Translation of Cognitive Models from ACT-R to Constraint Handling Rules

Daniel Gall and Thom Frühwirth

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

Abstract. Cognitive architectures are used to abstract and simplify the process of computational cognitive modeling. The popular cognitive architecture ACT-R has a well-defined psychological theory, but lacks a formalization of its computational system. This inhibits computational analysis of cognitive models, e.g. confluence or complexity analysis.
In this paper we present a source to source transformation of ACT-R models to Constraint Handling Rules (CHR) programs enabling the use of analysis tools for CHR to analyze computational cognitive models. This translation is the first that matches the current abstract operational semantics of ACT-R.

Keywords: computational cognitive modeling, ACT-R, operational semantics, source to source transformation, Constraint Handling Rules

1 Introduction

Computational cognitive modeling is a research field at the interface of computer science and psychology. It tries to explore human cognition by building detailed computational models of cognitive processes [20]. Cognitive architectures support the modeling process by offering a formal, well-investigated base that unifies various psychological theories to an abstract theory of cognition. Based upon the architecture, domain specific models are built. In the best case, cognitive architectures constrain the model space to models that are plausible, i.e. a cognitive architecture should only allow models that correspond to human behavior [21].

Adaptive Control of Thought – Rational (ACT-R) [5] is a popular cognitive architecture. It is a modular production rule system with a special architecture of the working memory that operates on data stored as so-called *chunks*, i.e. the unit of knowledge in the human brain. Although it has a well-defined psychological theory, its computational system is not described formally leading to implementations that are full of technical artifacts [14, 4, 19]. This inhibits analysis of cognitive models for features like confluence, termination and computational complexity. Thus, to the best of our knowledge, there are no theoretical results on (semi-)automatic methods deciding one of those computational properties for ACT-R. Nevertheless, since computational models are computer programs, those properties are important since they reveal a lot of information about the semantics of the program or model. For instance, cognitive models with exponential complexity are often implausible, when humans usually find approximate solutions with non-exponential time complexity [17].

Constraint Handling Rules (CHR)[1] [10] is a rule-based language with a strong foundation in logic. In contrast to ACT-R, it has a well-defined operational semantics [6] and even a declarative semantics – the logical reading of a program. There are many theoretical results and practical applications [11] like an automatic confluence test [2, 10], an algorithm to decide operational equivalence [1, 10] and semi-automatic methods for complexity analysis [9].

Due to the strong relation of logic to human deduction and the analysis features of CHR, *we want to use CHR for analysis of cognitive models.* This approach already has been used successfully for analysis of graph transformation systems [16]. In this paper we therefore build on our work in [14], where we have defined the abstract operational semantics of ACT-R. We use the abstract semantics and not an implementation semantics because it is most suitable for analysis of the aforementioned computational properties as it abstracts from details like timings or conflict resolution. Thus it captures the essence of the core transition system of ACT-R as we have shown by a soundness result between the abstract and the implementation semantics in [14]. This makes analysis of the abstract semantics meaningful for implementations.

The *main contribution of this paper* is the translation scheme from ACT-R models to CHR programs to make CHR analysis tools accessible for cognitive models. The translation is constructed such that every computation in the original ACT-R model is also possible in the translated CHR program and, vice versa, only the computations that are possible in ACT-R are possible in the CHR translation. This is important to ensure that the analytical tools of CHR can be used for cognitive models.

The work in this paper extends our prior work from [14] where we have defined the abstract semantics of ACT-R that is suitable for analysis of cognitive models due to its abstraction level. In [13] we have given a first, rough definition of the abstract semantics of ACT-R and a corresponding simple translation to CHR. However, due to differences between the semantics and errors in the previous formulations, the translation from prior work cannot be used for the current, improved semantics of ACT-R in [14]. We want to close this gap in this paper by a formally defined translation of cognitive models to CHR suiting the current operational semantics of ACT-R. This enables sound, elegant analysis of cognitive models through CHR.

2 Preliminaries

In this section we give a short description of Constraint Handling Rules and the cognitive architecture ACT-R. Therefore, we first describe ACT-R very

[1] http://www.constraint-handling-rules.org

briefly and then summarize our results on syntax and semantics from [14]. We concentrate on our so-called *abstract semantics* that we have first described in [13] and improved under the consideration of recent work [4] in [14].

2.1 Constraint Handling Rules

We recapitulate syntax and semantics of CHR briefly. For an extensive introduction to CHR, its semantics, analysis and applications, we refer to [10]. The syntax of CHR is defined over *constraints*, i.e. (first-order) logical predicates. There are two disjoint sets of constraints: user-defined (CHR) constraints and built-in constraints (that come from the host-language CHR is embedded in). A CHR program consists of rules of the form $H_k \setminus H_r \Leftrightarrow G \mid B$ where the heads H_k and H_r are conjunctions of user-defined constraints, the guard G is a conjunction of built-in constraints and the body B is a conjunction of both types of constraints. Note that at most one of H_k and H_r can be empty. If G is empty, it is interpreted as the built-in constraint *true*.

The operational semantics is defined by the following transition scheme over CHR states that are defined as conjunctions of constraints:

$$(H_k \wedge H_r \wedge C) \mapsto (H_k \wedge G \wedge B \wedge C)$$

if there is an instance with new local variables $\bar{x}$ of above rule in head normal form, i.e. all constants in the head of the rule are replaced by variables and respective bindings in the guard, and $CT \models \forall(C \rightarrow \exists \bar{x} G)$ for a constraint theory CT [10].

