EPiC Series in Computing

Volume 41, 2016, Pages 253–265

GCAI 2016. 2nd Global Conference on Artificial Intelligence

Animating Cognitive Models and Architectures: A Rule-Based Approach

Nada Sharaf[1], Slim Abdennadher[1], Thom Frühwirth[2], and Daniel Gall[2]

[1] The German University in Cairo, Egypt.
{nada.hamed, slim.abdennadher}@guc.edu.eg
[2] Ulm University, Germany.
{thom.fruehwirth, daniel.gall}@uni-ulm.de

Abstract

Computational psychology provides computational models exploring different aspects of cognition. A cognitive architecture includes the basic aspects of any cognitive agent. It consists of different correlated modules. In general, cognitive architectures provide the needed layouts for building intelligent agents. The paper presents a rule-based approach to visually animate the simulations of models done through cognitive architectures. As a proof of concept, simulations through Adaptive Control of Thought-Rational (ACT-R) were animated. ACT-R is a well-known cognitive architecture. It was deployed to create models in different fields including, among others, learning, problem solving and languages.

1 Introduction

The main aim of computational cognitive modeling or computational psychology is to study human cognition. This is done through the development and use of thorough "cognitive models". Such models encode different elements of human cognition using computer programs. Running these models could thus simulate human cognition [26].

Cognitive models are usually simulated through cognitive architectures. Such architectures include the shared aspects of cognitive agents. They include the primary structures and processes of cognition. In order to achieve plausible social and context-aware agents, the development of cognitive architectures is crucial since they support the key goal of building agents with the same capabilities as humans [17, 26, 25].

Adaptive Control of Thought-Rational (ACT-R) is one of the well-known cognitive architectures. ACT-R consists of different interlinked modules. The modular approach provided by ACT-R 6.0 makes sure that information processing for each module is performed separately [2, 1]. ACT-R was able to model a wide range of applications for different types of agents. The fields are not only psychological ones. They include language processing and problem solving. For example, cognitive modeling and ACT-R were deployed with tutoring and mathematics [12, 19, 15]. ACT-R was also used with cognitive social simulation experiments [24, 23].

Unfortunately, one of the problems faced with ACT-R was debugging and tracing. To debug the execution, users need to use primitive methods. This, however, is time-consuming

and difficult especially for big models. ACT-R environment was introduced in order to overcome these problems [3]. It provides graphical views to keep track of the data and steps performed to execute a model. Despite of the graphical window offered to the user, the information viewed is basically text. The user could thus easily lose track of the actual architecture and how the different modules are connected.

The work presented in this paper aims at providing users with a step-by-step animation for the execution of models in ACT-R showing its architecture and structures at each point in time. The basic methodology is based on visualization/annotation rules described later. It should thus be applicable to any modular architecture. It was applied on ACT-R as a proof of the concept. Visualizations have proven to be crucial with programming tools. In general, animation tools were proven to increase the effectiveness of learning. In [14], a meta-study of 24 experimental studies was performed. The performed analysis concluded that such visualizations are educationally effective. The study has shown a significant difference between a group using a visualization technology and a group using none.

Thus, cognitive architectures and models should be no different and animation tools should be provided for them. Such visualization is useful for tracing purposes. However, it could also make it easier to learn about cognitive modeling and ACT-R in general. This is due to the previously mentioned finding that using animations with learning has proved to increase the effectiveness of learning [14]. This thus benefits beginners to computational psychology as well as anyone in need of a visual debugger. Such a visual tracer keeps users aware of what is happening through the architecture at each point. To the best of our knowledge, this is the first attempt for embedding visualization features into ACT-R to produce animations showing the modules of the architecture and the effect of the simulations on them.

The paper is organized as follows: Section 2 gives an introduction about Constraint Handling Rules (CHR). Section 3 introduces the basic modules of ACT-R. In Section 4, the animation methodology is introduced in more details. Conclusions and directions to future work are given in Section 5.

2 Constraint Handling Rules

This section introduces Constraint Handling Rules (CHR) [5, 6, 7]. CHR is a declarative rule-based language. CHR was introduced for writing constraint solvers. However, due to its declarativity and ease-of-use, it has developed into a general purpose language. Simple few CHR rules could be used to perform interesting tasks as shown below. The rules of a CHR program act on constraints in the constraint store. There are two types of constraints: CHR or user-defined constraints and built-in constraints handled by the host language. A CHR rule consists of a head (H), a body (B) and an optional guard (G). The guard is a condition for applying the rule. A rule could also have a name. A head contains CHR constraints only. A guard could only have built-in constraints. The body, on the other side, could contain CHR or built-in constraints. A CHR program consists of a set of "simpagation" rules. A CHR rule is applicable if the store contains constraints matching the constraints in the head of the rule and if the guard of the rule is satisfied [6]. A simpagation rule has the form:

$$optional\_rule\_name \ @ \ H_K \setminus H_R \ \Leftrightarrow \ G \mid B.$$

A simpagation rule has two types of head constraints: H_K and H_R. H_K are the kept head constraints while H_R are the removed ones. On applying a simpagation rule, constraints matching H_R are removed from the constraint store while the ones matching H_K are kept in the store. There are two special types of simpagation rules: simplification and propagation rules. A simplification rule is a simpagation rule with empty H_K while a propagation rule has an empty

