A Formal Semantics for the Cognitive Architecture ACT-R

Daniel Gall and Thom Frühwirth

Institute of Software Engineering and Compiler Construction
University of Ulm
89069 Ulm, Germany
{daniel.gall,thom.fruehwirth}@uni-ulm.de

Abstract. The cognitive architecture ACT-R is very popular in cognitive sciences. It merges well-investigated results of psychology to a unified model of cognition. This enables researchers to implement and execute domain-specific cognitive models. ACT-R is implemented as a production rule system. Although its underlying psychological theory has been investigated in many psychological experiments, ACT-R lacks a formal definition from a mathematical-computational point of view.
In this paper, we present a formalization of ACT-R's fundamental concepts including an operational semantics of the core features of its production rule system. The semantics abstracts from technical artifacts of the implementation. Due to its abstract formulation, the semantics is eligible for analysis. To the best of our knowledge, this operational semantics is the first of its kind.
Furthermore, we show a formal translation of ACT-R production rules to Constraint Handling Rules (CHR) and prove soundness and completeness of the translation mechanism according to our operational semantics.

Keywords: computational psychology, cognitive systems, ACT-R, production rule systems, Constraint Handling Rules, operational semantics

1 Introduction

Computational psychology is a field at the interface of psychology and computer science. It explores human cognition by implementing detailed computational models. The models are executable and hence capable of simulating human behavior. This enables researchers to conduct the same experiments with humans and a computational model to verify the behavior of the model. By this procedure, cognitive models are gradually improved. Furthermore, due to their executability, computational models have to be defined precisely. Hence, ambiguities which often appear in verbal-conceptual models can be eliminated.

Cognitive Architectures support the modeling process by bundling well-investigated research results from several disciplines of psychology to a unified theory. Domain-specific models are built upon such cognitive architectures. Ideally, cognitive architectures constrain modeling to only plausible domain-specific models.

Adaptive Control of Thought – Rational (ACT-R) is one of the most popular cognitive architectures [4]. It is implemented as a production rule system. Although its underlying psychological theory is well-investigated and verified in many psychological experiments, ACT-R lacks a formal definition of its production rule system from a mathematical-computational point of view. I.e. the main data structures and the resulting operational semantics suggested by the psychological theory are not defined properly. This led to a reference implementation full of assumptions and technical artifacts beyond the theory making it difficult to overlook. Furthermore, the lack of a formal operational semantics inhibits analysis of the models like termination or confluence analysis.

In this paper, we present a formalization of the fundamental concepts of ACT-R leading to an operational semantics. The semantics abstracts from many details and artifacts of the implementation. Additionally, aspects like time or external modules are ignored to concentrate on the basic state transitions of the ACT-R production rule system. Those abstractions lead to a short and concise definition of the semantics making it suitable for theoretical analysis of the main aspects of ACT-R. Nevertheless, the semantics is still closely related to the general computation process of ACT-R implementations as we exemplify by a simple model.

The formalization of ACT-R relates to the reference manual of the ACT-R production rule system [6]. However, due to the power of logic programming and CHR, an executable CHR version of ACT-R has been developed, which is very close to the formal description of the system. In this paper, we define the translation from ACT-R production rules to CHR rules formally. The translation is closely related to the translation process described informally in [10], but respects the changes necessary to correspond to the abstract semantics. Additionally, it is the first formal description of our translation mechanism which is described formally. Finally, we prove soundness and completeness of our translation according to our abstract operational semantics.

The paper is structured as follows: Section 2, covers the preliminaries. In section 3, we first recapitulate the formalization of the basic notions of ACT-R and then present its abstract operational semantics. The formal translation of ACT-R to Constraint Handling Rules is shown in section 4. Then, the translation is proved to be sound and complete in relation to our abstract operational semantics in section 5. We conclude in section 6.

2 Preliminaries

First, we cover some notational aspects and introduce ACT-R and CHR.

2.1 Notation

We assume some basic notions of first-order logic and logic programming like syntactic equality or unification. Substitution is denoted by $t[x/y]$ where all occurrences of the variable x in the term t are replaced by the variable y. For the sake of brevity, we treat logical conjunctions as multi-sets and vice-versa at some

points. I.e. we use (multi-)set operators on logical conjunctions or multi-sets of terms as conjunctions. We use the relational notation for some functions and for binary relations we use infix notation.

2.2 ACT-R

First of all, we describe ACT-R informally. For a detailed introduction to ACT-R we refer to [4], [12] or [3]. Then we introduce a subset of its syntax and describe its operational semantics informally. The formalization is presented in section 3.

ACT-R is a production rule system which distinguishes two types of knowledge: *declarative knowledge* holding static facts and *procedural knowledge* representing processes which control human cognition. For example, in a model of the game *rock, paper, scissors*, a declarative fact could be "The opponent played scissors", whereas a procedural information could be that a round is won, if we played rock and the opponent played scissors.

Declarative knowledge is represented as *chunks*. Each chunk consists of a symbolic name and labeled slots which hold symbolic values. The values can refer to other chunk names, i.e. chunks can be connected. Hence, a network of chunks can build complex constructs. The names of chunks are just symbols (which are only important for the modeler) but they get a meaning through their connections. For example, there can be chunks that represent numbers and are named *one*, *two*, ... In a model, such chunks could be the internal representation of the concept of numbers 1, 2, ... However, the names do not give them a meaning but are just helpful for the modeler. The meaning in such a model could come from other chunks, that link two number chunks to represent a count fact, for example. Such a count fact has the slots *first* and *second* and e.g. connects the chunks with name *one* and *two* in the *first* and *second* slot. This represents the concept of an ordering between numbers. To compare the concept of chunks with logic programming or Constraint Handling Rules, the names of chunks can be seen as constants (since they are just symbolic values) and the connections between chunks can relate to complex terms. Chunks are typed, i.e. the number and names of the slots provided by a chunk are determined by a type.

As usual for production rule systems, procedural knowledge is represented as rules of the form IF *conditions* THEN *actions*. Conditions match values of chunks, actions modify them.

ACT-R has a modular architecture. For instance, there is a declarative module holding the declarative knowledge or a visual module perceiving the visual field and controlling visual attention. Each module has a set of affiliated buffers which can hold at most one chunk at a time. For example, there is a retrieval buffer which is associated to the declarative module and which holds the last retrieved chunk from the declarative memory.