Informally, a CHR program is run on a constraint store, that is a conjunction of constraints. A rule is applicable, if the head matches constraints from the store and the guard holds. In that case, the matching constraints from H_k are kept in the store, the constraints matching H_r are removed and the constraints from B and G are added.

Throughout this paper, we use multi-set notation to describe logical conjunctions, e.g. to describe CHR states. Thereby, $\uplus$ denotes multi-set union. We also implicitly convert (multi-)sets to corresponding lists (denoted by square brackets) when using them within a constraint.

2.2 Informal Description of ACT-R

ACT-R is a modular production rule system. Its data elements are so-called *chunks*. A chunk has a *type* and a set of *slots* (determined by the type) that are connected to other chunks. Hence, human declarative knowledge is represented in ACT-R as a network of chunks. Figure 1 shows an example chunk network that models the family relations between some persons.

In figure 2, there is an overview of ACT-R's architecture. The modules are responsible for different cognitive features. For instance, the declarative knowledge (represented as a chunk network) can be found in the *declarative module*. The heart of ACT-R is the *procedural system* that consists of a set of *production rules*.

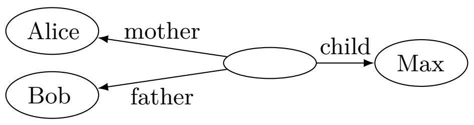

Fig. 1. A chunk network that stores some family relations. Thereby, the chunks named *Alice*, *Bob* and *Max* are of a chunk type that does not have further slots. The central chunk (not named in the figure) is of type *parent* with three slots: *mother*, *father* and *child*. If Alice and Bob had more children, there would be more such chunks connecting them to the chunks representing their other children.

Those rules do not have access to all information from other modules, but only to parts of it that are stored in *buffers*. A buffer is connected to a module and can hold at most one chunk at a time. Rules match the contents of the buffer, i.e. they check if the chunks of particular buffers have certain values. If a rule is applicable, it can *modify* particular slots of the chunk in the buffer, *request* the module to put a whole new chunk in its buffer or *clear* a buffer. Modifications and clearings are available for the production rule system, whereas requests can take some time while the procedural system is continuing work in parallel.

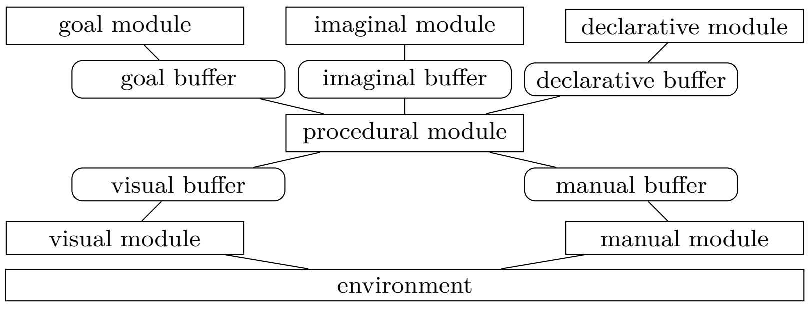

Fig. 2. Modular architecture of ACT-R. This illustration is inspired by [21] and [5].

2.3 Syntax

We use the term representation of the ACT-R syntax that we have introduced in [14]. This syntax can be obtained directly from the original syntax of ACT-R. However, it simplifies its handling using logical or set operators.

All terms in ACT-R are defined over two disjoint, possibly infinite sets of constant symbols $\mathcal{C}$ and variable symbols $\mathcal{V}$. An ACT-R architecture defines the set of buffers $\mathbb{B} \subseteq \mathcal{C}$ and the set of actions A. In this paper, we restrict the set of actions to $\{\texttt{=}, \texttt{+}, \texttt{-}\}$ for modifications, requests and clearings respectively.

An ACT-R model consists of a set of *rules* Σ, a set of type names $\mathbb{T} \subseteq \mathcal{C}$ and a (total) *typing function* $\tau : \mathbb{T} \to 2^{\mathcal{C}}$ that defines the slots for each type. A *production rule* in Σ is defined as $L \Rightarrow R$, where L is a set of buffer tests of the form $\texttt{=}(b, t, P)$ and R is a set of actions of the form $a(b, t, P)$ where $a \in A$ is an action symbol, $b \in \mathbb{B}$ is a buffer, $t \in \mathbb{T}$ is a type and $P \subseteq \mathcal{C} \times (\mathcal{C} \cup \mathcal{V})$ is a set of slot-value pairs.

The function *vars* maps an arbitrary set of terms to its set of variables in $\mathcal{V}$. We require $\mathit{vars}(R) \subseteq \mathit{vars}(L)$ for a production rule $L \Rightarrow R$, i.e. no new variables must be introduced on the right-hand side.

There are some further syntactic restrictions: Actions $a(b, t, P) \in R$ are only allowed for tested buffers, i.e. if $\texttt{=}(b, t', P') \in L$. A modification action may not change the type of the chunk, i.e. if $\texttt{=}(b, t, P) \in R$ then $\texttt{=}(b, t, P') \in L$. We also require that each test refers to another buffer. Additionally, the actions are only allowed to specify each slot at most once in their set of slot-value pairs.

In the following example, we show the syntax of an ACT-R production rule and its informal semantics.

Example 1 (production rules). This example builds on chunks of type *parent* as in figure 1. By the following rule we want to determine the parents of a given person. Therefore, we have special goal chunks of type g that represent our query. They have the slots *query*, *mother*, *father* and *state*. This means that they hold the person whose parents are of interest (*query*), the current state of the derivation (*state*) and the result (*mother* and *father*). In the beginning, a goal chunk is only connected to a person chunk by the *query* slot and has the value *start* in its state slot. The model will connect the other slots with corresponding chunks.