H_R. A simplification rule has the form:

$$optional\_rule\_name \ @ \ H_R \ \Leftrightarrow \ G \mid B.$$

Simplification rules replace the head constraints (H_R) with the body constraints (B). In other words, when a simplification rule is applied, the constraints matching the head constraints are removed from the constraint store and the body constraints are added. A propagation rule has the form:

$$optional\_rule\_name \ @ \ H_K \ \Rightarrow \ G \mid B.$$

If a propagation rule is applied, it adds the body constraints (B) to the constraint store. No constraints are thus removed from the store [6].

A wide range of algorithms were implemented through CHR. The following single-ruled program is able to find the smallest number among a set of numbers. The numbers are represented by the constraint `min/1`. Every time the rule `find_min` is applied, two numbers (`A` and `B`) are compared against each other. The guard makes sure that `A` is less than `B`. The smaller number is kept in the constraint store and the larger number is removed from the store. On successive application of this rule, only the smallest number survives.

```
find_min @ min(A) \ min(B) <=> A<B | true.
```

The following example shows the constraint store after each application of the rule for the query `min(6), min(5), min(4)`. The underlined constraints represent the ones that matched the head of the rule. The first application of the rule removes `6` and keeps `5`. The second application removes `5` and keeps only the constraint holding `4` in the constraint store. `4` is indeed the minimum number.

$$\underline{min(6), min(5)}, min(4)$$
$$\downarrow$$
$$\underline{min(5), min(4)}$$
$$\downarrow$$
$$min(4)$$

3 ACT-R

ACT-R [2, 1] is a cognitive architecture used to execute different type of models simulating the human behavior. In order to have comprehensive cognition, ACT-R has different modules that are integrated together. Such modules simulate the different components of the mind that have to work together to reach plausible cognition. ACT-R has different types of buffers holding pieces of information/chunks. A buffer has at a time one chunk of information. A module can only access the contents of a buffer through issuing a request that is handled by the procedural module. ACT-R chooses at each step an applicable production rule for execution [2, 1, 8]. Figure 1 shows the basic ACT-R modules [2, 1, 8]. The rest of the section introduces the basic modules of ACT-R. The work presented in [8, 9, 10, 11] gave a more detailed introduction. Some of the examples given in the section were also introduced in [8].

3.1 Declarative Module

This module holds the information humans are aware of. Its corresponding buffer is referred to as the retrieval buffer. Information is represented as chunks of data. Each chunk has a type. For example, humans are aware of the fact that 1 is less than 2 and that 2 is less than 3. Such information could be represented by a chunk of the type *count_order*. *count_order* has 2 slots namely $first$ representing the smaller number and *second* representing the bigger number. For

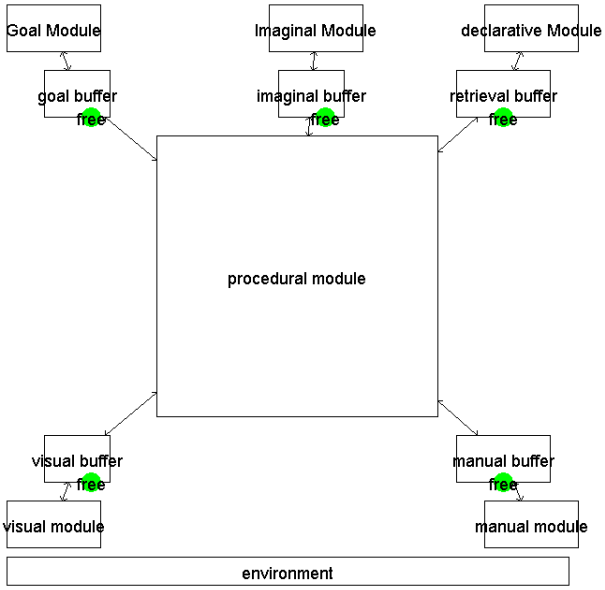


Figure 1: A snapshot of the first panel the user gets. It shows the basic modules of ACT-R as described in [8, 27, 1]

example, the information $(2 < 3)$ would be represented by a *count_order* chunk having 2 in the $first$ slot and 3 in the slot *second*.