The procedural system consists of a *procedural memory* with a set of production rules. The conditions of a production rule refer to the contents of the buffers, i.e. they match the values of the chunk's slots.

There are three types of actions whose arguments are encoded as chunks as well: First of all, *buffer modifications* change the content of a buffer, i.e. the

values of some of the slots of a chunk in a buffer. Secondly, the procedural module can state *requests* to external modules which then change the contents of buffers associated with them. Eventually, *buffer clearings* remove the chunk from a buffer. For the sake of brevity, we only regard buffer modifications and requests in this work. Nevertheless, our formalization and translation can be easily extended by other actions [10]. Additionally, to keep our definitions extensible and as general as possible, we refer to an arbitrary set of buffers instead of a concrete instantiation of the theory with defined buffers and modules. In the example section (section 3.4) we show a concrete instantiation of our theory.

Syntax We define the syntax of an ACT-R production rule over a set of symbols $\mathfrak{S}$ as follows. The set of symbols is possibly infinite and contains all the symbols that are valid to name objects in ACT-R (e.g. chunks, slot names or values). ACT-R does not know complex terms like in first order logic or in CHR. Such terms can rather be constructed by chunks that link the primitive symbols to a more complex construct. Additionally, there is a set of variable symbols $\mathfrak{V}$ that is disjoint from $\mathfrak{S}$.

Definition 1 (production rule). *An ACT-R production rule r has the form $LHS \Rightarrow RHS$) where LHS is a set of terms of the form $test(b, SVP)$ where the b values are called* buffers *and SVP is a set of pairs (s, v) where s refers to a* slot *and v refers to a* value*. Such a pair is called* slot-value pair*. Note that b and s must be constants from $\mathfrak{S}$, whereas v can be a variable or constant, i.e. has the domain $\mathfrak{S} \cup \mathfrak{V}$.*

RHS is a set of terms of the form $action(a, b, SVP)$, where $a \in \{mod, req\}$ (denoting either a modification *or a* request*). b again refers to a buffer and SVP is a set of slot-value pairs.*

The function vars takes a set of tests or actions and returns their variables. Note that the following must hold: $vars(RHS) \subseteq vars(LHS)$, i.e. no new variables must be introduced on the right hand side of a rule. Buffers appearing in RHS must also appear in LHS. However, slots on RHS are not required to appear on LHS. The buffers and slots of the RHS are assumed to be pairwise distinct and must refer to slots which are available for the chunk in the modified buffer (i.e. which exist for the chunk). [6].

We can ensure the condition that a modified slot must exist for a chunk by a typing system. An implementation of such a typing system compliant with the ACT-R reference [6] can be found in [10] and [11]. However, for this abstract paper we assume the rules to be valid. Note that we use a representation of production rules as sets of first-order terms which differs from the original ACT-R syntax. This allows for the use of typical set operators in the rest of the paper. It is easy to derive our syntactic representation from original ACT-R rules and vice-versa.

Informal Operational Semantics A production rule as defined in definition 1 is read as follows: The LHS of the rule are conditions matching the contents of

the buffers. I.e. for a condition $test(b, \{(s_1, v_1), (s_2, v_2)\})$ the buffer b is checked for a chunk with the value v_1 in its s_1 slot and the value v_2 in its s_2 slot. If all conditions on the LHS match, the rule can be applied, i.e. the chunks in the buffers are modified according to the specification on the RHS. For an action $action(mod, b, \{(s_1, v_1')\})$ the value in the slot s_1 of the chunk in buffer b is overwritten by the value v_1'. This type of action is called a *modification*. Since the buffers and slots on the RHS are pairwise distinct, there are no conflicting modifications.

A *request* of the form $action(req, b, \{(arg_1, argv_1), (arg_2, argv_2), \dots\})$ states a request to the corresponding module of buffer b. The arguments are defined by slot-value pairs, where the first part of the pair is the name of the argument and the second part its value. The request returns a pair $(c, \{(res_1, resv_1), \dots\})$ which represents a chunk c with corresponding slot-value pairs. This chunk is put into buffer b after the request has finished. Since arguments and result are chunks, the domain of the argument names, values and results is $\mathfrak{S}$.

Running Example: Counting We investigate the first example from the official ACT-R tutorial [1] using our semantics and translation procedure. The model implements the cognitive task of counting by retrieving counting facts from the declarative memory. This method models the way how little children usually learn counting: They know that after one follows two, after two follows three, etc.

Example 1 (production rule). In the following, we define the production rule which counts to the next number. This rule has been derived from the ACT-R tutorial as mentioned above:

$$\begin{aligned} &\{test(goal, \{(count, Num_1)\}),\\ &\ test(retrieval, \{(first, Num_1), (second, Num_2)\})\}\\ \Rightarrow\\ &\{action(mod, goal, \{(count, Num_2)\}),\\ &\ action(req, retrieval, \{(first, Num_2)\})\} \end{aligned}$$

The goal buffer is tested for slot *count* and Num_1 is bound to the value in this slot. The second test checks if there is a chunk in the retrieval buffer with Num_1 in its *first* slot and some number Num_2 in its *second* slot. If the conditions hold, the *goal* buffer is modified such that the *count* slot is updated with Num_2. Then the declarative memory is requested for a chunk which has Num_2 in its first slot.

2.3 Constraint Handling Rules

We recap the syntax and semantics of Constraint Handling Rules (CHR) shortly. For a detailed introduction to the language, we refer to [8].

Syntax We briefly introduce a subset of the abstract CHR syntax as defined in [8]. Constraints are first-order logic predicates of the form $c(t_1, \ldots, t_n)$ where the t values are first-order terms, i.e. function terms or variables. There are two distinct types of constraints: *built-in* and *user-defined* constraints. We constrain the allowed built-in constraints to $true, false$ and the syntactic equality $=$.

