# An Imperative Alternative to Productions for ACT-R

**Anthony M Harrison (anthony.harrison@nrl.navy.mil)**
U.S. Naval Research Laboratory
Washington, DC, 20375 USA

**Abstract**

As cognitive modeling has matured, so too have its tools. High-level languages are such tools and present a rich opportunity for the acceleration and simplification of model development. Reviewing some of a the major contributors to this area, a new language (Jass) is introduced for building ACT-R models. Jass simplifies and accelerates model development by providing an imperative language that is compiled to production rules. A complex model implemented using this language is detailed.

**Keywords:** cognitive modeling, ACT-R, high-level languages, cognitive architectures

## Introduction

Cognitive modeling allows us to generate high-fidelity models of human behavior for a wide array of applications and research questions. These models are the result of complex, time-consuming development processes. Many researchers and engineers have developed tools and theories in order to simplify, democratize, or otherwise accelerate this development process (e.g., John, Prevas, Salvucci, & Koedinger, 2004). Of particular interest at present is the work on higher-level languages for cognitive modeling. These languages all share a common goal of simplifying the process of modeling by abstracting away some set of low-level features, while retaining or extending functionality. The theoretic commitments and approaches vary, but common across most of the studies in recent years has been the commitment to a *compilation model* (Ritter et al., 2006). In a *compilation model* high-level source code is compiled into production rules to be run on the underlying cognitive architecture. This permits models developed in the high-level language to avoid low-level pain points while still having access to the full explanatory power of the underlying architecture.

To differentiate the various approaches to the compilation model it is worth considering the intended scope of the generated models and the degree of theoretic commitment the language makes. Degree of commitment is the extent to which the language makes modeling commitments for the modeler. For instance, some of these languages provide a *press-button* construct for computer interaction. While the construct is simple enough, it can actually be implemented in many different ways at the lowest level. The language could employ a single implementation, theoretically committing the user of the language to that approach. Alternatively, the language could provide a set of implementations, allowing the user to select a particular commitment. Yet another alternative is to minimize the commitment by not providing the construct at all, leaving the implementation entirely up to the user. These design decisions make theoretic commitments that can obstruct or hinder some theoretic accounts. Generally speaking, the higher the level of abstraction the greater the number and degree of theoretic commitments made. More precisely, all of these languages commit the modeler to a specific goal structure and management which would make it difficult to study alternative accounts for those elements.

The earliest high-level compiled language for ACT-R was ACT-Simple (Salvucci & Lee, 2003). This GOMS-like language took basic primitives (e.g., click, type, speak, think) and mapped them to a fixed set of specific production sequences. Intended for simple, serial, computer-based tasks, the executable models are able to make realistic time-based predictions. The basic primitives used in ACT-Simple are appealing from a user-design evaluation perspective but they are too limiting for the modeling of general tasks. GOMS is an abstract analysis formalism, not a programming language proper, and therefor lacks many of the constructs necessary for general modeling tasks (e.g., error handling). As described in the earlier *press-button* example, ACT-Simple used fixed mappings between behaviors (e.g., *press-button*) and production sequences, committing the user to those particular theoretic accounts.

GOMS to ACT-R (G2A) (St Amant, Freed, & Ritter, 2005) adopts a similar approach as ACT-Simple. Also based on GOMS, G2A inherits some of the challenges therein. It simply lacks many of the concrete formalisms necessary to express arbitrary task and flow structures. While still well suited for simple, computer-based tasks, G2A does vary the primitive-production mapping. Recognizing that multiple production solutions exist for each primitive, G2A allows those mappings to be manipulated at compile time. This low degree of commitment permits G2A to generate families of related models that vary in individual theoretic commitments. It is unclear if the user of the language is able to contribute these primitive-production mappings directly.

The High Level Symbolic Representation (HLSR) language (Jones, Crossman, Lebiere, & Best, 2006) shares many features with Prolog, and brings with it the rich functionality of that programming language. What sets HLSR apart is that the compiler is able to target multiple cognitive architectures, generating productions for both ACT-R and SOAR. This opens the door to performing architectural comparisons while holding the model itself fixed. At such a high-level of abstraction, HLSR has a high degree of commitment, which while enabling greater productivity, ultimately limits the language's ability to provide alternative accounts.

