

```
> (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
ISA ADD
ARG1 1
ARG2 1
SUM 2
COUNT NIL
```

The goal shows the correct sum for 1+1 and the model performed the sequence of productions that we would expect.

## Verification

Before going on and performing more new tests, we should consider whether or not the changes that we have recently made will affect any of the other tests which we have already run i.e. 0+0

and 1+0. In both of those cases the terminate-addition production was fired, and we have had to change that to work correctly to perform the addition of 1+1, so it is a little curious that the “broken” production did those tasks correctly. Thus, to be safe we should probably retest at least one of those to make sure that adding zero still works correctly and was not just a fluke. Here is the result of testing 1+0 again:

```
> (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
0.100 PROCEDURAL PRODUCTION-FIRED TERMINATE-ADDITION
0.100 ----- Stopped because no events left to process
```

```
> (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
ISA ADD
ARG1 1
ARG2 0
SUM 1
COUNT NIL
```

Everything looks correct there and given that terminate-addition now works as it was intended we may feel confident enough in the tests so far that we can move on, but if one wants to be cautious, then running the 0+0 test could also be done.

Now that the model has successfully performed three different addition problems we might be tempted to call it complete, but those were all very simple problems and it is supposed to be able to add any numbers from 0-10 which sum to 10 or less. So, we should perform some more tests before consider it done.

## Test of a large sum

Since our early tests were for small sums it would be useful to also test the other end of the range. There are multiple options for numbers which sum to 10, but if we pick 0+10 that will test both the maximum possible sum as well as also testing the largest number of additions it is expected to be able to do. To run the test we again need to change the goal to represent that problem, save it, load it, and then run it. Here are the trace and resulting goal chunk:

```
> (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.200 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.250 PROCEDURAL PRODUCTION-FIRED INCREMENT-COUNT
0.300 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
```

0.350	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
0.400	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL B
0.450	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
0.500	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL C
0.550	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
0.600	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL C
0.650	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
0.700	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL D
0.750	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
0.800	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL D
0.850	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
0.900	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL E
0.950	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.000	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL E
1.050	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.100	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL F
1.150	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.200	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL F
1.250	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.300	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL G
1.350	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.400	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL G
1.450	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.500	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL H
1.550	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.600	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL H
1.650	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.700	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL I
1.750	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.800	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL I
1.850	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.900	DECLARATIVE	RETRIEVAL-FAILURE	
1.900	PROCEDURAL	PRODUCTION-FIRED	TERMINATE-ADDITION
1.900	-----	Stopped because no events left to process	

```

> (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
  ISA ADD
  ARG1  0
  ARG2  10
  SUM   10
  COUNT NIL

```

The goal chunk is correct with a sum of 10, and thus one might think that it was a successful test. However, if we look at the trace more carefully we will see that something is not quite right. Since the count was 10 we would expect to see 10 firings of each of increment-sum and increment-count, but the model only fires each 9 times. So, there is something else wrong in the model, because even though it got the right answer it did not get there the right way. As with all of the other problems, one could just immediately start looking at the model code to try to find the issue, but here again we will walk through a more rigorous approach.

To determine what went wrong along the way we will walk through the model with the stepper and watch the chunks in the goal and retrieval buffers as the model progresses. For this test we can leave the trace-detail at low for a first pass because that will require the least amount of steps

through the task, and only if we do not find a problem at that level will we move it up to a higher level.

Reset the model and open the stepper along with two buffer viewer windows, one for the goal and one for retrieval. Now run the model and start stepping through the actions watching the changes which occur in the two buffers as it goes. Everything starts off well with the sum and count both incrementing by one each time as the model goes along. However, after executing the event at time 1.300 we see something wrong in the retrieval buffer. The count-order fact has a value of 6 in the first slot and 8 in the second slot. If we continue to step through the model's actions we see that increment-sum uses that chunk to incorrectly increment the sum from 6 to 8, and then that chunk is retrieved again and increment-count also skips over the number 7 as it goes. So, we need to correct the g chunk in the model's declarative memory so that it goes from 6 to 7 instead of 6 to 8. Had we only looked at the result in the goal chunk we would not have noticed this problem. We may have caught it with other tests, but when running a test it is best to make sure that it is completely successful before moving on to test other values.