Definition 2 (CHR syntax). *A* CHR program P *is a finite set of rules.* Simpagation rules *have the form*

$$r \ @ \ H_k \backslash H_r \Leftrightarrow G \mid B.$$

r *is an optional name of the rule,* H_k *and* H_r *are conjunctions of user-defined constraints (at least one of them is non-empty) called* head constraints*.* G *is a conjunction of built-in constraints and is called the* guard*. Eventually,* B *a conjunction of built-in and user-defined constraints and called the* body *of the rule.*

Operational Semantics The operational semantics of CHR is defined as a state transition system. Hence, we first define the notion of a CHR state and then introduce the so-called very abstract operational semantics of CHR [8] [9].

Definition 3 (CHR state). *A CHR state is a goal, i.e. either true, false, a built-in constraint, a user-defined constraint or a conjunction of goals.*

Definition 4 (head normal form). *A CHR rule is in* head normal form (HNF) *if each argument of a head constraint is a unique variable.*

A CHR rule can be put into HNF by replacing its head arguments t_i with a new variable V_i and adding the equations $V_i = t_i$ to its guard.

The operational semantics of CHR is defined upon a constraint theory $\mathcal{CT}$ which is nonempty, consistent and complete and contains at least an axiomatization of the syntactic equality $=$ together with the built-in constraints *true* and *false*.

Definition 5 (CHR operational semantics). *For CHR constraints* H_k *and* H_r*, built-in constraints* G *and constraints of both types* R *the following transition relation is defined:*

$$(H_k \wedge H_r \wedge G \wedge R) \mapsto_r (H_k \wedge C \wedge B \wedge G \wedge R)$$

if there is an instance with new variables $\bar{x}$ *of a rule* r *in HNF,*

$$r \ @ \ H'_k \ \backslash \ H'_r \Leftrightarrow C \mid B.$$

and $\mathcal{CT} \models \forall (G \rightarrow \exists \bar{x} (C \wedge (H_k = H'_k) \wedge (H_r = H'_r)))$.

I.e., there is a state transition using the rule r, if (a part of) the built-in constraints G of the state imply that the guard holds and the heads the match.

For the successor state, the constraints in H_k are kept, the constraints in H_r are removed and the body constraints are added. Additionally, the state contains the constraints C from the guard. Since the rule is in HNF, the state contains equality constraints from the variable bindings of the matching $H_k = H'_k$ and $H_r = H'_r$.

3 Formalization of the ACT-R Production System

In this section, we formalize the core data structures of ACT-R formally. We follow the definitions from [10].

3.1 Chunk Stores

Intuitively, a chunk store represents a network of chunks. I.e., it contains a set of chunks. Each chunk has a set of slots. In the slots, there are symbols referring either to a name of another chunk (denoting a connection between the two chunks) or primitive elements (i.e. symbols which do not refer to another chunk).

Definition 6 (chunk store). *A* chunk-store *over a set of symbols* $\mathfrak{S}$ *is a tuple* $(\mathbb{C}, \text{HasSlot})$*, where* $\mathbb{C}$ *is a finite set of chunk identifiers.* $\text{HasSlot} : \mathbb{C} \times \mathfrak{S} \to \mathfrak{S}$ *is a partial function which receives a chunk identifier and a symbol referring to a slot. It returns the value of a chunk's slot. If a slot does not have a value,* HasSlot *is undefined (or in relational notation, if chunk* c *does not have a value in its slot* s*, then there is no* v *such that* $(c, s, v) \in \text{HasSlot}$*).*

3.2 Buffer Systems

Buffer systems extend the definition of chunk stores by buffers. Each buffer can hold at most one chunk from its chunk store. This is modeled by the relation Holds in the following definition:

Definition 7 (buffer system). *A* buffer system *with buffers* $\mathbb{B}$ *is a tuple* $(\mathbb{C}; \text{HasSlot}; \text{Holds})$*, where* $\mathbb{B} \subseteq \mathfrak{S}$ *is a finite set of buffer names,* $(\mathbb{C}, \text{HasSlot})$ *is a chunk-store and* $\text{Holds} : \mathbb{B} \to \mathbb{C}$ *a partial function that assigns every buffer at most one chunk that it holds. Buffers that do not appear in the* Holds *relation are called* empty*.*

3.3 The Operational Semantics of ACT-R

A main contribution of this work is the formal definition of an abstract operational semantics of ACT-R which is suitable for analysis. The semantics abstracts from details like timings, latencies and conflict resolution but introduces non-determinism to cover those aspects. This has the advantage that analysis is simplified since the details like timings are difficult to analyze and secondly to let those details exchangeable. For instance, there are different conflict resolution mechanisms for ACT-R which are interchangeable at least in our implementation of ACT-R as we have shown in [10]. However, for confluence analysis for example, the used conflict resolution mechanism does not matter since conflicts are resolved by some method. At some points though, we do not want to introduce rule conflicts and they are regarded as a serious error. An operational semantics making analysis possible to detect such conflicts in advance is capable of improving and simplifying

the modeling process which is one of the goals of a cognitive architecture like ACT-R.

We define the operational semantics of ACT-R as a state transition system $(\mathbb{S}, \rightarrowtail)$. The state space $\mathbb{S}$ consists of states defined as follows:

Definition 8 (ACT-R states). $S := \langle \mathbb{C}; \text{HasSlot}; \text{Holds}; \mathbb{R} \rangle^{\mathcal{V}}$ *is called an* ACT-R state. *Thereby,* $(\mathbb{C}, \text{HasSlot}, \text{Holds})$ *form a buffer system of buffers* $\mathbb{B}$, $\mathcal{V}$ *is a set of variable bindings and* $\mathbb{R}$ *(the set of* pending requests*) is a subset of tuples* $\mathbb{B} \times 2^{\mathfrak{S} \times \mathfrak{S}}$*, i.e. tuples of the form* (b, SVP) *where* $b \in \mathbb{B}$ *and* SVP *is a set of slot-value pairs.*Initial states *are states where* $\mathbb{R} = \emptyset$*.*

Before we define the transitions $\rightarrowtail$, we introduce the notion of a holding buffer test and consequently a matching l.h.s. of a production rule in a state.

