An Operational Semantics for the Cognitive Architecture ACT-R and its Translation to Constraint Handling Rules

DANIEL GALL and THOM FRÜHWIRTH, Ulm University

Computational psychology has the aim to explain human cognition by computational models of cognitive processes. The cognitive architecture Adaptive Control of Thought – Rational (ACT-R) is popular to develop such models. Although ACT-R has a well-defined psychological theory and has been used to explain many cognitive processes, there are two problems that make it hard to reason formally about its cognitive models: First, ACT-R lacks a computational formalization of its underlying production rule system and secondly, there are many different implementations and extensions of ACT-R with many technical artifacts complicating formal reasoning even more.

This paper describes a formal operational semantics – the *very abstract semantics* – that abstracts from as many technical details as possible keeping it open to extensions and different implementations of the ACT-R theory. In a second step, this semantics is refined to define some of its abstract features that are found in many implementations of ACT-R – called the *abstract semantics*. It concentrates on the procedural core of ACT-R and is suitable for analysis of the general transition system since it still abstracts from details like timing, the sub-symbolic layer of ACT-R or conflict resolution.

Furthermore, a translation of ACT-R models to the declarative programming language Constraint Handling Rules (CHR) is defined. This makes the abstract semantics an executable specification of ACT-R. CHR has been used successfully to embed other rule-based formalisms like graph transformation systems or functional programming. There are many theoretical results and practical tools that support formal reasoning about and analysis of CHR programs. The translation of ACT-R models to CHR is proven sound and complete w.r.t. the abstract operational semantics of ACT-R. This paves the way to analysis of ACT-R models through CHR analysis results and tools. Therefore, to the best of our knowledge, our abstract semantics is the first abstract formulation of ACT-R suitable for both analysis and execution.

CCS Concepts: **•Theory of computation →Operational semantics; •Computing methodologies →Cognitive science; •Applied computing →Psychology; •Software and its engineering →Constraint and logic languages; Semantics;**

Additional Key Words and Phrases: Computational cognitive modeling, ACT-R, operational semantics, source to source transformation, Constraint Handling Rules

ACM Reference format:
Daniel Gall and Thom Frühwirth. 2018. An Operational Semantics for the Cognitive Architecture ACT-R and its Translation to Constraint Handling Rules. *ACM Trans. Comput. Logic* 0, 0, Article 0 (June 2018), 42 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Computational cognitive modeling tries to explore human cognition by building detailed computational models of cognitive processes [32]. Cognitive architectures support the modeling process by providing a formal, well-investigated theory of cognition that allows for building cognitive models of specific tasks and cognitive features.

Currently, computational cognitive modeling architectures as well as the implementations of cognitive models are typically ad-hoc constructs. They lack a formalization from the computer science point of view. For instance, *Adaptive Control of Thought – Rational (ACT-R)* [5] is a widely employed 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. It has a well-defined psychological theory, however, its computational system is not described formally leading to implementations that are full of technical artifacts as claimed in [3, 31], for instance. Due to the many technical artifacts it is hard to merge different cognitive models in a greater context.

Formal analysis of cognitive models can support the modeling process to ensure model quality and to prove certain properties of cognitive models to validate their plausibility. However, the lack of formalization, the several implementations and the technical artifacts impede formal reasoning about the underlying languages and the programmed models. It makes it hard to compare different implementation variants of the languages. Furthermore, it complicates verifying properties of the models. These issues call for a formal semantics of cognitive modeling languages together with proper analysis techniques. This semantics should be an elegant formulation of the core features all those implementations of cognitive architectures use to explain human cognition.

In this paper, we describe a very abstract formulation of the operational semantics of ACT-R, called the *very abstract semantics*. This formalization of the fundamental parts of ACT-R is the basis for the specification of concrete implementations of the theory of the architecture, as well as for analysis of cognitive models. This very abstract semantics, extending and adapting the work of [3], captures all possible kinds of ACT-R implementations. This can be too abstract to formally reason about actual computational models meaningfully.

Therefore, we give a concrete instance of this very abstract semantics that is suitable for analysis and implementation – the *abstract semantics*. It still abstracts from a variety of technical details like conflict resolution, times and latency that strongly depend on the actual implementation or even configuration of ACT-R, but includes the typical matching process of rules and some common actions that can be extended. This paves the way for computational analysis of actual cognitive models.