3.2 Procedural Module

The procedural module is the central module in ACT-R. It is responsible for firing production rules. A cognitive model consists of different "production rules" with the following format: `(p name condition* ==> action*)`. The name of the rule is represented by *name*. `condition` represents a set of pre-conditions for applying the rule. On executing a rule, its corresponding *action* is performed such as a change in the contents of a buffer or requesting a module to put some data into its buffer. The following is a production rule from a counting model [8].

```
(p start
    =goal>
     ISA count_from
     start =num1
     count nil
    ==>
     =goal>
     count =num1
     +retrieval>
      ISA count_order
      first =num1
)
```

The rule *start* is applied if the goal buffer has a chunk of the type $count\_from$ with the value $num1$ in the slot *start* and *nil* (no value) in the slot *count*. On applying the rule, the *count* slot of the chunk in the goal buffer is changed to hold the value $num1$. In addition, a request is made to the retrieval buffer to get a chunk of the type *count_order* from the declarative

memory. Its $first$ slot should have the same value $num1$. Thus, the rule *start* initializes the counting process by retrieving from the memory a *count_order* chunk whose $first$ slot contains the number that counting should start from.

3.3 Other Components

ACTR-R also has a "goal module" to keep track of the required goal. In addition, different modules such as the manual, visual and the imaginal modules are incorporated to keep track of interactions with the outside world. The symbolic layer of ACT-R is concerned with the production rules, buffers and information chunks. However, the subsymbolic layer is the main drive for ACT-R to simulate human cognition [13]. The subsymbolic layer for example accounts for the time taken to retrieve a chunk relating it to the current context. Chunks related to the current context should take less time to retrieve. In addition, when different production rules are applicable, the subsymbolic layer determines which one to choose through the current context.

4 Animation Methodology

This section introduces the methodology of animating the simulated models. The main aim of the animation is for the user to be able to see at each step in time the changes happening through the different modules of the architecture (ACT-R in this case). The animation has some static visual objects representing the architecture itself showing for example:

- how the different modules are connected
- the buffers associated to each module

Such elements are static since over the course of the animation, they will always have the same look. On the other hand, the animation will have elements changing over time. This includes the available information in the buffers, the queue of actions to be executed, the status of a buffer whether it is busy or free, etc. The idea is to have a generic methodology for animating different cognitive architectures. The proposed idea is to make use of simple structures to animate all the aspects of the architecture. The difficulty emerges from the aim to maintain a generic scheme even with the dynamic units. The proposed solution is to use annotation rules to link every module/static component of the system with a visual object. Once the user starts the animation, these visual objects would appear showing the architecture. Dynamic items, such as data inside a buffer, could also be associated with visual objects. Changes in any of such items should affect the corresponding visual objects. The idea is thus to annotate different parts of the system with visual objects to produce the required animation over time.

To realize an animation of ACT-R simulations, one approach would be to modify any of the existing implementations in such a way that links the system to some visualization library. The implementation should be modified so that every part of the system is associated with a visual object. One of the main goals of the work is to make using cognitive architectures like ACT-R easier. Thus, a different direction was needed; a direction that does not require performing changes to the existing systems. The CHR implementation of ACT-R [8, 9, 10, 11] was used. This implementation uses source-to-source transformation removing the need for any changes. The CHR implementation is also highly adaptable providing an ACT-R implementation using logical rules. Due to the use of source-to-source transformation, the system is still usable by any user whether they are CHR programmers or not.

4.1 Animation Objects

This section introduces how the general animation methodology could be realized through a visualization system with basic visual objects such as circles and rectangles. As a proof of concept the visualization system Jawaa [18] was used. Jawaa provides basic visual objects such as circles and rectangles in addition to some actions on objects. Jawaa was chosen for different reasons. It certainly contains a good range of basic visual objects. Jawaa is based on Java which would also make it easy to integrate with the Java implementation of ACT-R [20]. Lastly, using Jawaa eliminated the need of any modifications as discussed later in Section 4.3.2. To be able to use Jawaa (or another system) to animate ACT-R simulations, some extensions were added. The implemented modifications are listed below.

- **Panels:**
  Instead of showing all animation steps in a single panel, users are now provided with the possibility to have the visualized objects split on different panels. This was done through adding to the parameters of any object a new parameter named *panel*. If this parameter had no supplied value, the default value 1 (the first panel shown to the user) is assumed. Jawaa is now thus able to show at each point only objects related to the currently active panel. An action *changepanel p* was added to switch from the currently active panel to another panel (p). In addition, an action *onclick obj action* which performs a specific *action* on clicking an object (obj) was also added.
- **Queue modifications:**
  Jawaa already had a queue object. Enqueuing was however done in the left end of the queue. A new action, $enqueueafter\ q\ new\_value\ preceding\_value$, was added. $enqueueafter$ adds to the Jawaa queue (q) a new object (new_value). Unlike the default positioning of new elements in a queue, this new element is inserted after a specific existing element determined by the parameter *precedingvalue*.
- **Graphs:**
  Unfortunately, when tested, the graph object did not work in Jawaa 2.0 as stated in `www.cs.duke.edu/csed/jawaa2/commands.html`. The object *vertex* and the action *connectvertices* were thus added. They are similar to the available *node* and *connectnodes* expect for the fact that vertices do not have x and y positions. *connectvertices* connects two vertices with an edge. The edge has a text parameter as well. With graphs, there is always the issue of placing vertices. Since this was a newly added object, a strategy had to be chosen. A force-directed graph algorithm for placing nodes [4] was selected. The positions of the vertices are thus automatically determined based on the force-directed graph algorithm.