Our example rule starts the retrieval of the queried chunk:

$$\begin{aligned} &\{\texttt{=}(\mathit{goal}, g, \{(\mathit{state}, \mathit{start}), (\mathit{query}, X)\})\} \\ \Rightarrow\ &\{\texttt{+}(\mathit{retrieval}, \mathit{parent}, \{(\mathit{child}, X)\}), \texttt{=}(\mathit{goal}, g, \{(\mathit{state}, \mathit{retrieval})\})\} \end{aligned}$$

The variable X denotes the name of the child. In the actions, we state a request to the *retrieval* buffer to look for a chunk of type *parent* that has X in its *child* slot. The state is modified from *start* to *retrieval*.

To complete the computation, we would need a second rule that takes the result in the retrieval buffer modifies the mother and father slots of the goal chunk accordingly.

2.4 Operational Semantics

In this section we describe the semantics of an ACT-R model as a state transition system based on our prior work [14]. Therefore, we first introduce the notion of an ACT-R state and then give the transition relation $\rightarrowtail$.

States ACT-R operates on a network of typed chunks that we call a *chunk store*. It is defined over a set of types $\mathbb{T}$ and a typing function τ. Chunks are defined as unique, immutable entities with a type and connections to other chunks:

Definition 1 (chunk store). *A* chunk store Δ *is a multi-set of tuples* (t, val) *where* $t \in \mathbb{T}$ *is a chunk type and* $val : \tau(t) \to \Delta$ *is a function that maps each slot of the chunk (determined by the type t) to another chunk. To identify an individual chunk, we define the total function* $id : \Delta \to \mathcal{C}$ *that maps each chunk to a unique identifier from the set of constants that is determined by its type and slot-value pairs.*

The typing function τ maps a type t from the set of type names $\mathbb{T}$ to a set of allowed slots, hence the function *val* of chunk c has the slots of c as domain. Note that a chunk store can contain multiple elements with the same values that still are unique entities representing different concepts.

We assume that there is a special type $\texttt{chunk} \in \mathbb{T}$ with $\tau(\texttt{chunk}) = \emptyset$. Additionally, there is a chunk $\texttt{nil} \in \Delta$ that is defined as $\texttt{nil} := (\texttt{chunk}, \emptyset)$.

We now define the notion of a *cognitive state* as the content of the buffers:

Definition 2 (cognitive state). *A* cognitive state γ *is a function* $\mathbb{B} \to \Delta \times \mathbb{R}_0^+$ *that maps each buffer to a chunk and a delay. The set of cognitive states is denoted as* Γ*, whereas* Γ_{part} *denotes the set of* partial cognitive states*, i.e. cognitive states that are partial functions and do not necessarily map each buffer to a chunk. A buffer b is called* empty*, if* $\gamma(b) = \texttt{nil}$.

The delay decides at which point in time the chunk in the buffer is available to the production system. A delay $d > 0$ indicates that the chunk is not yet available to the production system. This implements delays of the processing of requests.

Definition 3 (ACT-R states). *An* abstract ACT-R state *is a tuple* $\langle \Delta; \gamma; \upsilon \rangle^{\mathbb{V}}$ *where* Δ *is a chunk store,* γ *is a cognitive state using* Δ*,* υ *is a multi-set of first-order predicates (called* additional information*) and* $\mathbb{V}$ *is a set of variable bindings. The set of allowed parameter valuations* Υ *is defined by the concrete architecture.*

The additional information is used to model the modularity of ACT-R where the procedural system does not have direct access to all information in the individual modules. For instance, it contains so-called sub-symbolic information that is used to define activation levels of chunks, e.g., to model learning and forgetting.

State Transitions First of all, we define the notion of matchings:

Definition 4 (matching). *A buffer test* $\theta := \texttt{=}(b, t, P)$ *for a buffer* $b \in \mathbb{B}$ *testing for a type t and slot-value pairs* $P \subseteq \mathcal{C} \times (\mathcal{C} \cup \mathcal{V})$ matches *a state* $\sigma := \langle \Delta; \gamma; \upsilon \rangle^{\mathbb{V}}$*, written* $\theta \sqsubseteq \sigma$*, if and only if* $\gamma(b) = ((t, val), 0)$ *and for all* $(s, v) \in P : \forall (\mathbb{V} \to \exists v' \in \mathcal{V} : id(val(s)) = v' \land v = v')$ *for a fresh variable* v'.

A rule $r := L \Rightarrow R$ *matches a state* σ*, written as* $r \sqsubseteq \sigma$*, if and only if all buffer tests* $t \in L$ *match* σ*. We define the set* $Bindings(r, \sigma)$ *as the bindings of the variables that follow from the matching* $r \sqsubseteq \sigma$*.*

A buffer test matches a state, if and only if all its slot tests hold in the state, i.e. the variable bindings imply that the values in the rule are the same as in the state (for the tested buffer). Note that a test can only match chunks in the cognitive state that are visible to the system, i.e. whose delay is zero. A test cannot match chunks with a delay greater than zero.