Eventually, we construct a sound and complete *translation* of ACT-R models in abstract semantics to Constraint Handling Rules (CHR). The abstract semantics is therefore directly executable via its CHR representation making it, to the best of our knowledge, the first formal operational semantics of ACT-R that is suitable for analysis and execution.

The paper is a revised and extended version of our prior work in [18] and [19]. It has the following contributions:

(1) A very abstract operational semantics of ACT-R (c.f. Section 3),
(2) an instance of this semantics for analysis (abstract semantics, c.f. Section 4),
(3) a translation of ACT-R models to the declarative programming language Constraint Handling Rules (CHR) (c.f. Section 5) and
(4) a soundness and completeness proof of the translation w.r.t. the operational semantics (c.f. Section 6).

The formulations of the semantics (items 1 and 2) have been improved compared to [18] and the translation (item 3) has been revised substantially compared to [19] making it suitable for the proofs. The soundness and completeness proofs (item 4) have not been published before and are fundamentally new.

Constraint Handling Rules (CHR) [13] has a formally defined operational semantics as well as a declarative semantics with corresponding soundness and completeness results. There are many

theoretical and practical results and tools for analysis of CHR programs [13]. Due to its strengths in formal program analysis and its strong relation to first order [13] and linear logic [7, 13], it has been used as a lingua franca that embeds many rule-based approaches [13] like term rewriting systems [26], graph transformation systems [24, 27] and business rules [23]. Such embeddings have been used successfully to make the analysis results of CHR available to other approaches. The sound and complete embedding of ACT-R in CHR enables the use of these results and tools to formally reason about cognitive models.

The paper is structured as follows: We first give a short introduction to ACT-R in Section 2. The formal definition of the very abstract semantics is given in Section 3 and the abstract semantics is defined as an instance in Section 4. The translation scheme of abstract ACT-R models to CHR programs is described in Section 5 and proven sound and complete w.r.t. the operational semantics in Section 6. Related work is discussed in Section 7 with a detailed comparison to prior work.

2 DESCRIPTION OF ACT-R

In this section, we describe ACT-R informally. For a detailed introduction to the theory, we refer to [4–6, 34]. Adaptive Control of Thought – Rational (ACT-R) is a popular cognitive architecture that is used in many cognitive models to describe and explain human cognition. There have been applications in language learning models [33] or in the field of human computer interaction [9]. The components of the ACT-R architecture have even been mapped to brain regions [4, chapter 2].

Using a cognitive architecture like ACT-R simplifies the modeling process, since well-investigated psychological results have been assembled to a unified theory about fundamental parts of human cognition. In the best-case, such an architecture constrains modeling to only plausible cognitive models [34]. Computational cognitive models are described clearly and unambiguously since they are executed by a computer producing detailed simulations of human behavior [32]. By performing the same experiments on humans and the implemented cognitive models, the resulting data can be compared and models can be validated.

2.1 Overview of the ACT-R Architecture

The ACT-R theory is built around a modular production rule system operating on data elements called *chunks*. A chunk is a structure consisting of a name and a set of labeled slots that are connected to other chunks. The slots of a chunk are determined by its *type*. The names of the chunks are only for internal reference – the information represented by a network of chunks comes from the connections. For instance, there could be chunks representing the cognitive concepts of numbers 1, 2, … By chunks with slots *number* and *successor* we can connect the individual numbers to an ordered sequence describing the concept of natural numbers. This is illustrated in Fig. 1.

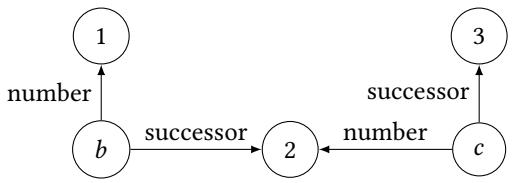

Fig. 1. Two count facts with names b and c that model the counting chain 1, 2, 3.