4.2 ACT-R using CHR

This section briefly describes how some of the basic components of ACT-R were built through CHR as introduced in [8, 9, 10, 11].

4.2.1 Declarative Module

The chunks are represented with the CHR constraints: $chunk/2$ and $chunk\_has\_slot/3$. $chunk(N, T)$ represents a chunk named N with the type T. For example, $chunk(d, count\_order)$ represents a chunk named d whose type is $count\_order$. On the other hand, $chunk\_has\_slot/3$ represents the values in the slots of a chunk. For example, the two constraints

$chunk\_has\_slot(d, first, 3)$ and $chunk\_has\_slot(d, second, 4)$ represent that chunk d has the values 3 and 4 in the slots $first$ and $second$ respectively. Primitive values (such as $1, 2, etc$) are represented as chunks with the reserved type *chunk*.

4.2.2 Buffer System

The CHR constraint $buffer(B, C)$ represents the fact that buffer B is holding the chunk C. The state S of a buffer B is represented by the constraint $buffer\_has\_state(B, S)$. A buffer has one of three states: either $free$, $busy$ or $error$. The buffer is $busy$ while completing a request. The state of a buffer is set to $error$ if the request was not successful. While executing a model, the various requests sent to the different buffers change their states continuously.

4.2.3 Procedural Module: Timing & Prioritizing Actions in ACT-R

The timing element is an interesting feature of the ACT-R system. It is represented in its CHR implementation by a constraint $now/1$. The implementation has the so-called central scheduling unit which keeps track of the events to be performed and their timings. Some of the available actions in ACT-R such as buffer actions could take some time to be performed. However, once all actions of a rule have started, the procedural module can fire another production rule. The CHR implementation uses a priority queue to keep track of the actions. The scheduler removes the first event in the queue from time to time. Event A precedes B in the queue if A has a timing that is less than the timing of B. If they have the same timings, A precedes B if it has a higher priority. The order of elements in the queue is represented by the constraint $- > /2$. $A - > B$ means that A precedes B. In case the dequeued event is $do\_conflict\_resolution$, the next matching rule is executed and $do\_conflict\_resolution$ is removed. Conflict resolution schedules the firing. Actions of the rule are scheduled according to the current time of the system. At last a $do\_conflict\_resolution$ constraint is added to the queue with the current timing and a low priority. The priority is set to a low value to ensure that actions of the executed rule were performed (corresponding events have been added at the specified time).

4.3 Animating ACT-R

Visual objects could be linked to various parts of the ACT-R system to produce the animation. The architecture itself including the modules and the buffers could be visualized with rectangular objects as shown in Figure 1 [1]. The data inside the buffers could be shown by linking each buffer to a textual object that changes its value over the course of execution. The state of a buffer could be similarly linked to a colored circular object such that the color represents the status.

Each chunk could be shown by a vertex. As stated before, even primitive values are represented as chunks. Consequently a value of a slot of any chunk is basically a connection between one chunk and another chunk. It should thus be visualized by an edge connecting two vertices. The rest of the section shows how the link between ACT-R and the visual library Jawaa was able to produce the needed animations.

4.3.1 Annotation Rules for Animation

Previous work was done to apply the idea of annotating different parts of a CHR program to produce animations of the executions. *CHRAnimation* [21, 22] is a tool that enables animating

[1] This figure shows the first animated panel the user gets.

different CHR programs through annotation rules. Such a generic system removes the need to implement a visual tracer for every type of algorithm. The approach introduced in [21, 22] outsourced the actual visualization to an existing system: Jawaa. *CHRAnimation* introduces annotation rules that bind different components of a CHR program such as constraints with visual objects. Such rules associate a CHR constraint $constraint\_name(Arg_0, \ldots, Arg_n)$ with a Jawaa object or action $Jawaa\_Obj\_Act(par_1, \ldots, par_m)$. Every time a *constraint_name* is added to the constraint store, $Jawaa\_Obj\_Act$ is added to the animation file. An annotation rule could also have a `pre-condition` for application. Each Jawaa object has parameters encoding its visual features such as width, height, color, etc. The annotation rule also determines what values these parameters should have. A value could be:

1. a constant c. It could be a number, a color, etc.
2. the built-in function $valueOf(Arg_i)$ which returns the value of Arg_i.
3. the built-in function $prologValue(Expr)$ where $Expr$ is a *Prolog* expression binding a variable, X, to a value. $prologValue(Expr)$ returns the value bound to X.