Herbal (Paik, Kim, & Ritter, 2009; Paik et al., 2010) is a graphical language, based on Newell's Problem Space Computational Model (Newell, Yost, Laird, Rosenbloom, & Altmann, 1993). Designed initially for SOAR, it was adapted

to also compile to multiple cognitive architectures, including ACT-R. Similar to HLSR, Herbal is intended as a general modeling language, even though it makes significant theoretical commitments. Herbal's most novel contribution is in the generation of productions at multiple levels of competency. Its production generator can output novice models that rely predominantly upon declarative representations and a few general purpose productions; full expert models where the declarative components have been fully proceduralized; and any mixture of the two endpoints. In order to provide this feature, Herbal commits the user to very specific goal structures and management, as well as specific declarative representations for instructions.

This paper introduces the jACT-R Assembler (Jass)[1], a high-level imperative language that compiles into ACT-R productions for the Java implementation of ACT-R, jACT-R [2]. It aims to simplify development by abstracting away low-level productions, while retaining the full range of ACT-R's explanatory power.

## jACT-R Assembler (Jass)

### Motivation

Jass[3] is intended for modelers specifically familiar with the theoretic underpinnings of ACT-R. It codifies ACT-R's theoretic constraints while providing significant control over how a model's commitments are expressed. The language was developed with three goals in mind: eliminate the production as the unit of development, maximize the modeler's theoretic control by minimizing the language's commitments, and to allow the language to co-evolve with ACT-R.

### Eliminate the Production

Central to this work is the thesis that developing in production rules is itself a hindrance to developing cognitive models. Because of the small granularity of productions, it is often challenging to understand what any one is doing without understanding those that are supposed to fire before and after it as well. This implicit ordering problem makes understanding and developing productions very difficult. Any cognitive resources that can be freed up by not working on productions can be directed towards the development of the model's core theoretic predictions. However, productions lie at the heart of ACT-R as a theory and cognitive architecture. Jass achieves this abstraction by providing an imperative C-style language from which productions are generated for execution by the architecture (see Figure 1).

**Production Generation** Unlike Herbal (Paik et al., 2010), Jass does not to produce novice or expert models, rather something in between, an intermediate model where all the declarative information has been compiled out, but the sequencing and timings have yet to be optimized. The productions generated by Jass are also sparse in that there is at most one instruction per production. This is in contrast to normal, dense, hand-coded productions which often load multiple instructions into a single production. Early work suggest that Jass models have around 3x more productions than hand-coded models. Productions generated by Jass still have room for compilation and optimization by the architecture's production compilation mechanism (Taatgen & Anderson, 2002).

```
/*
 * See something? press button
 */
function void TaskA() {
  slot tmp = null
  //reset the visual system
  request visual(resetVisual)
    =>{
    tmp=null
  }
  // wait for something to be seen
  while(goal(tmpIsNull))
  {
    request visual-location(newVisualLocation)
      -> tmp = visual-location
      =>{
      tmp=null
    }
  }
 // and press a button
  request motor(buttonPress)
    => {
    tmp = null
  }
  return
}
```

Figure 1: Simple perceptual-motor task in Jass.

**Productions for Goal Management** At the heart of all ACT-R models are the productions that manage the current goal. A random sampling of available published models [4] shows that the vast majority of models (90%) use an explicit state representation to control production flow. That is, the goal has a single, perfectly predictive variable devoted to controlling production sequencing, as opposed to controlling sequencing using multiple variables or states. Given that fact, Jass adopts a similar goal structure, with an explicit state variable. However, because goal management is still an area of active research, management is implemented as a pluggable interface. This allows the goal management to be swapped out as necessary so long as there exists a variable devoted to maintaining the explicit state. Jass subsumes goal manage-

[1] https://github.com/amharrison/jass

[2] http://jact-r.org/

[3] Jass is implemented using Eclipse's Xtext language development toolkit. http://eclipse.org/xtext/

[4] http://act-r.psy.cmu.edu/publication/, as of 2/20/20

ment for the modeler by transforming goal management into imperative function-calls. That is, each goal is expressed as a callable function with parameters. In this way, the modeler is freed from managing goal representations themselves and instead just structure function calls to complete the actual goal. Without productions, the language is effectively one of managing the contents of the working memory buffers.

