An Adaptable Implementation of ACT-R with Refraction in Constraint Handling Rules

Daniel Gall (daniel.gall@uni-ulm.de)
Institute of Software Engineering and Compiler Construction, Ulm University
89069 Ulm, Germany

Thom Frühwirth (thom.fruehwirth@uni-ulm.de)
Institute of Software Engineering and Compiler Construction, Ulm University
89069 Ulm, Germany

Abstract

ACT-R is a popular cognitive architecture. Although its psychological theory is well-investigated, it lacks a formal foundation. This inhibits computational analysis of cognitive models and leads to technical artifacts in ACT-R implementations.

In this paper we present an adaptable implementation of ACT-R derived from our formalization presented in previous work. We show how this formal approach supported by the use of the declarative programming language Constraint Handling Rules (CHR) leads to an implementation of the ACT-R close to the theory while maintaining interoperability. Due to the adaptability of our implementation we are able to extend the conflict resolution strategy of the system by production rule refraction in our implementation easily avoiding the problem of over-programming in some ACT-R models. The use of CHR facilitates the application of analytical methods from the CHR ecosystem paving the way for ACT-R model analysis.

Keywords: ACT-R, Constraint Handling Rules, conflict resolution, refraction, model analysis

Introduction

ACT-R is a popular cognitive architecture with lots of users and application models. The psychological theory is well-investigated and allows for studying of human-behavior by performing experiments both with humans and artificial ACT-R agents. However, from a computational point of view, ACT-R lacks a formal theory of its underlying computational concepts which inhibits analysis of the computational properties of a model. Like in every production rule system, properties like confluence (i.e. the order of rules does not matter for the result) and termination can be important to the soundness of a cognitive model. While confluence may be regained by a conflict resolution mechanism, unwanted or unexpected non-confluence could be the result of a programming error. Confluence analysis can help to detect all rules that could lead to undesired behavior and help the modeler to check the validity of his model.

In this paper, we present our translation of ACT-R models to the language Constraint Handling Rules (CHR). CHR is a declarative rule-based programming language which comes from the field of logic programming (Frühwirth, 2009). Due to the close relation of CHR to logic, a formalization of the ACT-R production rule system can be derived from the translation. This closes the gap between the formalization and execution of ACT-R models simplifying analysis. There are analysis methods and tools for CHR programs, e.g. for analysis of confluence, termination and operational equivalence which can be applied to ACT-R models. In (Gall & Frühwirth, 2014b) we have shown the first steps towards such an analysis toolbox for ACT-R by defining an abstract operational semantics of its procedural system and showing soundness and completeness of our translation with respect to the CHR very abstract semantics (Frühwirth, 2009). This result is crucial for lifting CHR results to ACT-R. Another benefit of our implementation is its adaptability. We exemplify this by adding refraction to the conflict resolution mechanism of ACT-R. Refraction inhibits rules to fire twice on the same state.

The contributions of this paper are a concise description of our adaptable implementation of ACT-R with CHR, the implementation of refraction as the first of its kind in ACT-R and its application to simplify an existing example model.

We concentrate on the symbolic parts of ACT-R in this paper, since we want to explain the formalization of the basic production rule system with a generalized conflict resolution. Nevertheless, in our implementation we have captured various conflict resolution mechanisms (Gall & Frühwirth, 2014a) and the declarative module with its sub-symbolic concepts (Gall, 2013). The restriction is also closer to our abstract semantics used for confluence analysis of ACT-R models: It is necessary to capture all possible transitions non-deterministically to find all transitions that could inhibit confluence. The abstraction can then be augmented by the details making our results applicable to actual ACT-R implementations in a next step.

Constraint Handling Rules

First, we give an informal introduction to the programming language Constraint Handling Rules which is the basis of our implementation. For a detailed description of the language we refer to (Frühwirth, 2009) and (Frühwirth, 1998). CHR programs consist of a set of *rules* operating on a *constraint store* comparable to the working memory in other production systems. Given an initial constraint store, matching rules are applied to the store to exhaustion. The data elements of the store are *(CHR) constraints* which are first-order predicates of the form $c(t_1,\ldots,t_n)$. For instance, `name(robert)`, `age_of(robert,75)` or `b` are constraints. The terms in the arguments of a constraint can also contain variables that are denoted by capital letters, e.g. `X`.