An annotation rule has the following form:
$constraint\_name(Arg_0, \ldots, Arg_n) ==> condition\#par1 = val1\#...\#park = valk.$
For example, the rule:
`cons(A)==>true#object=text#name=valueOf(A)#x=10#y=10#text=valueOf(A)#color=blue`
adds for every $cons(A)$ a Jawaa *text* object with an x-coordinate equal to 10 and a y-coordinate equal to 10. The color of the text is blue and the actual string shown is the same as the value of the argument A [2]. The name of the text object is also derived from the value of the argument A. The rule has no precondition for application. Source-to-source transformation was used to generate new CHR programs that are able to communicate with the visual system in order to populate the animation file.

4.3.2 Producing ACT-R Animations

The execution of the model is triggered, in the CHR implementation, by the constraint $run/0$. The constraint $run/0$ has different annotation rules associating it to different rectangular nodes for visualizing the different modules and buffers forming up the ACT-R architecture. The initial animation code produced by the *run* constraint makes sure the initial structure of the architecture is shown. For example, one of the annotation rules produce the following Jawaa code that shows the rectangular node associated with the goal module:
`node goal_module 20 10 100 50 1 "Goal Module" black white black RECTANGLE.`
The result is shown in Figure 1 which is the first panel the user gets once the animation starts. It shows the hierarchy of the modules.

4.3.3 Declarative Module

The declarative module is animated through associating every $chunk(Name, Type)$ constraint with a vertex. The text inside the vertex is the value of `Name` in addition to its `Type` in order to be descriptive. Once the user clicks on the "declarative module" shown in Figure 1, they are transferred to a new panel showing the available chunks in the form of a graph. An edge represents the slot relationship. Thus, the constraint $chunk\_has\_slot(Chunk, Slot, Value)$ is associated with the action *connectvertices* to add an edge between the two vertices, $Chunk$ and $Value$. The edge is labeled with the $Slot$ name. The result of connecting the *vertices* is

[2]The object is considered as a parameter with a constant value.

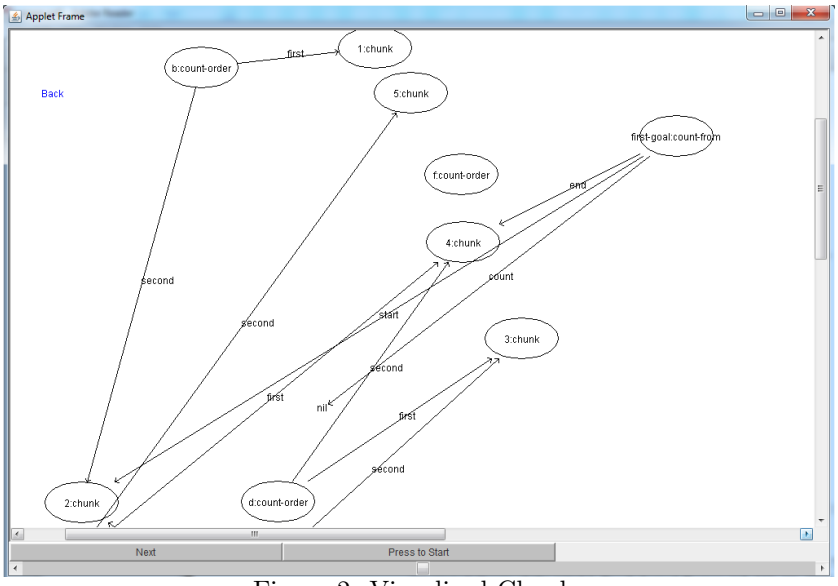


Figure 2: Visualized Chunks

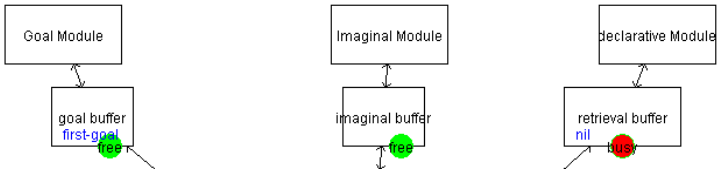
a graph. An example is shown in Figure 2. The chunk labeled `d:count-order` is connected through a line (labeled with `first`) to the chunk labeled with 3. It is also connected through the line labeled `second` to the chunk 4. Figure 2 also shows a *back* link that transfers the user to the basic panel shown in Figure 1.