## Theoretic Control

Theoretic control is the ability to express a particular modeled behavior using the full extent of the theoretic framework. Tools with a high degree of theoretic commitment limit one's theoretic control. The aim of Jass was a language designed around the architecture as it is used, hopefully enabling it to avoid many of the theoretical commitments present in prior work. By designing around the architecture we can support or fully codify the five major modeling paradigms in ACT-R (Taatgen, Lebiere, & Anderson, 2006). This allows Jass to exploit ACT-R's full theoretic coverage and not just a subset of it. Two of these paradigms, instance learning and competing strategies, are so pervasive in ACT-R models that they were reified as language constructs.

**Competing Strategies** Production rules are heavily parallel by their very nature. But since ACT-R imposes a serial processing bottleneck, only one of many competing productions can be selected for firing at any given time. This very process accounts for many modeled phenomena in ACT-R. But because of low-level of productions it can be difficult on casual inspection to determine which productions are supposed compete without running the model directly. Jass makes this explicit through the use of the *match-case* statement. Normally Jass's generated productions are strictly serial, following the imperative instruction order. When it reaches the *match-case* statement, it knows to generate a competing production branch for each *case* encountered. Figure 2 shows a snippet that picks between three strategies and falls back to *default* in case none match the current system state.

**Buffer Requests** Instance learning in ACT-R is the retrieval and application of prior problem or goal state information. To achieve this, productions must make a request of the retrieval buffer to fetch from declarative memory some matching pattern (see Figure 3). The pattern of requesting and using information from a particular buffer is so pervasive that most major predictions are derived from the consequences of these requests. As such all request patterns are subsumed by Jass's *request* statement. The behavior of the *request* instruction is determined by the buffer that it is making the request of. Jass includes a contributable meta-definition for buffers that defines their expected behavior. For instance, a buffer can be marked as having a potential error state which will require the *request* instruction to have an error handler. Figure 3 shows a retrieval request (7) of something matching *underspecifiedEpisode* with success and error handlers.

```
...
match{
  case goal(nextIsA) : {
    TaskA()
  }
  case goal(nextIsB) : {
    TaskB()
  }
  case goal(nextIsC) : {
    TaskC()
  }
  case goal(nextIsD) : {
    TaskD()
  }
  default : {
     TaskA()
 }
```

Figure 2: Match-case statement with three alternative branches competing with the default branch. Priorities can be specified using [#] after the case. Function calls denote a change of goal.

**Commitments** The greatest commitment that Jass makes is to the mapping of language constructs to production rules generated. Implemented as a pluggable, extensible interface, new mappings can be swapped in or added if the current commitments are deemed inadequate. The next major commitment is to the goal structure, but as previously mentioned it should be able to handle the majority of models. It too is implemented as a pluggable interface should the goal commitments need to be modified.

## Evolve with Architecture

Cognitive architectures are implementations of evolving theories. To be truly useful, any high-level language must be able to evolve with its underlying architecture. Failing to do so ultimately undermines the tools influence and utility (Ritter et al., 2006). As mentioned previously, Jass uses a pluggable software architecture for all of its major components. This allows goal management and even individual language constructs to be swapped out as theoretical explorations dictate. The language also directly supports the contribution of new modules and buffers through the buffer meta-descriptor (Figure 4). This makes it possible to consolidate the various buffer behaviors into the singular buffer *request* construct discussed earlier.