There are three types of rules in CHR – simplification, propagation and simpagation:

```
simplification @ Hr <=> G | B.
propagation    @ Hk ==> G | B.
simpagation    @ Hk \ Hr <=> G | B.
```

H_r, H_k and B are conjunctions of CHR constraints, whereas the so-called guard G consists of a conjunction of simple built-in tests like arithmetic comparisons, syntactic equality etc. comparable to the modifiers `-`, `<`, `>` in ACT-R production rules. Guards are optional and can be omitted.

Simplification rules match the constraints on the left-hand side of the rule with the store binding variables of the rule with the contents of the store. If matching constraints are found and the tests in the guard hold, the matching constraints are removed from the store and replaced with the constraints in the body B. An example simplification rule is `blue, yellow <=> green` modeling the mixture of the colors blue and yellow. If both colors are found in the store, they are replaced by the color green.

In contrast to simplification rules, propagation rules leave the matching constraints in the store and add the body. Simpagation rules are a mixture of both rules: The constraints in H_k are kept in the store, whereas the constraints in H_r are removed.

To execute CHR programs, an initial constraint store is specified by a so-called query and then rules are applied to exhaustion: E.g., a program consisting of the color-mixing rule from above would simplify the query `blue, blue, yellow` to `blue, green`. Since there are no more yellow – blue pairs in the store, the rule is not applicable anymore.

In implementations, CHR rules and queries are applied in textual order: from top to bottom and from left to right. Rules in CHR can be read as logical formulae giving programs a declarative semantics (Frühwirth, 2009). The logical reading of CHR programs together with analysis methods and tools for e.g. confluence or termination makes CHR suitable for program analysis.

The Basic Implementation of ACT-R in CHR

For an introduction of ACT-R we refer to (Anderson et al., 2004; Anderson & Lebiere, 1998; Taatgen, Lebiere, & Anderson, 2006). Our formalization and implementation of ACT-R and hence the following description of the implementation of ACT-R features is based on those sources together with the ACT-R reference manual (Bothell, n.d.).

Our implementation of ACT-R in CHR consists of two parts: a *compiler* and a *runtime environment*. The compiler takes a set of ACT-R production rules and translates it to CHR rules and the runtime environment implements the features necessary to execute the translated CHR rules according to ACT-R.

Basic Translation of ACT-R models to CHR

This section describes the work of the compiler translating ACT-R models to CHR rules. The description first explains how the state of the procedural system of ACT-R can be represented in terms of constraints. Based on these considerations, the actual translation of production rules is described.

States The state of the procedural module in ACT-R is represented by the chunks in the buffers. We can represent a chunk by the following constraints: a constraint `chunk(cname,type)` and for each slot-value pair of the chunk a `chunk_has_slot(cname,slot,value)` constraint. The buffers with their contents are represented by `buffer(bname,cname)`. There are some assumptions on the state, like for instance "There is at most one `chunk` constraint for each chunk name", "There is at most one `buffer` constraint for each buffer", "There is at most one `chunk_has_slot` constraint for each combination of chunk name and slot" and "Slots and types are consistent". Those assumptions follow from our formal description of ACT-R in (Gall, 2013) and (Gall & Frühwirth, 2014a). The assumptions can be checked by some simple CHR rules, if needed. They play an important role for model analysis.