4.3.4 Buffer System

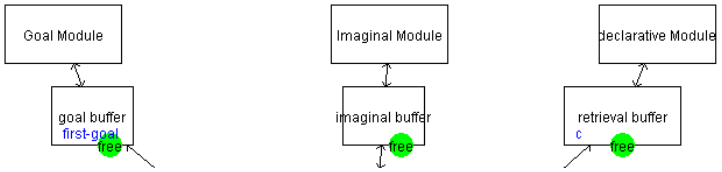
Each buffer has two other elements attached to it: a state and some text. The constraint `run` generates, for each buffer B, a corresponding Jawaa circular *node* named B_state with a color corresponding to its current state. It also generates a Jawaa *text* object named B_text to hold the information inside the buffer. To visualize changes in the states, the constraint $buffer\_state(B, S)$ was associated with annotation rules. The annotation rules update the text of the circular node to be the state S. The rules also change the color of the node to red, green or magenta on having busy, free or error states respectively. These rules first check what the new state is. Accordingly, the color of the Jawaa circular node changes. The information available in the buffer should also be shown to the user. A new annotation rule is thus added to the constraint $buffer(B, C)$ to change the text inside the visualized buffer B to C. Figure 3 shows part of the buffers of ACT-R while executing the counting model. The retrieval buffer starts off by being free. While a request is being processed, its state changes to busy as shown in Figure 3a. As a result of the first request, the chunk `c` is retrieved as shown in Figure 3b. The rest of the figure shows the effect of performing another request to the retrieval buffer.

4.3.5 Scheduling Unit

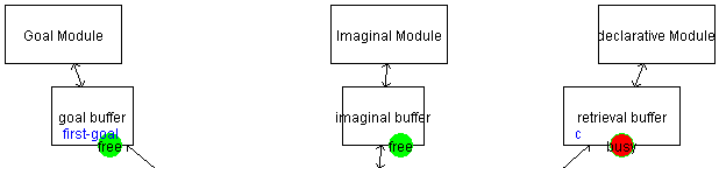
The constraint $run/0$ generates a Jawaa *text* object (`nowText`) initially holding an empty string. The constraint $now(T)$ is associated with a rule that changes the text of `nowText` to *time now is* T. The visualization also shows the events in the priority queue as well. The initialization of the queue is associated with the constraint $run/0$ which automatically



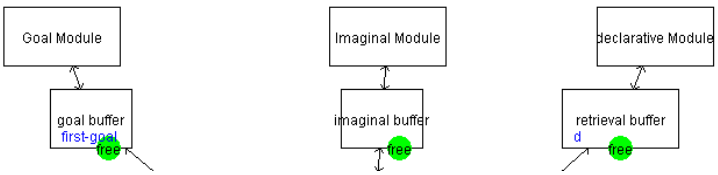
(a) The retrieval buffer is empty. Its state is busy.



(b) As a result of a request made, the retrieval buffer has c and its state is free.



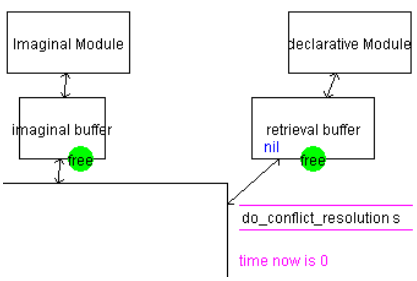
(c) The buffer is performing a request. Its state is busy.



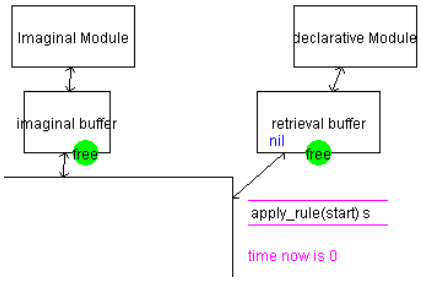
(d) The buffer contains d and it is free again

Figure 3: Buffer System Visualization

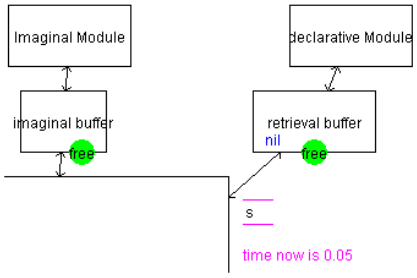
adds a Jawaa queue named $q1$ and enqueues to it the element s (start of the queue). The constraint $F- > q(T, Pr, Evt, ID)$ encodes the fact that the event `Evt` is queued after `F`. Its priority is `Pr`. The time associated it to is `T`. The identifier of this queue element is `ID`. The constraint `->/2` was annotated with an *enqueueafter* action adding *Evt* to the queue after its predecessor F. The text of the queue element is the event itself. Figure 4 shows in more details what happens for the system to change from the state shown in Figure 3a to the state shown in Figure 3b (with chunk `c` in the retrieval buffer) once the rule "start" is being applied. Figure 4a shows the system ready to fire the next matching rule. In Figure 4b, `apply_rule(start)` is added to the buffer since it was the rule chosen to be applied. Execution starts by removing $apply\_rule(start)$ as shown in 4c. Applying "start" adds the events in its action part to the queue. In addition $do\_conflict\_resolution$ is added. The contents of the