Definition 9 (buffer test). *A buffer test* t *of the form* $test(b, SVP)$ holds *in state* $S := \langle \mathbb{C}; \text{HasSlot}; \text{Holds}; \mathbb{R} \rangle^{\mathcal{V}}$*, written* $t \mathrel{\widehat{=}} S$*, if* $\exists b^S \in \mathbb{B}, c^S \in \mathbb{C}$ *such that the variable bindings* $\mathcal{V}$ *of the state imply that* $b^S = b$, $\text{Holds}(b^S) = c^S$ *and* $\forall (s, v) \in SVP \;\; \exists s^S, v^S : (c^S, s^S, v^S) \in \text{HasSlot}$ *with* $s^S = s$ *and* $v^S = v$*.*

Definition 10 (matching). *A set* T *of buffer tests* matches *a state* S*, written* $T \mathrel{\widehat{=}} S$*, if all buffer tests in* T *hold in* S*.*

We define the following functions which simplify notations in the definition of the operational semantics. Since the behavior of a rule depends on the fact if a certain slot is modified or requested on r.h.s. of the rule, we introduce two functions to test this:

Definition 11 (modified and requested slots). *For an ACT-R rule* r *the following functions are defined as follows:*

–
$$modified_r(b, s) = \begin{cases} true & if \quad \exists action(mod, b, SVP) \in RHS(r) \\ & \quad \wedge \exists v : (s, v) \in SVP \\ false & otherwise \end{cases}$$

–
$$requested_r(b) = \begin{cases} true & if\, \exists action(req, b, SVP) \in RHS(r) \\ false & otherwise \end{cases}$$

With the two functions from definition 11, it can be tested, if a certain buffer is modified (in a certain slot) or requested. As a next step, we regard the actions of a production rule. An action adds or deletes information from the state. The following definition covers these aspects:

Definition 12 (add and delete lists). *For an ACT-R rule* r *and a state* S*, we define the following sets:*

$$\begin{aligned} mod\_add_S(r) =& \{(c, s, v) \in \text{HasSlot} \mid (b, c) \in \text{Holds} \\ & \wedge action(mod, b, SVP) \in RHS(r) \wedge (s, v) \in SVP\} \\ mod\_del_S(r) =& \{(c, s, v) \in \text{HasSlot} \mid (b, c) \in \text{Holds} \wedge modified_r(b)\} \\ req\_del_S(r) =& \{(b, c) \in \text{Holds} \mid requested_r(b), c \in S\} \end{aligned}$$

The functions *mod_add* and *mod_del* will overwrite modified slots by new values in the operational semantics, whereas the function *req_del* simply clears a buffer. As mentioned before, this happens when a request is stated and the buffer waits for its answer. The result of a request is module-dependent and is defined by a buffer-specific function $request_b : 2^{\mathfrak{S}\times\mathfrak{S}} \to \mathfrak{S} \times 2^{\mathfrak{S}\times\mathfrak{S}}$ which receives a finite set of slot-value pairs as input and produces a tuple with a symbol denoting a chunk name and a set of slot-value pairs. Hence, a request is stated by specifying a (partial) chunk derived from the slot-value pairs in the request action of a rule. Its result is again a chunk description (but also containing a name).

We now can define the transition relation $\rightarrowtail$ of our state transition system:

Definition 13 (operational semantics of ACT-R). *For a production rule* $r = (LHS \Rightarrow RHS)$ *the transition* $\rightarrowtail_r$ *is defined as follows.*

rule application: *If there is a fresh variant* $r' := r[\bar{x}/\bar{y}]$ *of rule* r *with variables* $\bar{x}$ *substituted by fresh variables* $\bar{y}$ *and* $\forall(\mathcal{V} \to \exists\bar{y}(LHS \widehat{=} S))$ *then*

$$S := \langle \mathbb{C}; \text{HasSlot}; \text{Holds}; \mathbb{R}\rangle^{\mathcal{V}} \rightarrowtail_r \langle \mathbb{C}; \text{HasSlot}'; \text{Holds}'; \mathbb{R}'\rangle^{\mathcal{V}\cup(LHS(r')\widehat{=}S)}$$

where
- $\text{Holds}' := \text{Holds} - req\_del_S(r')$
- $\text{HasSlot}' := \text{HasSlot} - mod\_del_S(r') \cup mod\_add_S(r')$
- $\mathbb{R}' = \mathbb{R} \cup \{(b, SVP) \mid action(req, b, SVP) \in RHS(r')\}$

We write $S \rightarrowtail_r S'$ *if the rule application transition is used by application of rule* r.

request *If the result of the request* $request_b(SVP^{in}) = (c, SVP)$ *then*
$\langle \mathbb{C}; \text{HasSlot}; \text{Holds}; \mathbb{R} \cup (b, SVP^{in})\rangle^{\mathcal{V}}$
$\rightarrowtail_r \langle \mathbb{C} \cup \{c\}; \text{HasSlot} \cup \bigcup_{(s,v)\in SVP}(c, s, v); \text{Holds} \cup \{(b, c)\}; \mathbb{R}\rangle^{\mathcal{V}}$.
We write $S \rightarrowtail_{request} S'$ *if the request transition is used.*

Informally spoken: If the buffer tests on the l.h.s. match a state, the actions are applied. This means that chunks are modified by replacing parts of the HasSlot relation or chunks are requested by extending $\mathbb{R}$. If a request occurs as action of a rule, the requested buffer is cleared (i.e. the Holds relation is adapted) and a pending request is added to $\mathbb{R}$ memorizing the requested buffer and the arguments of the request in form of slot-value pairs. The variable bindings of the matching are added to the state, i.e. that the fresh variables from the rules are bound to the values from the state. The set of variable binding contains the equality predicates from the matching.

The request transition is possible as soon as a request has been stated, i.e. the last argument of the state is not empty. Then the arguments are passed to the corresponding $request_b$ function and the output chunk is put into the requested buffer.

Note that after a request has been stated the rule application transition might be used since other rules (testing other buffers) might be applicable. This non-deterministic formulation simulates the background execution of requests in the ACT-R reference implementation where a request can take some time until its results are present. During this time, other rules might fire.

3.4 Running Example: Operational Semantics of ACT-R

We exemplify the operational semantics of ACT-R by continuing with our running example – the counting model (section 2.2). The actual instantiation of ACT-R is kept open in the formal semantics (section 3.3): For instance, the semantics talks about an arbitrary set of buffers and corresponding request handling functions. In the following section, we describe the actual instantiation of ACT-R used in our running example. It is the default instantiation of ACT-R models which only use the declarative memory and the goal buffer without interaction with the environment.