Production Rules We translate each ACT-R rule to a CHR propagation rule, i.e. the tested constraints are left in the store. The buffer tests on the left-hand side of an ACT-R rule consist of a buffer name, usually a chunk type and a set of slot-value pairs. Such a test signifies that the specified buffer holds a chunk of the specified type with specified slot-value pairs. The values of the slot-value pairs can also be variables which are bound to the actual values of the slot in the buffer. The following rule shows the translation of such tests to a set of constraints. Actions are translated to special action constraints described later:

```
buffer(b,C), chunk(C,t), chunk_has_slot(C,s,v), ...
==> action(...), ... .
```

`C` is a fresh variable which will be bound to the name of the chunk in the specified buffer with name `b`. The other conditions expressed by constraints depend on the value of `C`, specifying that `C` has to have the type `t` and respective values in its slots (for each slot-value pair in the original rule, a `chunk_has_slot` constraint is in the head of the CHR rule). If the value of a slot-value pair is a variable it is also translated to a variable in the resulting rule.

The actions on the right-hand side of an ACT-R rule are translated to constraints `buffer_modification(b,CD)` for modifications, `buffer_request(b,CD)` for requests and `buffer_clearing(b)` for clearings. In those constraints `b` stands for the buffer name the action refers to and `CD` is a term `chunk(CName,Type,LSVP)` which describes the chunk defined by the action in the rule. If name `CName` or type `Type` are not defined they have no value (i.e. they remain an unbound variable). `LSVP` is a list of slot-value pairs taken directly from the original rule, e.g. `[(s1,v1), (s2,v2)]`. For each of the three action constraints there is a rule in the run-time environment which actually performs the action described in the next section.

In ACT-R, the procedural module blocks as soon as a rule has been selected to fire. Contrarily, CHR implementations execute the right-hand side of a rule depth-first from left to right. This leads to the effect that after the adding of the first constraint on the right-hand side of a CHR rule, the next rule

might fire directly, before the other constraints are added. The behavior of ACT-R can be modeled by two phases: a *match* phase and an *apply* phase. Those two phases are represented in CHR by a constraints `match`. The presence of this constraint in the store indicates that the procedural module is in the *match* phase. Each ACT-R production rule can be translated to a CHR rule as follows:

buffertests `\ match <=>` *bufferactions*`, match.`

If all buffer tests succeed and the program is in the *match* phase, i.e. a `match` constraint is in the store, then this constraint is removed prohibiting other rules from firing and the actions are performed. At the end, a new *match* constraint is added allowing other rules to fire.

Runtime Environment

The runtime environment is a framework which offers some features needed to actually execute the rules produced by the compiler.

Scheduler ACT-R implementations usually include a scheduling unit which takes track of the points in simulation time when a certain event is executed. Events in this context are triples `(T,P,C)` signifying that the constraint `C` is added to the store at simulation time `T` with priority `P`. In ACT-R, priority decides which event is executed first, if they are due at the same simulation time. The constraints added can be for example the phase controlling constraint `match` or the action of a production rule. The scheduler implements the following interface described in table 1.

Table 1: Interface of the scheduler.

Constraint present	Action
`get_time(T)`	`T` is bound to current simulation time
`add_event(T,P,C)`	event triple `(T,P,C)` is memorized
`next_event`	constraint from event with smallest time (and highest priority) is added to the store

Production Rule Actions In the last section, the production rule actions have been translated to some special constraints. In the following, we describe the rules in the runtime environment which perform the actions specified by these constraints. For details consult (Gall, 2013).

The rule for *buffer modifications* ignores any name or type in the chunk description, since they are not allowed to be modified (Bothell, n.d.). The symbol _ denotes an anonymous variable.

```
buffer(B,C) \ buffer_modification(B,chunk(_,_,SVP))
<=> modify_slots(C,SVP).
```

It adds a constraint `modify_slots` which takes care of the actual modification of the individual slots in the chunk of the specified buffer. The *buffer clearing* action is implemented similarly. However, for the *request* action, the implementation involves a module which returns a result. The modules have their individual constraint stores and simply have implement the following interface: They have a rule which reacts if a constraint `module_request(Request,Result,Time)` is added to their store. `Request` is the chunk description from the original production rule and it is the only argument with a value. The request action then binds the variable `Result` to a chunk description with the result chunk and the variable `Time` to the simulation time it takes to calculate the request. The request action is then implemented as follows:

```
buffer_request(B,Request)
<=> M:module_request(Request,Result,Time),
    get_time(Now),
    add_event(Now+Time,P,replace_chunk(B,Result)).
```

In this code `M` is the module associated with buffer `B`. The `replace_chunk` is a helper constraint which removes the old chunk from the buffer and adds the result chunk to the store. It is scheduled at the moment the request has finished, i.e. that the result of the request is only applied after the module has finished the request.