To correct the problem we need to change the chunk g. Looking at the other declarative memory chunks we can see that not only was g skipping over 7, but that there is not even a chunk which indicates 7 precedes 8. So, we will also have to add one for that as well. Although it does not matter for the model, to keep things easier for reading the trace we should probably name the chunk for 7 h and then adjust the names of the chunks for 8 and 9 to i and j respectively. Here is the updated set of chunks which we now have for the model to use:

```
(add-dm
(a ISA count-order first 0 second 1)
(b ISA count-order first 1 second 2)
(c ISA count-order first 2 second 3)
(d ISA count-order first 3 second 4)
(e ISA count-order first 4 second 5)
(f ISA count-order first 5 second 6)
(g ISA count-order first 6 second 7)
(h ISA count-order first 7 second 8)
(i ISA count-order first 8 second 9)
(j ISA count-order first 9 second 10)
(test-goal ISA add arg1 0 arg2 10))
```

If we save that and run it again we get this trace and resulting goal chunk which shows the correct sum and which takes the correct number of steps to get there:

```
> (run 10)
0.000 GOAL SET-BUFFER-CHUNK GOAL TEST-GOAL REQUESTED NIL
0.050 PROCEDURAL PRODUCTION-FIRED INITIALIZE-ADDITION
0.100 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.150 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.200 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL A
0.250 PROCEDURAL PRODUCTION-FIRED INCREMENT-COUNT
0.300 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
0.350 PROCEDURAL PRODUCTION-FIRED INCREMENT-SUM
0.400 DECLARATIVE SET-BUFFER-CHUNK RETRIEVAL B
```

0.450	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
0.500	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL C
0.550	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
0.600	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL C
0.650	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
0.700	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL D
0.750	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
0.800	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL D
0.850	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
0.900	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL E
0.950	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.000	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL E
1.050	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.100	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL F
1.150	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.200	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL F
1.250	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.300	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL G
1.350	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.400	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL G
1.450	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.500	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL H
1.550	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.600	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL H
1.650	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.700	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL I
1.750	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
1.800	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL I
1.850	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
1.900	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL J
1.950	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-SUM
2.000	DECLARATIVE	SET-BUFFER-CHUNK	RETRIEVAL J
2.050	PROCEDURAL	PRODUCTION-FIRED	INCREMENT-COUNT
2.100	DECLARATIVE	RETRIEVAL-FAILURE	
2.100	PROCEDURAL	PRODUCTION-FIRED	TERMINATE-ADDITION
2.100	-----	Stopped because no events left to process	

```

> (buffer-chunk goal)
GOAL: TEST-GOAL-0
TEST-GOAL-0
  ISA ADD
  ARG1  0
  ARG2 10
  SUM   10
  COUNT NIL

```

Now that we have successfully tested the other extreme we may feel more confident that the model works correctly, but we should probably test a few sums in the middle of the range just to be certain before calling it complete. Some values that seem worthwhile for testing would be things like 3+4 since we have recently added a chunk for 7 to make sure it is correct, and similarly 7+1 and 1+7 might be good tests to perform to make sure our new chunk gets used correctly. Another test that may be useful would be 5+5 because it both counts to the maximum sum and checks whether the model works correctly for matching sum and count values.

We will not work through those tests here, but you should perform some of those as well as others that you choose for additional practice in testing and verifying results. In testing the model further you should find a curious situation for some types of addition problems. In those problems the model will produce the correct answer in the intended way, but a thorough inspection will show that it had the possibility to do things wrong along the way. Why it always does the correct thing is beyond the scope of this unit, but issues like that will be addressed in later units.