The modification of a state by a transition is defined by interpretation functions $I : \mathcal{A} \times \mathcal{S}_{va} \to 2^{\Gamma_{\text{part}} \times \Upsilon}$ of actions that determine the possible effects of an action. An interpretation maps each state and action of the form $a(b, t, P)$ – where $a \in A$ is an action symbol, $b \in \mathcal{C}$ a constant denoting a buffer, $t \in \mathcal{C}$ a type, and $P \subseteq \mathcal{C} \times (\mathcal{C} \cup \mathcal{V})$ is a set of slot-value pairs – to a tuple $(\gamma_{\text{part}}, \upsilon)$. Thereby, γ_{part} is a partial cognitive state, i.e. a partial function that assigns some buffers a chunk. The partial cognitive state γ_{part} will be taken in the operational semantics to overwrite the changed buffer contents, i.e. it contains the new contents of the changed buffers. Analogously, the additional information υ defines changes of parameter valuations induced by the action.

Note that the interpretation of an action can return more than one possible effect. For example, the declarative module can find more than one chunk matching the retrieval request. In implementations usually one chunk is returned according to certain additional information (called chunk activation which is an elementary concept of ACT-R to model learning). However, in the abstract semantics all matching chunks are regarded to find potential conflicts in a model.

From the interpretation of one action, the interpretation of a rule can be derived by combining the individual actions. Since the actions refer to different buffers, the changes of the partial cognitive state are disjoint and can be combined to a larger partial cognitive states. Additional parameters can simply be merged by multi-set union. Due to space reasons, we refer to [14] for a formal description.

We now define the transition relation $\rightarrowtail$ of ACT-R. The first class of transitions (*rule transitions*) is defined for a fresh variant r' of a rule $r \in \Sigma$ with $\mathit{vars}(r') = \bar{y}$:

$$\frac{r' \sqsubseteq \sigma \wedge \mathbb{V}^* = \mathit{Bindings}(r', \sigma) \wedge (\gamma_{\text{part}}, \upsilon^*) \in I(r')}{\sigma := \langle \Delta; \gamma; \upsilon \rangle^{\mathbb{V}} \rightarrowtail \langle \Delta \uplus \Delta'; \gamma'; \upsilon' \rangle^{\mathbb{V} \cup \mathbb{V}^*}}$$

where Δ' are the chunks added in the interpretation function. The *id* function is extended for the chunks in Δ' by fresh names from $\mathcal{C}$. γ' has the values of γ_{part} where defined or the values of γ otherwise, and $\upsilon' := \upsilon \uplus \upsilon^*$. The interpretation function I is defined as follows:

- $I(=(b, t, P), \sigma) = \{(\gamma_p, \emptyset)\}$ for modifications where $\gamma_p(b) := ((t, \mathit{val}'_b), 0)$. For $\gamma(b) = ((t, \mathit{val}_b), d)$ (from the state σ), the new values are defined as $\mathit{val}'_b(s) := v$ if $(s, v) \in P$ and $\mathit{val}'_b(s) := \mathit{val}_b(s)$ otherwise.
  Thus, a modification creates a copy of the chunk in the buffer with modified values as specified in P. Modifications are deterministic, i.e. that there is only one possible effect.
- $(\gamma_p, \upsilon_b) \in I(+(b, t, P), \sigma)$ for requests if $(c_b, \upsilon_b) \in \mathit{request}_b(t, P, \upsilon)$ and $\gamma_p(b) := (c_b, 1)$. Thereby, the function $\mathit{request}_b : \mathbb{T} \times 2^{\mathcal{C} \times (\mathcal{C} \cup \mathcal{V})} \times \Upsilon \to 2^{\Delta \times \Upsilon}$ is a function defined by the architecture for each buffer. It calculates the set of possible answers for a request that is specified by a type and a set of slot value pairs. Possible answers are tuples of a chunk and additional information.

– $(\gamma_p, \upsilon_p) \in I(\texttt{-}(b, \texttt{chunk}, \texttt{nil}), \sigma)$ for clearings where $\gamma_p(b) = (\texttt{nil}, 0)$ and

$$\begin{aligned}\upsilon_p := \{&\mathit{dmchunk}(\mathit{id}(c), t) \mid \gamma(b) = (c, d) \wedge c = (t, \mathit{val})\} \uplus \\ \{&\mathit{dmchs}(\mathit{id}(c), s, v) \mid \gamma(b) = (c, d) \wedge c = (t, \mathit{val}) \wedge s \in \tau(t) \wedge \mathit{val}(s) = v\}.\end{aligned}$$

The buffer is emptied and its chunk is added to declarative memory represented as additional information.

There are also transitions without a rule (*no rule transitions*):

$$\frac{b^* \in \mathbb{B} \wedge \gamma(b^*) = (c^*, d^*) \wedge d^* > 0}{\sigma := \langle \Delta; \gamma; \upsilon \rangle^{\mathbb{V}} \rightarrowtail \langle \Delta; \gamma'; \upsilon \rangle^{\mathbb{V}}}$$

where $\gamma'(b^*) := (c^*, 0)$. Thus, one pending request is chosen non-deterministically to be applied for one buffer b^*.

3 Translation

In this section we show how to translate an ACT-R model to a CHR program. This is the main contribution of this paper. The translation is the first that matches the current operational semantics of ACT-R.

3.1 Set Normal Form

To simplify the translation scheme, we assume the ACT-R production rules to be in *set normal form*, i.e. that each buffer test only contains each slot at most once. Every production rule can be transformed to a production rule preserving operational semantics: If a rule has (s, v) and (s, v') in one buffer test, then one of the two must be a variable or $v = v'$, otherwise the rule can never fire since one slot cannot have two different values. Let v be a variable and v' a variable or constant. Then the operational semantics will add the following bindings to the state: $v = v' = v^*$ for some constant v^* (that is the identifier of a chunk) from the state. We can now simply replace each occurrence of v by v' in the rule directly and have the same semantics.