Modules As described before, a module can be added to the system by writing a new Prolog module which implements a rule reacting on a `module_request` constraints. Such Prolog modules have their own constraint store which does not interfere with the store from the procedural system.

Implementation of Advanced Concepts

By now we have seen how ACT-R rules can be translated to CHR and how the ACT-R framework can be built using CHR. However, we have ignored timings and therefore scheduling. We extend our translation scheme with those considerations and show the ACT-R main cycle. Furthermore, the basic implementation of conflict resolution is described.

Scheduling

To realize scheduling of production rule action, we translate each ACT-R rule to two CHR rule of the following form:

```
buffertests \ match
<=> get_time(Now),
    add_event(Now+0.05, 0,
      apply_rule(rule_name, buffertests).

buffertests \ apply_rule(rule_name, buffertests)
<=> schedule actions now,
    get_time(Now), add_event(Now, -10, match).
```

In the first rule, the `match` constraint is removed to end the match phase. Since in ACT-R the procedural module block for 50 ms simulation time, the rule application is postponed by that time by scheduling an `apply_rule` event 50 ms from the current time with the rule name as argument. Additionally, the buffer tests, i.e. the resulting variable bindings, are memorized in a second argument making sure that the conditions still match at the rule application time.

The second rule can be applied after the scheduler has added the corresponding `apply_rule` constraint and the memorized variable bindings in the buffer test still apply. Then the actions are scheduled at the current time with priorities as defined in (Bothell, n.d.). Finally, a `match` constraint

is scheduled at the current time with low priority to make sure that the actions are performed before the next rule can match. Note that for requests first are only scheduled to be stated at the current simulation time. The resulting changes are applied after a time offset defined by the requested module.

ACT-R Main Cycle

With the scheduler and the rules modified for scheduling, the ACT-R main cycle can be built: First, the initialization code is executed (adding chunks, ...), then the initial time is set to 0 and an initial event `match` is added to the scheduling queue. As soon as there are no more events in the queue, computation is stopped.

Conflict Resolution

The current considerations have ignored the case when more than one production rule is applicable. Like other production rule systems, ACT-R resolves such conflicts by a certain conflict resolution strategy. In the CHR implementation described so far, only the first matching rule will be applied, since CHR tries rules in textual order.

To implement conflict resolution, we exchange the first rule of our translation scheme for production rules by the following rule scheme:

```
buffertests, match
  ==> conflict_set(rule(rule_name, buffertests)).
```

Since we have a propagation rule, the `match` constraint stays in the store, even if a matching rule has been found. Instead of scheduling the rule application directly, the rule is added to the so-called *conflict set* which collects all matching rules by adding a corresponding `conflict_set` constraint memorizing the rule and its matching variable binding.

Since the `match` constraint is still present afterwards, another rule can match again. In the end, the constraint store is filled with `conflict_set` constraints of all matching rules. As a last rule, we remove the `match` constraint and start the *select* phase which selects the rule being applied according to a certain strategy: `match <=> select.`

In the runtime environment, we can simply add a rule which reacts on the presence of a `select` constraint and prunes the conflict set. For instance, the rule could simply select on arbitrary rule and discard all other rules without defining an order on the rules:

```
select, conflict_set(R1) \ conflict_set(R2) <=> true.
select, conflict_set(R)
<=> get_time(Now),
    add_event(Now+0.05, 0, apply_rule(R)).
```

The first of the two rules will remove `conflict_set` constraints repeatedly, as long as there is more than one such constraint in the store. As soon as only one `conflict_set` constraint is left, the second rule can be applied which simply schedules the rule application event of that rule.