# Memory for Goals: A Test Case

To gauge the relative success at achieving the design goals of Jass a validation model was implemented. That model should in someway inform each of the design goals discussed previously. Specifically, the elimination of the production should facilitate the development of more complex models; a high

```
underspecifiedEpisode = {
  isa episode
}

...

request retrieval(underspecifiedEpisode)
  -> current = retrieval.reference
  =>{
      ... //error handler
    }
...
}
```

Figure 3: *Request* of retrieval module to fetch a chunk matching *underspecifiedEpisode*. On success, grab the reference. On failure, do something else.

```
goal writable requests * -> *
imaginal writable requests * -> *
retrieval readable requests * -> *
motor error requests motor-command -> ,
                     motor-clear ->
visual readable error
          requests move-attention ->
                   visual-object ,
          clear ->
```

Figure 4: Jass's buffer meta-descriptor specifying writability, potential for error, and the expectations of the *request* statement.

degree of theoretic control should allow the modeler fully exploit the underlying architecture; and it should adapt new theoretic contributions seamlessly. Altmann & Trafton's Memory for Goals models (2007; 2011) fit these requirements.

## Memory for Goals

Memory for goals is a theory of goal management extensively applied to interruptions that accounts for various resumption errors (Trafton et al., 2011) and lags (Altmann & Trafton, 2007). It posits that we rely upon short-lived episodic traces and their retrieval to manage our goals. Resumption errors are due to the inappropriate retrieval of noisy episodes, and lags are due to incrementally rebuilding episodic context after interruption. These features map nicely to the design goals of Jass. First, the model is complex requiring multiple tasks and their interleaving due to interruption. Second, it relies upon unanticipated uses of ACT-R's underlying architecture (i.e., clearing the goal buffer for an interruption) and makes strong predictions about goal usage. Finally, their account makes use of custom episodic module (i.e., a novel theoretic contribution outside of Jass's initial design scope).

**Experimental Task** The original primary task was a complex computer game (Trafton, Altmann, Brock, & Mintz, 2003) that exhibited two primary features. First, the frequency of response was high permitting the collection of numerous samples as recovery interruption recovery progresses. Second, the task is complex enough that it requires some cognitive state for an interruption to disrupt. The interrupting task was a radar-classification task (e.g., Brock, Stroup, & Ballas, 2002) where subjects selected targets and classified them based on simple rules. For every twenty minute block of the primary task, there were twelve randomly distributed interruption phases. Reaction times were recorded for the first ten responses after an interruption resumption. Each participant completed three blocks (early, middle, late) to assess learning. The remaining details can be found in (Altmann & Trafton, 2007).

## Model

Since this modeling endeavour was more of a proof-of-concept than a rigorously validated model, large portions of the primary and interruption tasks were simplified, focusing primarily on the core of memory for goals.

**Modeled Tasks** Three independent Jass libraries were developed, one for each of the interruption, primary, and management tasks. The interruption task was modeled as an exhaustive visual search, followed by some key inputs. The primary task was itself made up of multiple smaller Jass libraries, each designed to be basic perceptual/action tasks strung into a repeating sequence. It was the manager task's job to determine which of the primary tasks to run at any given time. During normal execution, the model alternates between the manager and the next primary task to be executed. On interruption, the working memory buffers were cleared, triggering the interruption task.

The manager encapsulates the majority of Memory for Goal's theoretic account. Under normal conditions, the manager tracks the prior and current tasks. This context allows it to rapidly retrieve the next task in the sequence. Under interrupted conditions, this context is wiped out and the task manager must try to retrieve the most recent episode which contains the tag representing the completed task. Assuming the task was completed, the task manager tries to retrieve the next task in the sequence. With the to-be-completed task known, the manager creates a new episodic encoding and optionally rehearses it, if it is currently rebuilding context. This new episode is then retrieved, relying only upon the spreading activation from the current context for priming. The retrieved episode (possibly incorrect due to noise) is then used to execute the next task.

**Goal Management** While memory for goals makes specific predictions regarding how goals are rehearsed and retrieved for execution, it is silent on the actual form of the goal. Because of this, Jass's default goal management was able to be used without any modification. However, Altmann

& Trafton's use of buffer clearing to model interruptions did require the inclusion of three hand-written productions to deal with the empty goal buffer state.