3.2 Translation of States

An ACT-R state $\sigma := \langle \Delta; \gamma; \upsilon \rangle^{\mathbb{V}}$ can be translated to the following CHR state:

$$\begin{aligned}&\biguplus_{b \in \mathbb{B}} \{\mathit{buffer}(b, \mathit{id}(c), t, d) \mid \gamma(b) = (c, d) \wedge c = (t, \mathit{val}_c)\} \\ \uplus \ &\biguplus_{b \in \mathbb{B}} \{\mathit{chs}(\mathit{id}(c), s, \mathit{id}(v)) \mid (c, d) \in \Delta \wedge c = (t, \mathit{val}_c) \wedge \mathit{val}_c(s) = v\} \\ \uplus \ &\upsilon \uplus \mathbb{V} \uplus \{\mathit{fire}\}.\end{aligned}$$

Hence, for every buffer a *buffer* constraint with the chunk id, the type and the delay is constructed. For every slot-value pair in the valuation function of a chunk, a corresponding *chs* constraint is added to the store that keeps track of the connections of a chunk. Note that by this definition chunks that appear in more than one buffer are copied in the CHR state and have the same identifier. This is made explicit in the definition by the multi-set union over all buffers.

Additional information and variable bindings are translated to corresponding built-in constraints. As we will see in the following sections, the *fire* constraint is needed to enable translated ACT-R rules to fire. This is necessary, since our translation needs additional transitions for one original rule application. Those additional transitions must not be interfered by other rule applications.

3.3 Translation of Rules

In our translation scheme, ACT-R rules are translated to corresponding CHR rules. However, as we will see, there are some additional rules needed to achieve the same behavior in both languages.

First of all, to manage relations between newly introduced variables, we define some auxiliary functions: Both the *chunk variable function* $cvar : \mathbb{B} \rightarrow \mathcal{V}_1$ and the *modified chunk function* $mvar : \mathbb{B} \rightarrow \mathcal{V}_2$ are defined as $b \mapsto C_b$ and return a fresh, unique variable C_b for each buffer b. The codomains $\mathcal{V}_1$ and $\mathcal{V}_2$ are disjoint subsets of $\mathcal{V}$. The first function will identify the chunk of a particular buffer in the translation. The second function is needed to introduce copies of chunks in the translation. Finally, the *value function* $chrval : \mathbb{B} \times \mathcal{C} \rightarrow \mathcal{V}, (b, s) \mapsto V_{b,s}$ returns a fresh, unique variable $V_{b,s}$ for each buffer b and slot name s. Those variables are needed to identify the values of slot-value pairs. Note that in the following we implicitly translate ACT-R constants and variables from $\mathcal{C}$ and $\mathcal{V}$ to corresponding CHR variables.

We define the translation of a production rule r of the form $L \Rightarrow R$ to a CHR rule $H_k \setminus H_r \Leftrightarrow G_= \uplus G_+ \mid B \uplus B_= \uplus B_+ \uplus B_-$ in the following sections that describe the translation of the individual parts of the rule.

Tests The tests of the ACT-R production rule roughly correspond to the head of the CHR rule, i.e. the head of the CHR rule mainly depends on the tests in L. If there is an action for a tested buffer, then the buffer constraint is removed, otherwise it is kept. We add *chs* constraints for all slots of every tested chunk to access all values. Hence, the heads of our CHR rule are defined as follows:

$$\begin{aligned}
H_r :=& \{\mathit{fire}\} \uplus \{\mathit{buffer}(b, \mathit{cvar}(b), t, 0) \mid {=}(b, t, P) \in L \wedge a \in A \wedge a(b, t', P') \in R\} \\
H_k :=& \{\mathit{buffer}(b, \mathit{cvar}(b), t, 0) \mid {=}(b, t, P) \in L \wedge a \in A \wedge a(b, t, P') \notin R\} \\
& \uplus \{\mathit{chs}(\mathit{cvar}(b), s, v) \mid {=}(b, t, P) \in L \wedge (s, v) \in P\} \\
& \uplus \{\mathit{chs}(\mathit{cvar}(b), s, \mathit{chrval}(b, s)) \mid {=}(b, t, P) \in L \wedge s \in \tau(t) \wedge s \notin \mathit{slots}(P)\}.
\end{aligned}$$

We require the *fire* constraint and remove it, hence no other translated production rule can fire. As mentioned before, we want to ensure that certain maintenance rules that are described in the following are completed before another rule is fired. Additionally, the rule deletes the connection between the buffer and its chunk, if there is an action for this buffer on the right hand side of the rule. The removed buffer constraints are later on replaced by new constraints that are connected to other chunks. This is because actions modify a copy of the original chunk instead of modifying it in-place.

General Translation of Actions As can be seen in the operational semantics of ACT-R, actions specify a (partial) cognitive state that describes how the contents of the buffers change. All types of actions can be described by this abstraction. We will implement this concept in CHR to handle rule actions. Therefore, we first need to solve the technical problem that we have to know how many actions a rule has to perform to make sure that no other rules interfere with the process of applying the modifications. We keep track of this information in the *actions* constraint that holds the value of pending actions. Note that this is a static information that is known at compile time. Hence, we can add this constraint in the general part of the translated rule: $B := \{\mathit{actions}(|R|)\}$.