In practice, the conflict resolution depends on some properties of the rules. This only needs a slight adjustment of the conflict set pruning rule. For example, production utilities can be taken into account. In (Gall & Frühwirth, 2014a) we have shown that our implementation allows to exchange the conflict resolution mechanism by simply exchanging the reaction on the select constraint. Since the rules for conflict resolution are split into a separate module, it suffices to exchange this file to modify conflict resolution. We refer to the original paper for details.

Implementation of Refraction

In (Young, 2003) the question is raised, if ACT-R should include rule refraction. Refraction is a concept introduced in (McDermott & Forgy, 1977) as a possible conflict resolution strategy for production rule systems. It inhibits production rules from firing twice on the exact same instance.

Young argues that the lack of refraction can lead to (as he calls it) *over-programming*, i.e. determining the order of rule applications in advance. This aspect destroys declarativity of ACT-R models, since it follows a more imperative – i.e. step-by-step/state-by-state – thinking. This seems to be inelegant and leads to the problem that adjustments of one production rule result in changing every production rule (Young, 2003). Furthermore, the question if such state-aware production rules explain human cognition can also be raised. Although it has been discussed in the community, to the best of our knowledge, refraction has not been included in the ACT-R reference implementation by now.

In the following, we describe how refraction can be included with our CHR implementation and exemplify again how easy it is to exchange fundamental parts of our implementation due to the power of CHR and logic programming. First of all, we memorize the instantiation of a rule that is being applied in an `instantiation` constraint. The instantiation of a rule is a list of all constraints with their matching values. Hence, we can use a propagation rule which reacts on the presence of an `apply_rule` constraint: `apply_rule(R) ==> instantiation(R).`

Before the actual conflict set pruning of an arbitrary conflict resolution strategy (as described before), we add a rule which removes a production rule from the conflict set if it is present in an `instantiation` constraint from the store:

```
instantiation(R) \ conflict_set(R) <=> true.
```

With those two rules the basic refraction mechanism is implemented. Rules should fire again on the same instantiation, if there were changes in one of the involved parts of the instantiation. In this case, if the instantiation has been "touched" intermediately, the `instantiation` constraint should be removed from the store to allow the rule to fire again on this instantiation if it occurs again later. For ACT-R this is the case if a modification or a request have changed the content of one of the buffers of the instantiation in the meantime. Hence, we add two rules which detect changes in `buffer` or `chunk_has_slot` constraints:

```
buffer(B,C1) \ instantiation(rule(_,Hk,_))
 <=> member(buffer(B,C2),Hk), C1 \== C2 | true.
chunk_has_slot(C,S,V1) \ instantiation(rule(_,Hk,_))
 <=> member(chunk_has_slot(C,S,V2),Hk), V1 \== V2 |
      true.
```

Those two rules react if new `buffer` or `chunk_has_slot` constraints enter the store. The guard with the `member` check tests if there is a constraint referring to the same buffer or chunk but having different values. In this case obviously a modification or request occurred and hence this particular instantiation can be removed from the history. Note that the use of refraction in our implementation is optional and can be exchanged and even combined with other conflict resolution strategies (Gall & Frühwirth, 2014a).

Evaluation

We show in an example model, how refraction can simplify the rules of a cognitive model and make them less imperative, i.e. defined from state to state. Our example is derived from the semantic model from ACT-R tutorial unit 1 (*The ACT-R 6.0 Tutorial*, 2012). This model implements a taxonomy of some animals and adds information about some of their properties. For example, it categorizes animals in categories like *fish* and *birds*. Additionally, properties like *swims* or *dangerous* are annotated to categories and representatives. We shortly describe the subset of the model which is the objective of our example.

The knowledge is organized as chunks of type `property` with slots `object` for the name of the object, e.g. *shark*, `attribute` for an attribute of the object, e.g. *dangerous* or *category*, and `value` for the value of the attribute, e.g. *true* in case of the *dangerous* attribute of the shark or *fish* for *category*. In the following, we concentrate on chunks with the attribute category. The goal of the model is to judge if a certain object is member of a category. Such a goal is encoded in a chunk of type `is-member` with slots `object` for the object to judge, `category` for the category, and `judgment` for the result of the query encoded by this chunk.