ACT-R instantiation For our cognitive model, we need two buffers:

- the *goal buffer* taking track of the current goal and serving as memory for intermediate steps, and
- the *retrieval buffer* giving access to the declarative memory which holds all declarative knowledge of our model, i.e. all known numbers and number sequences.

This means, that the set of buffers is defined as follows: $\mathbb{B} = \{goal, retrieval\}$.

In ACT-R, declarative memory can be seen as an (independent) chunk store $DM = (\mathbb{C}_{DM}, \text{HasSlot}_{DM})$. In the following example, we show the initial content of the declarative memory for our counting model:

Example 2 (ontent of declarative memory).

$$\begin{aligned} \mathbb{C}_{DM} &= \{a, b, c, d, \dots\} \\ \text{HasSlot}_{DM} &= \{(a, first, 1), (a, second, 2), \\ &\quad (b, first, 2), (b, second, 3), \\ &\quad \dots\} \end{aligned}$$

A request to the retrieval buffer (and hence to the declarative memory) is defined as follows:

$$\begin{aligned} request_{retrieval}(SVP) = (c, SVP^{out}) \text{ if } & c \in \mathbb{C}_{DM} \\ & \forall (s, v) \in SVP : \exists (c, s', v') \in \text{HasSlot}_{DM} \\ & \text{such that } s' = s \text{ and } v' = v. \\ & SVP^{out} := \{(c', s', v') \in \text{HasSlot}_{DM} | c' = c\} \end{aligned}$$

This means that a chunk from declarative memory is returned which has all slots and values (matches all conditions) in SVP.

Example Derivation For our counting model, we use the previously defined ACT-R instantiation with a goal and a retrieval buffer. As described before, requests to the declarative memory return a matching chunk based on the arguments given in the request. The next step to an example derivation of the counting model is to define an initial state.

Example 3 (initial state). The initial state is $S_1 := \langle \mathbb{C}, \text{HasSlot}, \text{Holds}, \emptyset \rangle^{\emptyset}$ with the following values:

$$\begin{aligned} \mathbb{C} &= \{a, goalch\} \\ \text{HasSlot} &= \{(a, \mathit{first}, 1), (a, \mathit{second}, 2), \\ &\quad (goalch, \mathit{count}, 1)\} \\ \text{Holds} &= \{(\mathit{goal}, goalch), (\mathit{retrieval}, a)\} \end{aligned}$$

This state has two chunks in its store: The chunk a which encodes the fact that 2 is successor of 1 and a *goalch* which has one slot *count* which is set to 1. This denotes that the current subgoal is to count from 1 to the next number.

We start the derivation from our initial state S_1. For better readability, we apply variable bindings directly in the state representation:

Example 4 (derivation).

$$\begin{aligned} & \langle \{a, goalch\}, \{(a, \mathit{first}, 1), (a, \mathit{second}, 2), (goalch, \mathit{count}, 1)\}, \\ & \quad \{(\mathit{goal}, goalch), (\mathit{retrieval}, a)\}, \emptyset \rangle \\ \rightarrowtail_{\mathit{count}} & \{a, goalch\}, \{(a, \mathit{first}, 1), (a, \mathit{second}, 2), (goalch, \mathit{count}, 2)\}, \\ & \quad \{(\mathit{goal}, goalch))\}, \{(\mathit{retrieval}, \{(\mathit{first}, 2)\})\} \rangle \\ \rightarrowtail_{\mathit{request}} & \{a, b, goalch\}, \\ & \quad \{(a, \mathit{first}, 1), (a, \mathit{second}, 2), (b, \mathit{first}, 2), (b, \mathit{second}, 3), (goalch, \mathit{count}, 2)\}, \\ & \quad \{(\mathit{goal}, goalch))\}, \emptyset \} \rangle \\ & \ldots \end{aligned}$$

It can be seen that as a first derivation step only the application of rule *count* is possible. After the application, only a request derivation step is possible, since the retrieval buffer is empty and hence the condition of rule *count* does not hold.

4 Translation of ACT-R rules to CHR

In this section, we define a translation function $chr(\cdot)$ which translates ACT-R production rules and states to corresponding CHR rules and states. We show later on that the transition is sound and complete w.r.t. the abstract operational semantics of ACT-R. This enables the use of CHR analysis tools like the confluence test to analyze ACT-R models. The translation procedure is very close to the technical implementation given in [10]. Nevertheless, it is the first formal description of the translation process.

Definition 14 (translation of production rules). *An ACT-R production rule r can be translated to a CHR rule $H_k \backslash H_r \Leftrightarrow G | B$ as follows. The translation is denoted as $chr(r)$.*

We introduce a set Θ which takes track of buffer-chunk mappings. We define H_k, H_r, B and Θ as follows:

- *For each* $test(b, SVP) \in LHS(r)$ *introduce a fresh variable* c *and set* $(b, c) \in \Theta$. *There are two cases:*
  case 1: *If* $requested_r(b)$, *then constraint* $buffer(b, c) \in H_r$.
  case 2: *If* $\neg requested_r(b)$, *then constraint* $buffer(b, c) \in H_k$.
  For each $(s, v) \in SVP$*:*
  case 1: *If* $modified_r(b, s)$, *then constraint* $chunk\_has\_slot(c, s, v) \in H_r$.
  case 2: *If* $\neg modified_r(b, s)$, *then constraint* $chunk\_has\_slot(c, s, v) \in H_k$.
- *For each* $action(a, b, SVP) \in RHS(r)$*:*
  case 1: *If* $a = mod$, *then for each* $(s, v) \in SVP$ *there is a constraint* $chunk\_has\_slot(c, s, v) \in B$ *where* $(b, c) \in \Theta$. *Additionally, if there is no* $test(b, SVP') \in LHS(r)$ *with* $(s, v) \in SVP'$, *then introduce fresh variables* c *and* v' *and set* $chunk\_has\_slot(c, s, v') \in H_r$ *and* $(b, c) \in \Theta$.
  case 2: *If* $a = req$, *then constraint* $request(b, SVP) \in B$

We assume a generic rule $request(b, SVP) \Leftrightarrow \ldots$ *in the program which implements the request handling function* $request_b$ *for every buffer* b. *The generation of such rules is given in definition 16.*

Note that the removed heads H_r are constructed by regarding the actions of the rule. If slots are modified that are not tested on the left hand side as mentioned in definition 1, constraints with fresh, singleton variables as values are introduced. Those and are not involved in the matching process of ACT-R rules (see definition 13). Nevertheless, the corresponding constraints must be removed from the store whis is why they appear in H_r. When writing an ACT-R rule it must be ensured that only slots are modified which are part of the modified chunk as required by definition 1. In the CHR translation, such rules would never be able to fire, since the respective constraint appearing in H_k can never be in the store.