The following two general rules are added to the translated program. They are needed to actually perform the actions:

$$\mathit{actions}(N) \wedge \mathit{mod}(C, []) \Leftrightarrow N > 0 \mid \mathit{actions}(N-1).$$
$$\mathit{actions}(N) \setminus \mathit{mod}(C, [(S,V)|P]) \Leftrightarrow N > 0 \mid \mathit{chs}(C,S,V) \wedge \mathit{mod}(C,P).$$

The two rules add *chs* constraints for a chunk C as specified in the list of slot-value pairs until it is empty. If all actions have been performed, i.e. the execution of the rule is finished, we make system able to fire again by the following general rule that is part of the translated program: $\mathit{actions}(0) \Leftrightarrow \mathit{fire}$. In the following we describe how the particular actions are translated to such *mod* constraints.

Modifications Modifications copy the chunk from the buffer and modify particular slots in it. The previous chunks are not removed from the chunk store, but their link to the buffer is removed by removing the *buffer* constraint if a modification of the corresponding buffer is present in the rule. Hence, a modification has to add a new *buffer* constraint that links the buffer to the modified copy of the chunk. Therefore, we call the function $\mathit{completion}_b : \mathbb{T} \times 2^{(\mathcal{C} \cup \mathcal{V})} \to 2^{(\mathcal{C} \cup \mathcal{V})}$ for a buffer $b \in \mathbb{B}$ a *chunk completion function* that is defined as $\mathit{completion}_b(t, P) := \{(s, \mathit{chrval}(b, s)) \mid s \in \tau(t) \wedge s \notin \mathit{slots}(P)\}$. The function gets a type and a set of slot-value pairs and returns the set of slot-value pairs for the slots that do not appear in P, but are part of the type t. The values in the result are variables that are generated by the *chrval* function depending on the buffer b to avoid variable name clashes.

The modification part of the translated rule is then defined as:

$$\begin{aligned} G_{=} &:= \{\mathit{newID}(\mathit{mvar}(b)) \mid {=}(b,t,P) \in R\} \\ B_{=} &:= \{\mathit{buffer}(b, \mathit{mvar}(b), t, 0) \mid {=}(b,t,P) \in R\} \\ &\quad \uplus \{\mathit{mod}(\mathit{mvar}(b), P \cup \mathit{completion}_b(t,P)) \mid {=}(b,t,P) \in R\} \end{aligned}$$

The built-in constraint *newID*/1 binds a new constant name to the variable that we obtain from the *mvar* function in the guard. This name corresponds to the new id in the operational semantics. Then a copy of the old chunk with some modified values is placed into the buffer.

Requests Requests to a module are modeled as built-in constraints. This means that the request function from the operational semantics of ACT-R is modeled by a built-in constraint $\mathit{request}(b, t, c_r, T, V)$ whose answer depends on the built-in store (that corresponds to the additional information υ). Hence, the following parts are added to our translated CHR rule, where for every buffer there are unique, fresh variables T^a_b and V^a_b:

$$\begin{aligned}
G_{+} :=& \{\mathit{request}(b, t, p, T^a_b, V^a_b) \mid \texttt{+}(b, t, p) \in R\} \uplus \{\mathit{newID}(\mathit{mvar}(b))\} \\
B_{+} :=& \{\mathit{buffer}(b, \mathit{mvar}(b), T^a_b, 1) \mid \texttt{+}(b, t, p) \in R\} \\
& \uplus \{\mathit{mod}(\mathit{mvar}(b), V^a_b) \mid \texttt{+}(b, t, c_r) \in R\}
\end{aligned}$$

A request simply takes the answer of the built-in request and puts it into the requested buffer. The *newID* built-in constraint again produces a new name for the chunk.

Clearings In ACT-R, chunks are copied to declarative memory when a buffer is cleared. To do this, we need to know the contents of all the slots that define the chunk. However, in the definition of our translation scheme of rules we do not have access to all constraints defining the chunk that is removed from the buffer. At compilation time it is not possible to know what type of chunk will be in the buffer to be cleared. However, the kind and number of *chs* constraints depends on that type. Thus, we have to delay the application of the clearing and handle it by introducing an extra rule for each type t of the form $H^t_r \setminus H^t_k \Leftrightarrow B^t$ with fresh CHR variables N, B, C, D, and (for all $s \in \tau(t)$) V_s:

$$\begin{aligned}
H^t_r &:= \{\mathit{actions}(N), \mathit{clear}(B), \mathit{buffer}(B, C, t, D)\} \\
H^t_k &:= \{\mathit{chs}(C, s, V_s) \mid s \in \tau(t)\} \\
B^t_c &:= \{\mathit{dmchs}(C, s, V_s) \mid s \in \tau(t)\} \\
&\quad \uplus \{\mathit{dmchunk}(C, t), \mathit{buffer}(B, \texttt{nil}, \texttt{chunk}, 0), \mathit{actions}(N-1)\}
\end{aligned}$$

This rule is only applicable, if a buffer clearing was triggered by the last rule applied (ensured by the *clear*/1 constraint that is introduced by the rule with the clearing action as we will see). Note that due to the removal of the *fire* constraint, only clearing rules can be applied.