In a first initialization step, the model requests a chunk from declarative memory which refers to the object the judgment refers to and which has the attribute category, since it wants to deduce the membership of the category (and not something about other properties of the objects) (see the rule `initial-retrieve` in figure 1). To judge the membership of an object in a category, the model can verify the membership directly, if the retrieved chunk already contains the information that the object is a member of the queried category (rule `direct-verify`). Otherwise, it can chain through the categories, i.e. take the category of the found object and check if it is a subcategory of the queried category (rule `chain-category`). Note that in our description the rule to deduce failure has been omitted due to space reasons.

The rules are over-programmed in the sense that the judgment slot always gets a value determining the state of the derivation. For instance, in the beginning, the judgment is expected to be `nil` and is then changed to `pending`. This prevents the first rule from firing repeatedly leading to an endless loop. The rules `direct-verify` and `chain-category` check if the judgment is pending in the beginning. This shows the imperative thinking behind such rules: Those two rules are always meant to fire after the initialization. This is ensured by an artificial state slot in the goal chunk.

```
(P initial-retrieve
  =goal>
     ISA      is-member
     object   =obj
     category =cat
     judgment nil
 ==>
  =goal>
     judgment pending
  +retrieval>
     ISA      property
     object   =obj
     attribute category)
```

```
(P direct-verify
  =goal>
     ISA      is-member
     object   =obj
     category =cat
     judgment pending
  =retrieval>
     ISA      property
     object   =obj
     attribute category
     value    =cat
 ==>
  =goal>
     judgment yes)
```

```
(P chain-category
  =goal>
     ISA       is-member     object     =obj1
     category  =cat          judgment   pending
  =retrieval>
     ISA       property      object     =obj1
     attribute category      value      =obj2
                           - value      =cat
 ==>
  =goal>
     object    =obj2
  +retrieval>
     ISA       property      object     =obj2
     attribute category)
```

Figure 1: Rules of the semantic model

With refraction, we can simplify the rules as follows: `direct-verify` and `chain-category` do not have to check for the `judgement` slot in their conditions and the rule `initial-retrieve` is not required to change the judgment slot of the goal in its actions to `pending` because it never fires again on the same goal. You can find the translation of the model to CHR together with commented example derivations with and without refraction on our homepage[1]. It can be seen that without refraction, the model ends in an infinite loop due to the repeated application of the initialization rule. With refraction, the model derives the correct judgments. For the initialization rule it seems legitimate to check for a `nil` judgment, since this is encodes that the goal has not yet been achieved. It could also check for values other than `yes` or `no` or check if the retrieval buffer is empty or does not contain a suitable chunk for the problem. This might further increase the declarativity of the model.

Related Work

As mentioned before, (Young, 2003) has raised the question if production rule refraction should be included in ACT-R. (Lebiere & Best, 2009) refer to this idea and discuss how the lack of refraction can lead to pathological behavior of a model like infinite looping. The authors also point out that the traditional strategies to avoid such behavior are difficult to model

[1] `http://www.uni-ulm.de/?id=59460`

and sometimes also lack cognitive plausibility. The paper also mentions strategies to inhibit repeated retrieval of declarative memory addressing another architectural problem that could be included in our implementation of declarative memory.

ACT-R has also been implemented in Python (Stewart & West, 2006, 2007) and Java (jACT-R, n.d.; Salvucci, n.d.). These implementations do not concentrate on formalization and analysis. We thank the reviewers for pointers to work on the formalization of cognitive modeling in general (Cooper & Fox, 1998; Howes, Vera, Lewis, & McCurdy, 2004). We plan to investigate how those approaches relate to our work.

Conclusion

In this paper we have presented our implementation of ACT-R in CHR including the translation of ACT-R models to CHR rules and the embedding of the basic ACT-R framework in CHR. We then explained the implementation of some more specific concepts of ACT-R like conflict resolution and finally refraction. To the best of our knowledge, our implementation is the first to include refraction in ACT-R.