Informally, H_k contains all *buffer* and *chunk* constraints as well as all *chunk_has_slot* constraints of the slots which are not modified on the r.h.s. In contrast, H_r contains all *chunk_has_slot* constraints of the slots which appear on the r.h.s., i.e. which are modified.

We now have defined how our subset of ACT-R production rules can be translated to CHR. In the following definition, we present the translation of ACT-R states to CHR states.

Definition 15 (translation of states). *An ACT-R state*

$$S := \langle \mathbb{C}; \text{HasSlot}; \text{Holds} \rangle^{\mathcal{V}}$$

can be translated to the corresponding CHR state (denoted by $chr(S)$*):*

$$\bigwedge_{(b,c)\in \text{Holds}} buffer(b, c) \ \wedge$$
$$\bigwedge_{(c,s,v)\in \text{HasSlot}} chunk\_has\_slot(c, s, v) \ \wedge$$
$$\bigwedge_{(b,SVP)\in\mathbb{R}} request(b, SVP) \wedge \mathcal{V}$$

The Holds and the HasSlot relations are translated to *buffer* and *chunk_has_slot* constraints respectively. Pending requests appear as *request* constraints in the CHR state. The variable bindings $\mathcal{V}$ are represented by built-in equality constraints. The next definition shows how request functions are represented in the CHR program.

Definition 16 (request functions). *A request function $request_b$ can be translated to a CHR rule as follows:*

$$\begin{aligned} request(b, SVP^{in}) \Leftrightarrow \\ & (c, SVP^{out}) = request_b(SVP^{in}) \ \wedge \\ & buffer(b, c) \ \wedge \\ & \forall (s, v) \in SVP^{out} : chunk\_has\_slot(c, s, v) \end{aligned}$$

To continue our running example of the counting model, we show the translation of the production rule in example 1 to CHR:

Example 5 (translation of rules). The rule *count* can be translated to the following CHR rule:

$$\begin{aligned} & buffer(goal, C_1) \ \wedge \\ & chunk\_has\_slot(C_2, first, Num_1) \ \wedge \\ & chunk\_has\_slot(C_2, second, Num_2) \ \backslash \\ & chunk\_has\_slot(C_1, count, Num_1) \ \wedge \\ & buffer(retrieval, C_2) \\ \Rightarrow \\ & chunk\_has\_slot(C_1, count, Num_2) \ \wedge \\ & request(retrieval, \{(first, Num_2)\}) \end{aligned}$$

It can be seen that two new variables are introduced: C_1 which represents the chunk in the goal buffer and C_2 which represents the chunk in the retrieval buffer. The derivation of the program is equivalent to the ACT-R derivation in section 3.4.

To analyze the program for confluence, the notion of observable confluence [7] is needed, since the definition of confluence is too strict: Intuitively, the program is (observably) confluent since there are no overlaps between the rule and the implicit *request* rule. However, there seems to be an overlap of the rule with itself. This overlap does not play a role, since both *buffer* and *chunk_has_slot* represent relations with functional dependency. Hence there is only one possibility to assign values to the variables and finding matching constraints if we only consider CHR representations of valid ACT-R states. However, the confluence test detects those states as non-joinable critical pairs, although they represent states that are not allowed in ACT-R. Hence, those states should not be considered in the confluence analysis, since they can never appear in a valid derivation. To formalize this intuitive observation, the invariants of the ACT-R formalization (like functional

dependency of some of the relations) have to be formulated mathematically to allow for *observable confluence* analysis.

It can be seen that requests potentially produce non-determinism, since either another rule might fire or a request could be performed. Usually, in ACT-R programs, the goal buffer keeps track of the current state of the program and encodes if a request should be awaited or if another rule can fire. However, this leads to a more imperative thinking in the conditions of the rules, since the application sequence of rules is defined in advance.

5 Soundness and Completeness

In this section, we prove soundness and completeness of our translation scheme from definition 14 and definition 15. I.e., we show that each transition of an ACT-R model in a certain state is also possible in the corresponding CHR program with the corresponding CHR state leading to the same results and vice versa. This is illustrated in figure 1. At first, we show that applicability is preserved by the translation and then extend this property to the soundness and completeness theorem 1.

$$\begin{array}{ccc} S & \rightarrowtail & S' \\ chr(\cdot)\big\downarrow & & \big\downarrow chr(\cdot) \\ chr(S) & \mapsto & chr(S') \end{array}$$

Fig. 1. The proposition of theorem 1. We show that applicability and actions are preserved by our translation.

Lemma 1 (applicability). *If the production rule r is applicable in ACT-R state S, then the corresponding CHR rule $chr(r)$ is applicable in state $chr(S) $ and vice-versa.*

Proof. "⇒":

Let $S := \langle \mathbb{C}; \text{HasSlot}; \text{Holds}; \mathbb{R} \rangle^{\mathcal{V}}$. Since r is applicable in S, the following holds:

$$\forall \left(V \rightarrow \exists \bar{x} \left(LHS(r) \ \widehat{=} \ s \right) \right)$$

This implies that for every $test(b, SVP) \in LHS(r)$ $\exists b^S \in \mathbb{B}, c^S \in \mathbb{C} : b = b^S$ and $\forall (s, v) \in SVP$ $\exists s^S, v^S : (c^S, s^S, v^S) \in \text{HasSlot}$ with $s^S = s$ and $v^S = v$ according to definitions 9 and 10.

By definition 15, the state $chr(S)$ has the following constraints: For each $(b^S, c^S) \in \text{Holds}$ there is a constraint $buffer(b^S, c^S) \in chr(S)$ and for every $(c^S, s^S, v^S) \in \text{HasSlot}$ there is a constraint $chunk\_has\_slot(c^S, s^S, v^S) \in chr(S)$. Additionally, $\mathcal{V} \in chr(S)$.