Our translation of the ACT-R rule with a buffer clearing has the following body:

$$B_{-} := \{\mathit{clear}(b) \mid \texttt{-}(b, \texttt{chunk}, \texttt{nil}) \in R\}.$$

We now exemplify the translation of rules:

Example 2 (translation of rules). The rule from example 1 can be translated to the following CHR rule:

$$\begin{aligned}
& \mathit{chs}(G, \mathit{state}, \mathit{start}) \wedge \mathit{chs}(G, \mathit{query}, X) \wedge \mathit{chs}(G, \mathit{mother}, M) \wedge \\
& \mathit{chs}(G, \mathit{father}, F) \backslash \mathit{buffer}(\mathit{goal}, g, G, 0) \wedge \mathit{fire} \\
\Leftrightarrow\ & \mathit{newID}(G') \wedge \mathit{newID}(R') \wedge \mathit{request}(\mathit{retrieval}, \mathit{parent}, [(\mathit{child}, X)], T, V, \_) \mid \\
& \mathit{buffer}(\mathit{goal}, g, G', 0) \wedge \mathit{buffer}(\mathit{retrieval}, T, R', 1) \wedge \mathit{mod}(G', V) \wedge \\
& \mathit{mod}(G', [(\mathit{state}, \mathit{retrieval}), (\mathit{query}, X), (\mathit{mother}, M), (\mathit{father}, F)]).
\end{aligned}$$

3.4 No Rule Transition

In addition to transitions by rule applications, ACT-R can also have state transitions without rule applications. This is useful for instance, if no rule is applicable (i.e. computation is stuck in a state) but there are pending requests, then simulation time can be forwarded to the point where the next request is finished and its results are visible to the procedural system. This may trigger new rules and continue the computation.

The *no rule* transition can be modeled in CHR by one individual generic rule:

$$\mathit{fire} \setminus \mathit{buffer}(B, T, C, D) \Leftrightarrow D > 0 \mid \mathit{buffer}(B, T, C, 0)$$

This rule application is only possible when generally a rule could fire (ensured by the *fire* constraint). This transition is possible for all requests that are pending (i.e. that have a delay $D = 1$). Hence, one request is chosen non-deterministically.

4 Discussion

We have constructed our translation such that the translated program behaves equivalently to the original ACT-R model, i.e. every transition that is possible in the ACT-R model is also possible in the translated program leading to equivalent subsequent states (soundness) and vice versa (completeness). However, our translation has the restriction that one ACT-R transition $\sigma \rightarrowtail \sigma'$ can correspond to possibly more than one but finitely many transitions in CHR: $\mathit{chr}(\sigma) \mapsto \ldots \mapsto \mathit{chr}(\sigma')$, so-called *macro-steps.* In the intermediate states no regular transitions are possible. This is ensured by the removal of the *fire* constraint in all of those translated rules. The only rules that are applicable in an intermediate state are the ones that replace *mod* constraints with their corresponding *chs* constraints (or the respective constraints for clearings), i.e. that actually apply the actions to the state described by those constraints. Each action of a rule only introduces n such constraints, where n is the number of actions of the rules. After n steps, all *mod* and *clear* constraints are removed and a new *fire* constraint is introduced leading to a state that is equivalent to σ'.

This state allows the same macro-transitions as the original ACT-R state and describes the equivalent cognitive state and additional information. The latter is argued in the translation of the particular actions. The applicability of translated CHR rules can be seen directly from the definition of matchings (c.f. definition 4) and the applicability condition of CHR in section 2.1. The set normal form of rules and the copying of chunks in the CHR state ensure that there are constraints available in the state to match the translated rule even if two buffers hold the same chunk or two slots of the same chunk are tested twice.

For the completeness, obviously only CHR states that model valid ACT-R states can be considered. In particular it is required that they contain a *fire* constraint, the functional character of a cognitive state is maintained and chunks are described completely according to their type by the respective constraints, for instance. Then it can easily be seen, that both applicability and equivalence of actions are maintained by the translation.

5 Related Work

We first want to relate the progress of this paper with our prior work. In [13] we first have presented an abstract operational semantics and a corresponding translation to CHR together with a soundness and completeness result. However, this semantics has ignored some details that are crucial for ACT-R, like the copying of chunks when they enter a buffer. The old semantics used in-place modification which does not directly correspond to how most implementations work. Furthermore, the formulation of the semantics led to complicated proofs.

In [14] we have improved our semantics and unified it with independent work from [4]. There we concentrated on the semantics that we use for the CHR translation in this paper.

Our approach abstracts from technical details that vary in different ACT-R implementations or depend on parameter settings like timings and conflict resolution. We rather capture all possible transitions non-deterministically. Those non-deterministic transitions are removed by a conflict resolution mechanism in implementations. However, when writing a model, one is often interested in the general sequence of transitions and only later in concrete timings. Our approach gives us the power to reason about the core of the procedural system of ACT-R. For instance, a model that is confluent under our abstract semantics is independent of the order of rules, initial utility values of rules (used for conflict resolution) and timings. This gives a more concise view on the model.

We model implementation details in a refined semantics that is an instance of our abstract semantics [14]. This leads to concise, flexible implementations as shown in [12] by exchanging conflict resolution in our implementation of ACT-R.

There are many implementations of ACT-R in different languages that reach from the Lisp reference implementation [7] to certain Java (e.g. [18]) or even Python implementations [19]. All of those approaches are efforts of getting rid of many technicalities that have been incorporated over time in the reference implementation, but none of them deal with formal analysis of ACT-R.

Closest to our work is F-ACT-R [4, 3], a formal formulation of the ACT-R semantics together with an implementation with the aim of simplifying model analysis. However, there are no confluence or complexity analysis tools, yet.