As shown in Fig. 2, ACT-R consists of modules. The *goal module* keeps track of the current (sub-) goal of the cognitive model. The *declarative module* contains *declarative knowledge*, i.e. factual

knowledge that is represented by a network of chunks. There are also modules for interaction with the environment like the *visual* and the *manual* module. The first perceives the visual field whereas the latter controls the hands of the cognitive agent. Each module is connected to a set of *buffers* that can hold at most one chunk at a time.

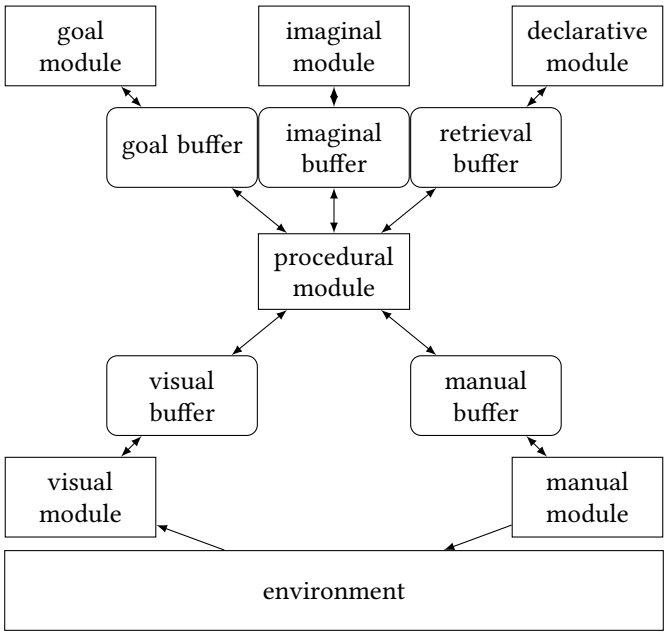

Fig. 2. Modular architecture of ACT-R. This illustration is inspired by [34] and [5].

The heart of the system is the *procedural module* that contains the production rules controlling the cognitive process. It only has access to a part of the declarative knowledge: the chunks that are in the buffers. A production rule matches the content of the buffers and – if applicable – executes its actions. There are three types of actions:

Modifications overwrite information in a subset of the slots of a buffer, i.e. they change the connections of a chunk.

Requests ask a module to put new information into its buffer. The request is encoded in form of a chunk. The implementation of the module defines how it reacts on a request. For instance, there are modules that only accept chunks of a certain form like the manual module that only accepts chunks that encode a movement command for the hand according to a predefined set of actions.

Nevertheless, all modules share the same interface for requests: The module receives the arguments of the request encoded as a chunk and puts its result in the requested buffer. For instance, a request to the declarative module is stated as a partial chunk and the result is a chunk from the declarative knowledge (the fact base) that matches the chunk from the request.

Clearings remove the chunk from a buffer.

The system described so far is the so-called *symbolic level* of ACT-R. It is similar to standard production rule systems operating on symbols (of a certain form) and matching rules that interact

with buffers and modules. However, to simulate the human mind, a notion of timing, latency, priorities etc. are needed. In ACT-R, those concepts are subsumed in the *sub-symbolic level*. It augments the symbolic structure of the system by additional information to simulate the previously mentioned concepts.

Therefore, ACT-R has a progressing simulation time. Certain actions can take some time that depends on the information from the sub-symbolic level. For instance, chunks are mapped to an *activation level* that determines how long it takes the declarative module to retrieve it. Activation levels also resolve conflicts between chunks that match the same request. The value of the activation level depends on the usage of the chunk in the model (inter alia): Chunks that have been retrieved recently and often have a high activation level. Hence, the activation level changes with the simulation time. This can be used to model learning and forgetting of declarative knowledge. Similar to the activation level of chunks, production rules have a *utility* that also depends on the context and the success of a production rule in prior applications. Conflicts between applicable rules are resolved by their utilities which serve as dynamic, learned rule priorities.

2.2 Syntax

We use a simplified syntax of ACT-R that we have introduced in [16]. It is based on sets of logical terms instead of the concatenation of syntactical elements. This enables an easier access to the syntactical parts. Our syntax can be transformed directly to the original ACT-R syntax and vice-versa.

An *ACT-R architecture* is defined over two possibly infinite, disjoint sets of (constant) symbols C and variable symbols $\mathcal{V}$. Each architecture defines a set of buffers $\mathbb{B} \subseteq C$.