This means that the following conditions hold. We refer to them by ($\star$):

$\forall test(b, SVP) \in LHS(r)\ \exists buffer(b^S, c^S) \in chr(S)$ with $b^S = b$ and $c^S = c$
and
$\forall (s, v) \in SVP\ \exists chunk\_has\_slot(c^S, s^S, v^S) \in chr(S)$ with $s^S = s$ and $v^S = v$

Let $chr(r) = H_k \backslash H_r \Leftrightarrow B$ with $H := H_k \cup H_r$ be the translated CHR rule. For every $test(b, SVP)$ there is a constraint $buffer(b, c) \in H$ with a fresh variable c and for every $(s, v) \in SVP$ there is a constraint $chunk\_has\_slot(c, s, v) \in H$. Additionally, there are constraints $chunk\_has\_slot(c, s, v^*) \in H$ with a fresh variable $v*$ for slots which are modified on r.h.s but which do not appear on l.h.s. $chr(r)$ is applicable in $chr(S)$, if $\exists(G \rightarrow \bar{y}(H = H'))$ where H' are constraints in the state. Due to ($\star$), this condition holds if we set $G = \mathcal{V}$ plus the bindings of the fresh v^* variables. Since for every test in the original ACT-R rule there are corresponding constraints in the state $chr(S)$ and in the rule $chr(r)$ the condition holds for all *chunk_has_slot* constraints who have a correspondent test in $LHS(r)$. The other constraints have a matching partner in $chr(S)$ since a well-formed ACT-R rule only modifies slots which exist for the chunk according to definition 1.

"$\Leftarrow$":

$chr(r)$ of form $H_k \backslash H_r \Leftrightarrow B$ with $H := H_k \cup H_r$ is applicable in state $chr(S) = \langle H' \wedge G \wedge R \rangle$. I.e. that $\forall(G \rightarrow (\exists \bar{x}(H = H'))$. Since $chr(r)$ is a translated ACT-R rule, it only consists of *buffer* and *chunk_has_slot* constraints. Since $H = H'$ there are matching constraints $H' \in chr(S)$, i.e. there is a matching M of the constraints in the state with the constraints in the rule. Set $unifier(LHS(r), S) = M$ and it follows that r is applicable in S.

Lemma 2 (request transitions). *For two ACT-R states S and S' and a CHR state S'', the two transitions $S \rightarrowtail_{request} S'$ and $chr(S) \mapsto_{request} S''$.*

Proof. "$\Rightarrow$":

$S \rightarrowtail_{request} S'$, i.e. $\mathbb{R} \neq \emptyset$ and there is some $(b^*, SVP^{in}) \in \mathbb{R}$. This means that in $chr(S)$ there is a constraint $request(b^*, SVP^{in})$ due to definition 15.

There is a rule with head $request(SVP)$ for every function $request_b(b, SVP) = (c, SVP^{out})$ which implements this function (i.e. which adds $chunk\_has\_slot(c, s, v)$ constraints according to $(SVP)^{out}$ and a $buffer(b, c)$ constraint for a new chunk c). The $request(b^*, SVP^{in})$ constraint is removed from the store like (b^*, SVP^{in}) is removed from $\mathbb{R}$ according to definition 13.

Hence, if the request transition is possible in S, the corresponding request rule is possible in $chr(S)$ and the resulting states $chr(S')$ and S'' are equivalent.

"$\Leftarrow$":

The argument is analogous to the other direction.

Lemma 3 (soundness and completeness of rule application). *For an ACT-R production rule r and two ACT-R states S and S' the transitions $S \rightarrowtail_r S'$ and $chr(S) \mapsto_r S''$ correspond to each other, i.e. $chr(S') = S''$.*

Proof. Let $chr(r) = r@H'_k \backslash H'_r \Leftrightarrow G|B$.

"⇒": According to lemma 1, r is applicable in S iff $chr(r)$ is applicable in $chr(S)$. Let $chr(S) \mapsto_r S'' = (H_k \wedge C \wedge H_k = H'_k \wedge H_r = H'_r \wedge B \wedge G)$ (definition 5). It remains to show that the resulting state $S'' = chr(S')$. Let $S = \langle \mathbb{C}; \text{HasSlot}, \text{Holds}, \mathbb{R} \rangle^{\mathcal{V}}$ and $S' = \langle \mathbb{C}; \text{HasSlot}', \text{Holds}', \mathbb{R}' \rangle^{\mathcal{V}'}$ be ACT-R states. Then

$$\begin{aligned} \text{Holds}' &= \text{Holds} - \mathit{req\_del}_s(r) \\ \text{HasSlot}' &= \text{HasSlot} - \mathit{mod\_del}_S(r) \cup \mathit{mod\_add}_S(r) \\ \mathbb{R}' &= \mathbb{R} \cup \{(b, SVP) | \mathit{action}(\mathit{req}, b, SVP) \in RHS(r')\} \end{aligned}$$

The corresponding CHR state $chr(S')$ contains the following constraints according to definition 15:

$$\bigwedge_{(b,c)\in \text{Holds}'} \mathit{buffer}(b,c) \wedge \bigwedge_{(c,s,v)\in \text{HasSlot}'} \mathit{chunk\_has\_slot}(c,s,v) \wedge \mathcal{V}'$$

Since Holds′, HasSlot′ and $\mathbb{R}'$ are derived from Holds, HasSlot and $\mathbb{R}$, we have to check whether the corresponding *buffer* and *chunk_has_slot* constraints are removed and added to $chr(S)$ by $chr(r)$. For the CHR rule, the body B contains for every $\mathit{action}(\mathit{mod}, b, SVP) \in RHS(r)$, there is a constraint $\mathit{chunk\_has\_slot}(c, s, v) \in \mathbb{B}$ according to definition 14 which is therefore also added to s'' according to definition 5. This corresponds to $\mathit{mod\_add}_s(r)$. According to definition 14, a constraint $\mathit{chunk\_has\_slot}(c, s, v)$ appears in H_r if it is modified on $RHS(r)$ (independent of appearing in a test or not, see case 1.a). This corresponds to $\mathit{mod\_del}_S(r)$. A constraint $\mathit{buffer}(b, c)$ is in H_r, if $\mathit{requested}_r(b)$ is true. This corresponds to $\mathit{req\_del}_s(r)$. For each $\mathit{action}(\mathit{req}, b, SVP) \in RHS(r)$ there appears a constraint $\mathit{request}(b, SVP) \in B$ of the rule. This corresponds to the adaptation of $\mathbb{R}$ in S.