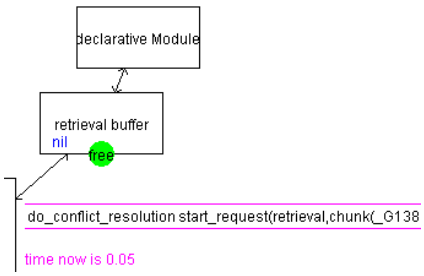
(a)



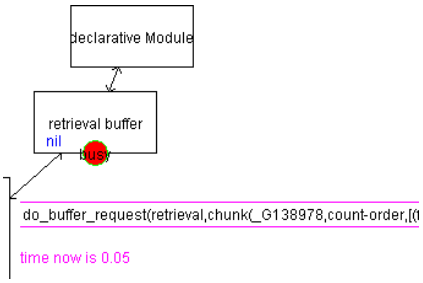
(b) do_conflict_resolution is dequeued. The rule *start* is a matching rule chosen to be applied.



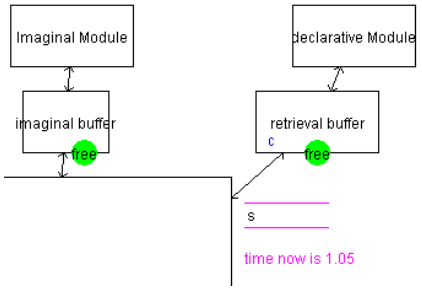
(c) apply_rule(start) is dequeued



(d) The action events of *start* are added to the queue.



(e)



(f)

Figure 4: Animating Scheduling Explained

queue is thus $[do\_conflict\_resolution, start\_request(\ retrieval, chunk(\_G138802, count-order, [(first, 2)])], do\_buffer\_change(goal, chunk(\ \_G137677, \_G137678, [(count, 2)]), s]$ as shown in Figure 4d. As seen from the production rule introduced in Section 3.2, the rule changes the slot *count* of the chunk in the goal buffer to hold the value 2 since in this case, the system tries to count from 2 to 4. It should also put in the retrieval buffer a chunk of the type *count_order* whose *first* slot has the value 2. *do_buffer_change* and *start_request* are dequeued at intermediate steps not shown in the figure. Since the system starts requesting the declarative module, the associated state turns into busy and is highlighted with red. The requests add $do\_buffer\_request(retrieval, chunk\ (\_G138978, count-order, [(first, 2)]))$ to the queue as shown in Figure 4e. Figure 4f shows the module after retrieving chunk *c* which is a *count_order* chunk whose *first* slot is 2. The state of the retrieval buffer is changed to free

changing the color to green. The time of the system is changed to is 1.05 [3].

5 Conclusion & Future Work

The paper presented a rule-based approach to animate the execution of cognitive models in ACT-R. The approach is a generic one that could be used with different applications. Unlike the previous work to visualize cognitive models such as the work shown in [16], the focus is not on the aspects of the cognitive model itself, it is rather on the ACT-R architecture and how it executes such models. Since the architecture is shown at all steps, users can keep track of the big picture. The presented work focused on the abstract and basic aspects of the ACT-R system. In the future, more detailed aspects of the execution could be considered such as the subsymbolic layer concerned with chunk activation. In addition, the different conflict resolution strategies that could be embedded into ACT-R could be included in the animation as well. More details regarding the events in the queue could be also shown.

References

[1] John R. Anderson, Daniel Bothell, Michael D. Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. An Integrated Theory of the Mind. *Psychological Review*, 111(4):1036–1060, 2004.

[2] John R. Anderson and Christian Lebiere. *The Atomic Components of Thought*. Lawrence Erlbaum associates, Mahwah, New Jersey, 1998.

[3] Dan Bothell. ACT-R Environment Manual. http://act-r.psy.cmu.edu/actr6/EnvironmentManual.pdf. Accessed: 2016.

[4] Thomas M. J. Fruchterman and Edward M. Reingold. Graph Drawing by Force-directed Placement. *Softw., Pract. Exper.*, 21(11):1129–1164, 1991.

[5] Thom W. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming. Special Issue on Constraint Logic Programming*, 37(1-3):95 – 138, 1998.

[6] Thom W. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, August 2009.

[7] Thom W. Frühwirth. Constraint Handling Rules - What Else? In Nick Bassiliades, Georg Gottlob, Fariba Sadri, Adrian Paschke, and Dumitru Roman, editors, *Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Proceedings*, volume 9202 of *Lecture Notes in Computer Science*, pages 13–34. Springer, 2015.