It can be seen that the fundamental ideas of ACT-R – rules, chunks, scheduling and conflict resolution – can be captured concisely and elegantly in CHR. By the implementation of refraction and the previous work in (Gall & Frühwirth, 2014a), we have shown the adaptability of our implementation.

Due to the conciseness and declarativity of the rules needed to describe the ACT-R in terms of Constraint Handling Rules, a formalization of ACT-R can be derived from our implementation. We have shown parts of this formalization in (Gall, 2013; Gall & Frühwirth, 2014a, 2014b). The formalization together with the analysis tools of the CHR world pave the way for an ACT-R analysis toolbox. In (Gall & Frühwirth, 2014b), we have taken the first steps towards ACT-R analysis by formulating and abstract operational semantics which is sound and complete with respect to the very abstract semantics of CHR.

Although we have concentrated on the procedural system of ACT-R in this particular work, we want to emphasize that we also have implemented other components like the declarative module with concepts like chunk activation. For details, we refer to (Gall, 2013).

For the future, we want to extend our ACT-R implementation by more features known from the ACT-R world. We also want to investigate how CHR analysis tools in detail can be applied to ACT-R models. Additionally, we plan to extend our abstract operational semantics with more details from ACT-R to investigate how they relate. Finally, the transfer of concepts, methods and ideas from the ACT-R production rule system like activation or reinforcement learning based conflict resolution could be included in an extension of CHR.

References

The ACT-R 6.0 tutorial. (2012). Retrieved from `http://act-r.psy.cmu.edu/actr6/units.zip`

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, *111*(4), 1036–1060.

Anderson, J. R., & Lebiere, C. (1998). *The atomic components of thought*. Lawrence Erlbaum Associates, Inc.

Bothell, D. (n.d.). Act-r 6.0 reference manual – working draft [Computer software manual]. Pittsburgh, Pennsylvania 15213.

Cooper, R., & Fox, J. (1998). Cogent: A visual design environment for cognitive modeling. *Behavior Research Methods, Instruments, & Computers*, *30*(4), 553–564.

Frühwirth, T. (1998). Theory and practice of constraint handling rules. *The Journal of Logic Programming*, *37*(1–3), 95–138.

Frühwirth, T. (2009). *Constraint Handling Rules*. Cambridge University Press.

Gall, D. (2013). A rule-based implementation of ACT-R using Constraint Handling Rules. *Master Thesis, Ulm University*.

Gall, D., & Frühwirth, T. (2014a). Exchanging conflict resolution in an adaptable implementation of ACT-R. *Theory and Practice of Logic Programming*, *14*, 525–538.

Gall, D., & Frühwirth, T. (2014b). *A Formal Semantics for the Cognitive Architecture ACT-R.* (Tech. Rep.).

The Homepage of jACT-R. (n.d.). Retrieved from `http://jactr.org/`

Howes, A., Vera, A., Lewis, R. L., & McCurdy, M. (2004). Cognitive constraint modeling: A formal approach to supporting reasoning about behavior. In *Proc. cognitive science society* (pp. 595–600).

Lebiere, C., & Best, B. J. (2009). Balancing long-term reinforcement and short-term inhibition. In *Proceedings of the 31st annual conference of the cognitive science society* (pp. 2378–2383).

McDermott, J., & Forgy, C. (1977, June). Production system conflict resolution strategies. *SIGART Bull.*(63), 37–37.

Salvucci, D. (n.d.). *ACT-R: The Java Simulation & Development Environment – Homepage.* Retrieved from `http://cog.cs.drexel.edu/act-r/`

Stewart, T. C., & West, R. L. (2006). Deconstructing ACT-R. In *Proceedings of the seventh international conference on cognitive modeling* (p. 298303).

Stewart, T. C., & West, R. L. (2007, September). Deconstructing and reconstructing ACT-R: exploring the architectural space. *Cognitive Systems Research*, *8*(3), 227–236.

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

Young, R. M. (2003). Should ACT-R include production refraction? In *Proceedings of 10th Annual ACT-R Workshop.*