Hence, the state S'' is equivalent to $chr(S')$.

"⇐": Let $chr(S) \mapsto_r chr(S')$ and $S \rightarrowtail_r S''$. According to lemma 1, $chr(r)$ is applicable in $chr(S)$ iff r is applicable in S. It remains to show that the resulting state $S'' = chr(S')$.

The removed constraints H_r in the CHR rule $chr(r)$ are either

(a) *chunk_has_slot* or
(b) *buffer* constraints.

In case (a) the constraints correspond to a modification action in $RHS(r)$. I.e., $\mathit{modified}_r(b, s)$ is true for a constraint $\mathit{chunk\_has\_slot}(c, s, v) \in chr(S)$ with $\mathit{buffer}(b, c) \in chr(S)$ iff it appears in H_r. This corresponds to $\mathit{mod\_del}_s(r)$. In case (b), $\mathit{requested}(b)$ is true for a constraint $\mathit{buffer}(b, c)$ if it appears in H_r according to definition 14. This corresponds to $\mathit{req\_del}_s(r)$.

The added *chunk_has_slot* constraints of B in the CHR rule correspond directly to $\mathit{mod\_add}_S(r)$ by definitions 14 and 12. The *request* constraints in B correspond directly to the adaptation in $\mathbb{R}'$.

Hence, $S'' = chr(S')$.

Theorem 1 (soundness and completeness). *Every ACT-R transition* $s \rightarrowtail s'$ *corresponds to a CHR transition* $chr(S) \mapsto_r chr(S')$ *and vice versa. I.e., every transition (not only rule applications) possible in* S *is also possible in* $chr(S)$ *and leads to equivalent states.*

Proof. By lemmas 3 and 2 the theorem follows directly.

6 Conclusion

In this paper, we have presented a formalization of the core of the production rule system ACT-R including an abstract operational semantics. Furthermore, we have shown a formal translation of ACT-R production rules to CHR. The translation is sound and complete.

The formalization of ACT-R is based on prior work. In [10] we have presented an informal description of the translation of ACT-R production rules to CHR rules. This informal translation has been implemented in a compiler transforming ACT-R models to CHR programs. Our implementation is modular and exchangeable in its core features as we have shown in [11] by exchanging the central part of the conflict resolution with four different methods. Although the implementation is very practical and covers a lot of practical details of the ACT-R implementations, it is not directly usable for analysis.

Our formalization of the translation process in this paper is very near to the practical implementation as it uses the same translation schemes for chunk stores, buffer systems and consequently states. Even the rules are a simplified version of our practical translation from [11]. However, it abstracts from practical aspects like time or conflict resolution. This is justifiable, since for confluence analysis, this kind of non-determinism in the operational semantics is useful. Additionally, as shown in our running example, the general computation process is reproduced closely by our semantics. Furthermore, due to the soundness and completeness of our translation, confluence analysis tools from CHR can be used on our models.

Hence, the contributions of this paper are

- an abstract operational semantics of ACT-R which is – to the best of our knowledge – the first formal representation of ACT-R's behavior,
- a formal description of our translation process (since in [10] a more technical description has been chosen),
- a soundness and completeness result of the abstract translation.

For the future, we want to extend our semantics such that it covers the more technical aspects of the ACT-R production rule system like time and conflict resolution. We then want to investigate how this refined semantics is related to our abstract operational semantics from this paper.

To overcome non-determinism, ACT-R uses a conflict resolution strategy. In [11] we have analyzed several conflict resolution strategies. A confluence test might be useful to reveal rules where the use of conflict resolution is undesired. For the future, we want to investigate how the CHR analysis tools perform for our

ACT-R semantics and how they might support modelers in testing their models for undesired behavior, since the informal application of the confluence test on our example is promising. We plan to lift the results for observable confluence of CHR to ACT-R models. Additionally, it could be interesting to use the CHR completion algorithm [2] to repair ACT-R models that are not confluent. We also want to investigate if the activation levels of ACT-R fit the soft constraints framework [5].

References

1. The ACT-R 6.0 tutorial. http://act-r.psy.cmu.edu/actr6/units.zip (2012), `http://act-r.psy.cmu.edu/actr6/units.zip`
2. Abdennadher, S., Frühwirth, T.: On completion of constraint handling rules. In: Principles and Practice of Constraint Programming, pp. 25–39. Springer Berlin Heidelberg (1998)
3. Anderson, J.R.: How can the human mind occur in the physical universe? Oxford University Press (2007)
4. Anderson, J.R., Bothell, D., Byrne, M.D., Douglass, S., Lebiere, C., Qin, Y.: An integrated theory of the mind. Psychological Review 111(4), 1036–1060 (2004)
5. Bistarelli, S., Frühwirth, T., Marte, M.: Soft constraint propagation and solving in chrs. In: Proceedings of the 2002 ACM symposium on Applied computing. pp. 1–5. ACM (2002)
6. Bothell, D.: ACT-R 6.0 Reference Manual – Working Draft. Department of Psychology, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213
7. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable Confluence for Constraint Handling Rules. In: Dahl, V., Niemelä, I. (eds.) Logic Programming, Lecture Notes in Computer Science, vol. 4670, pp. 224–239. Springer Berlin Heidelberg (2007)
8. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
9. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming (2003)
10. Gall, D.: A rule-based implementation of ACT-R using Constraint Handling Rules. Master Thesis, Ulm University (2013)
11. Gall, D., Frühwirth, T.: Exchanging conflict resolution in an adaptable implementation of ACT-R. Theory and Practice of Logic Programming (to appear) (2014)
12. Taatgen, N.A., Lebiere, C., Anderson, J.: Modeling paradigms in ACT-R. In: Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation., pp. 29–52. Cambridge University Press (2006)