**Episodic Module** Memory for goals depends upon some form of an episodic module. Altmann & Trafton underspecify this component, choosing instead for a minimal commitment. All this episodic module does is create a unique, time-stamped, chunk with a single reference. This reference can be to anything and in their models it is the task representation of the to-be-completed task. Jass was easily able to accomodate the new module using the buffer meta-descriptions mentioned previously (see Figure 4).

## Results

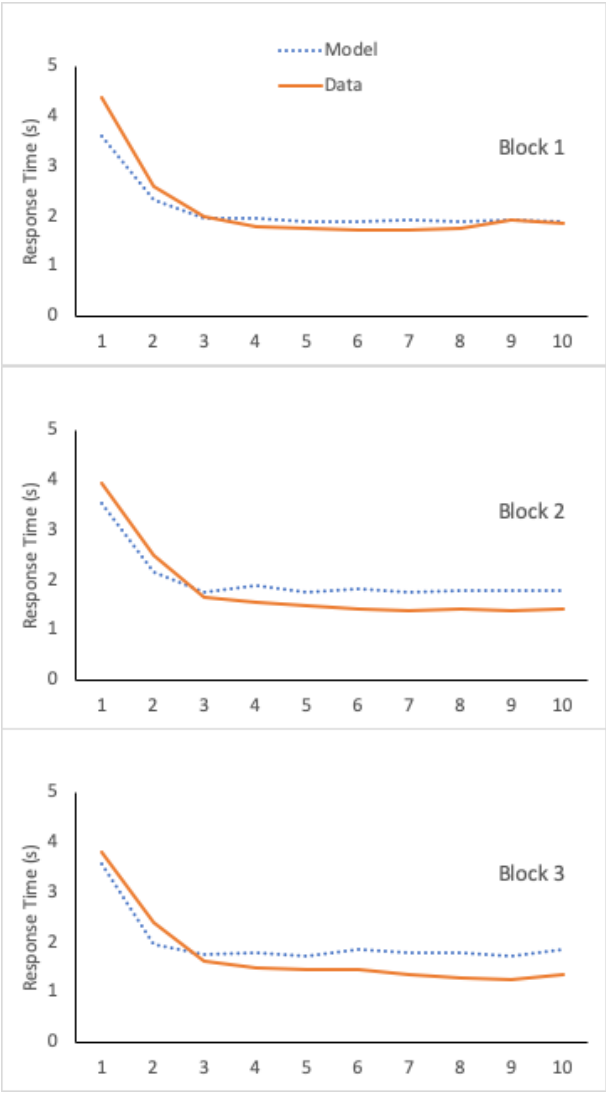


Figure 5: Average response times from the Altmann & Trafton (2007) experiment (solid lines) and average model fits (dotted lines), plotted by block (1-3) and serial position after interruption (1-10).

**Model** The model was run one hundred times with an activation noise of 0.1, all other parameters were set at their defaults. Model and empirical response times are plotted in Figure 5. The model fits well ($RMSE = 0.243$, $R^2 = 0.94$). This shows the primary resumption lag effect as it rebuilds its context after interruption. Unfortunately, the model does not show the same learning effect across blocks as seen in the empirical data. This is due to the relatively lean declarative needs of the model, that is, only the episodes and task tags are retrieved. Had there been task information retrieved during the execution of the primary and interruption tasks, we'd expect to see a greater effect of practice.

While the general pattern of the reaction times is consistent with the empirical findings, the model is consistently slower at the later positions. At this late point, declarative retrievals are effectively immediate, the latency is largely due to the overhead of the productions. Had production compilation been enabled, we'd expect the generated production overhead to be reduced, making up the difference.

**Jass** While too early for a formal study, it is worth considering the anecdotal experience of modeling memory for goals in Jass. From inception to first batch runs was less than four days of engineering time. The coding of the three tasks took less than eight hours, yielding a moderately sized model of 183 productions, approximately 22.5 productions per hour. In terms of lines of code, the Jass models took a combined 592 lines versus the generated productions taking a combined 2945 lines ( 5x more compact).