[8] Daniel Gall. A Rule-Based Implementation of ACT-R Using Constraint Handling Rules. Master's thesis, University of Ulm, Germany, 2013.

[9] Daniel Gall and Thom W. Frühwirth. Exchanging Conflict Resolution in an Adaptable Implementation of ACT-R. *TPLP*, 14(4-5):525–538, 2014.

[10] Daniel Gall and Thom W. Frühwirth. A Rened Operational Semantics for ACT-R: Investigating the Relations between Different ACT-R Formalizations. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 114–124, 2015.

[11] Daniel Gall and Thom W. Frühwirth. Translation of Cognitive Models from ACT-R to Constraint Handling Rules. In José Júlio Alferes, Leopoldo E. Bertossi, Guido Governatori, Paul Fodor, and Dumitru Roman, editors, *Rule Technologies. Research, Tools, and Applications - 10th International Symposium, RuleML 2016. Proceedings*, volume 9718 of *Lecture Notes in Computer Science*, pages 223–237. Springer, 2016.

[3] Note that some intermediate steps are not shown in Figure 4 for lack of space.

[12] Kevin A Gluck. An ACT-R/PM Model of Algebra Symbolization. In N. Taatgen and J. Aasman, editors, *Proceedings of the Third International Conference on Cognitive Modeling*, pages 134–141. Veenendal, NL: Universal Press, 2000.

[13] Michael E. Hansen, Andrew Lumsdaine, and Robert L. Goldstone. Cognitive Architectures: A Way Forward for the Psychology of Programming. In Gary T. Leavens and Jonathan Edwards, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2012, part of SPLASH '12, Tucson, AZ, USA, October 21-26, 2012*, pages 27–38. ACM, 2012.

[14] Christopher Hundhausen, Sarah Douglas, and John Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, 2002.

[15] Kenneth R. Koedinger and John R. Anderson. Illustrating Principled Design: The Early Evolution of a Cognitive Tutor for Algebra Symbolization. *Interactive Learning Environments*, 5(1):161–179, 1998.

[16] Trent Kriete, Matthew House, Bobby Bodenheimer, and David C. Noelle. NAV: A Tool for Producing Presentation-Quality Animations of Graphical Cognitive Model Dynamics. *Behavior Research Methods*, 37(2):335–339, 2005.

[17] Pat Langley, John E. Laird, and Seth Rogers. Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160, 2009.

[18] Willard C. Pierson and Susan H. Rodger. Web-based Animation of Data Structures Using JAWAA. In John Lewis, Jane Prey, Daniel Joyce, and John Impagliazzo, editors, *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, 1998, Atlanta, Georgia, USA, February 26 - March 1, 1998*, pages 267–271. ACM, 1998.

[19] Steven Ritter, John R. Anderson, Kenneth R. Koedinger, and Albert Corbett. Cognitive Tutor: Applied research in mathematics education. *Psychonomic Bulletin & Review*, 14(2):249–255, 2007.

[20] Dario Salvucci. The Java Simulation & Development Environment. . http://cog.cs.drexel.edu/act-r/.

[21] Nada Sharaf, Slim Abdennadher, and Thom W. Frühwirth. CHRAnimation: An Animation Tool for Constraint Handling Rules. In Maurizio Proietti and Hirohisa Seki, editors, *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014.*, volume 8981 of *Lecture Notes in Computer Science*, pages 92–110. Springer, 2014.

[22] Nada Sharaf, Slim Abdennadher, and Thom W. Frühwirth. Visualization of Constraint Handling Rules. *CoRR*, abs/1405.3793, 2014.

[23] Paul R Smart, Darren P. Richardson, Katia Sycara, and Yuqing Tang. Towards a Cognitively Realistic Computational Model of Team Problem Solving Using ACT-R Agents and the ELICIT Experimentation Framework. In *19th International Command and Control Research Technology Symposium (ICCRTS'14)*, June 2014.

[24] Paul R. Smart and Katia Sycara. Cognitive Social Simulation and Collective Sensemaking: An Approach Using the ACT-R Cognitive Architecture. In *6th International Conference on Advanced Cognitive Technologies and Applications (COGNITIVE14)*, 2014.

[25] Ron Sun. Desiderata for Cognitive Architectures.

[26] Ron Sun. Introduction to Computational Cognitive Modeling. In Ron Sun, editor, *The Cambridge Handbook of Computational Psychology*, pages 3–20. Cambridge University Press, 2008. Cambridge Books Online.

[27] Niels A Taatgen, Christian Lebiere, and John R Anderson. Modeling Paradigms in ACT-R. In R. Sun, editor, *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, pages 29–52. Cambridge University Press, 2006.