6 Conclusion and Future Work

In this paper we have presented a translation scheme of ACT-R models to CHR programs. It is the first of its kind that suits the current definition of the abstract operational semantics of ACT-R [14] that introduced significant changes compared to the prior rough definition of the semantics from [13]. The translation does not guarantee that each ACT-R transition corresponds to exactly one CHR transition, but in finitely many steps a valid ACT-R state is reached in CHR representing exactly the ACT-R state from the original transition.

This property enables us to use analysis methods and tools from the CHR world to analyze cognitive models. For example, in CHR confluence is decidable for terminating programs [10] and there is a tool that decides it automatically [15]. Another example are methods for semi-automatic complexity analysis [9, 10] that exist for CHR. Complexity is an important property of cognitive models, since it can decide if a model is plausible or not. For instance, there are cognitive tasks that can be solved by humans in short time for growing problem size (but with errors) where the best cognitive models have exponential complexity [17]. Hence, such models are not plausible, since they seem to pursue the wrong approach.

Although CHR analysis tools can now be used on the translated programs, there are some practical limitations: we want to investigate how our translation can be used for analysis of cognitive models in practice. for instance, that confluence is often too strict. Hence, the confluence criterion of CHR classifies ACT-R models as non-confluent that should be confluent since there are certain invariants on valid ACT-R states that are not considered by the confluence criterion. For example, one invariant is that there can only be one *buffer* constraint for each buffer. Therefore, we want to use *observable confluence* to improve the behavior of the confluence criterion for cognitive models [8] in future work. Additionally, we want to investigate how reasoning on declarative knowledge can be improved by a constraint system using CHR.

References

1. Abdennadher, S., Frühwirth, T.: Operational equivalence of CHR programs and constraints. In: Jaffar, J. (ed.) Principles and Practice of Constraint Programming – CP'99, Lecture Notes in Computer Science, vol. 1713, pp. 43–57. Springer Berlin Heidelberg (1999)
2. Abdennadher, S., Frühwirth, T., Meuss, H.: On confluence of Constraint Handling Rules. In: Freuder, E.C. (ed.) Principles and Practice of Constraint Programming — CP96, Lecture Notes in Computer Science, vol. 1118, pp. 1–15. Springer Berlin Heidelberg (1996)
3. Albrecht, R., Gießwein, M., Westphal, B.: Towards formally founded ACT-R simulation and analysis. In: Proceedings of the 12th Biannual conference of the German cognitive science society (Gesellschaft für Kognitionswissenschaft). Cognitive Processing, vol. 15 (Suppl. 1), pp. 27–28. Springer (2014)
4. Albrecht, R., Westphal, B.: F-ACT-R: defining the ACT-R architectural space. In: Proceedings of the 12th Biannual conference of the German cognitive science society

(Gesellschaft für Kognitionswissenschaft). Cognitive Processing, vol. 15 (Suppl. 1), pp. 79–81. Springer (2014)

5. 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)
6. Betz, H., Raiser, F., Frühwirth, T.: A complete and terminating execution model for Constraint Handling Rules. Theory and Practice of Logic Programming 10, 597–610 (7 2010)
7. Bothell, D.: ACT-R 6.0 Reference Manual – Working Draft. Department of Psychology, Carnegie Mellon University, Pittsburgh, PA
8. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for Constraint Handling Rules. In: Dahl, V., Niemelä, I. (eds.) ICLP '07. Lecture Notes in Computer Science, vol. 4670, pp. 224–239. Springer-Verlag (Sep 2007)
9. Frühwirth, T.: As time goes by: Automatic complexity analysis of simplification rules. In: Fensel, D., Giunchiglia, F., McGuinness, D., Williams, M.A. (eds.) KR '02: Proc. 8th Intl. Conf. Princ. Knowledge Representation and Reasoning. pp. 547–557 (Apr 2002)
10. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
11. Frühwirth, T.: Constraint Handling Rules-what else? In: Rule Technologies: Foundations, Tools, and Applications, pp. 13–34. Springer International Publishing (2015)
12. Gall, D., Frühwirth, T.: Exchanging conflict resolution in an adaptable implementation of ACT-R. Theory and Practice of Logic Programming 14, 525–538 (2014)
13. Gall, D., Frühwirth, T.: A Formal Semantics for the Cognitive Architecture ACT-R. In: Maurizio Proietti, H.S. (ed.) Logic-Based Program Synthesis and Transformation, 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8981. Springer (2015)
14. Gall, D., Frühwirth, T.: A refined 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. pp. 114–124. PPDP '15, ACM, New York, NY, USA (2015)
15. Langbein, J., Raiser, F., Frühwirth, T.: A state equivalence and confluence checker for CHR. In: Van Weert, P., De Koninck, L. (eds.) CHR '10. K.U.Leuven, Department of Computer Science, Technical report CW 588 (Jul 2010)
16. Raiser, F., Frühwirth, T.: Analysing graph transformation systems through Constraint Handling Rules. Theory Practice of Logic Programming 11(1), 65–109 (Jan 2011)
17. van Rooij, I., Wright, C.D., Wareham, T.: Intractability and the use of heuristics in psychological explanations. Synthese 187(2), 471–487 (2012)
18. Salvucci, D.: ACT-R: The Java Simulation & Development Environment – Homepage, `http://cog.cs.drexel.edu/act-r/`
19. Stewart, T.C., West, R.L.: Deconstructing and reconstructing ACT-R: exploring the architectural space. Cognitive Systems Research 8(3), 227–236 (2007)
20. Sun, R.: Introduction to computational cognitive modeling. In: Sun, R. (ed.) The Cambridge Handbook of Computational Psychology, pp. 3–19. Cambridge University Press, New York (2008)
21. 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)