## Discussion

We successfully demonstrate the use of Jass to develop complex cognitive models for ACT-R. The imperative model simplifies the temporal sequencing of actions required for task completion relative to working with productions directly. The design of Jass allows it to accommodate many different theoretical accounts, even for core elements such as goal management. The design flexibility also permits Jass to adapt to changes in the underlying architecture, allowing it to keep abreast of current theoretical trends. Jass's compilation mechanism effectively creates goal-based libraries of functionality. Each of the modeled tasks was implemented separately and only combined into a single model at run time. This is a promising feature as it applies to model reuse across projects and researchers. As a tool for cognitive modeling, this simple proof-of-concept bodes well for the utility of Jass. However, much more rigorous usability testing is required to get a full sense of the tool's benefits and drawbacks (Ritter et al., 2006).

## Acknowledgments

This work was supported by ONR under funding document N0001420WX00496 awarded to Dr. Laura Hiatt. The views and conclusions contained in this document should not be interpreted as necessarily representing the official policies of the U.S. Navy.

## References


Altmann, E. M., & Trafton, J. G. (2007). Timecourse of recovery from task interruption: Data and a model. *Psychonomic Bulletin & Review*, *14*(6), 1079–1084.

Brock, D., Stroup, J. L., & Ballas, J. A. (2002). Effects of 3d auditory display on dual task performance in a simulated multiscreen watchstation environment. In *Proceedings of the human factors and ergonomics society annual meeting* (Vol. 46, pp. 1570–1573).

John, B. E., Prevas, K., Salvucci, D. D., & Koedinger, K. (2004). Predictive human performance modeling made easy. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 455–462).

Jones, R. M., Crossman, J., Lebiere, C., & Best, B. J. (2006). An abstract language for cognitive modeling. In *Proceedings of the 7th iccm.*

Newell, A., Yost, G. R., Laird, J. E., Rosenbloom, P. S., & Altmann, E. (1993). Formulating the problem space computational model. In *The soar papers (vol. ii) research on integrated intelligence* (pp. 1321–1359).

Paik, J., Kim, J. W., & Ritter, F. E. (2009). A preliminary act-r compiler in herbal. In *Proceedings of iccm-2009-ninth international conference on cognitive modeling* (pp. 466–467).

Paik, J., Kim, J. W., Ritter, F. E., Morgan, J. H., Haynes, S. R., & Cohen, M. A. (2010). Building large learning models with herbal. In *Proceedings of iccm-2010-tenth international conference on cognitive modeling* (pp. 187–192).

Ritter, F. E., Haynes, S. R., Cohen, M., Howes, A., John, B., Best, B., ... Lewis, R. L. (2006). *High-level behavior representation languages revisited* (Tech. Rep.). PENNSYLVANIA STATE UNIV STATE COLLEGE COLL OF INFORMATION SCIENCES AND ....

Salvucci, D. D., & Lee, F. J. (2003). Simple cognitive modeling in a complex cognitive architecture. In *Proceedings of the sigchi conference on human factors in computing systems* (pp. 265–272).

St Amant, R., Freed, A. R., & Ritter, F. E. (2005). Specifying act-r models of user interaction with a goms language. *Cognitive Systems Research*, *6*(1), 71–88.

Taatgen, N. A., & Anderson, J. R. (2002). Why do children learn to say "broke"? a model of learning the past tense without feedback. *Cognition*, *86*(2), 123–155.

Taatgen, N. A., Lebiere, C., & Anderson, J. R. (2006). Modeling paradigms in act-r. *Cognition and multi-agent interaction: From cognitive modeling to social simulation*, 29–52.

Trafton, J. G., Altmann, E. M., Brock, D. P., & Mintz, F. E. (2003). Preparing to resume an interrupted task: Effects of prospective goal encoding and retrospective rehearsal. *International Journal of Human-Computer Studies*, *58*(5), 583–603.

Trafton, J. G., Altmann, E. M., & Ratwani, R. M. (2011). A memory for goals model of sequence errors. *Cognitive Systems Research*, *12*(2), 134–143.