An *ACT-R model* in an architecture consists of a set of types $\mathbb{T} \subseteq C$ with type definitions and a set of rules Σ. A production rule has the form $L \Rightarrow R$ where L is a finite set of buffer tests and queries. A buffer test is a first-order term of the form $=(b, t, P)$ where $b \in \mathbb{B}$ is a buffer, $t \in \mathbb{T}$ a type and $P \subseteq C \times (C \cup \mathcal{V})$ is a set of slot-value pairs (s, v) where $s \in C$ and $v \in C \cup \mathcal{V}$. This means that only the values in the slot-value pairs can consist of both constants and variables.

The architecture defines a set of action symbols A from which the right-hand side of the rules can be formed. Usually, the action symbols are defined as $A := \{=, +, -\}$ for modifications, requests and clearings respectively.

The right-hand side $R \subseteq \mathcal{A}$ of a rule is a finite set of actions where $\mathcal{A} = \{a(b, t, P) \mid a \in A, b \in \mathbb{B}, t \in \mathbb{T} \text{ and } P \subseteq C \times (C \cup \mathcal{V})\}$. This means that an action is a term of the form $a(b, t, P)$ where the functor a of the action is in A, the set of action symbols, the first argument b is a constant (denoting a buffer), the second argument is a constant t denoting a type, and the last argument is a set of slot-value pairs, i.e. a pair of a constant and a constant or variable. In this paper, we only one action per buffer is allowed, i.e. if $a(b, t, P) \in R$ and $a'(b, t', P') \in R$ then $a = a', t = t'$ and $P = P'$.

Note that the same symbol (=) is used for both buffer tests and modifications, although they denote different concepts. Nevertheless, this paper adheres to those symbols as they appear in the ACT-R reference implementation and are widely used in the context of ACT-R.

The function *vars* maps an arbitrary set of terms to its set of variables in $\mathcal{V}$. For a production rule $L \Rightarrow R$ the following must hold: $vars(R) \subseteq vars(L)$, i.e. no new variables must be introduced on the right-hand side of a rule. As we will see in the following sections about semantics, this restriction demands that all variables are bound on the left-hand side.

The syntactic restrictions on rules are summarized in the following definition of *well-formedness* as an abstract syntax description. We allow to transfer information in non-terminal symbols as

arguments and denote enumerations of non-terminal symbols as subsets.

$$
\begin{aligned}
\text{production rule: } P &::= L \Rightarrow R(L) \\
\text{left-hand side: } L &::= \{T_1, \ldots T_n\}, n > 0 \\
\text{buffer test: } T &::= =(b, t, \{ST_1(t), \ldots, ST_n(t)\}), n \geq 0 \\
\text{slot test: } ST(t) &::= (s, v) \text{ if } s \in \tau(t) \\
\text{right-hand side: } R(L) &::= \{RA_1(L), \ldots RA_n(L)\}, n > 0 \text{ if the buffers in } RA_1(L) \text{ and} \\
&\qquad\qquad RA_n(L) \text{ are disjoint} \\
\text{rule action: } RA(L) &::= a(b, t, \{SA_1(L, t_1), \ldots, SA_n(L, t_n)\}), n \geq 0 \\
\text{slot action: } SA(L, t) &::= (s, v) \text{ if } s \in \tau(t) \text{ and } v \in vars(L)
\end{aligned}
$$

where $b \in \mathbb{B}$, $t \in \mathbb{T}$, $a \in A$, $s \in C$ and $v \in \mathcal{V} \cup C$. This definition is a reduced version of the definition given in [8, p. 17 et seq.] and has only been transformed from LISP style syntax to sets of first-order terms.

2.3 Informal Operational Semantics

In this section, we describe ACT-R's operational semantics informally. The production rule system constantly checks for matching rules and applies their actions to the buffers. This means that it tests the conditions on the left hand side with the contents of the buffers (which are chunks) and applies the actions on the right hand side, i.e. modifies individual slots, requests a new chunk from a module or clears a buffer.

The left hand side of a production rule consists of buffer tests – that are terms $=(b, t, P)$ with a buffer b, a type t and a set of slot-value pairs P. The values of a slot-value pair can be either constants or variables. The test matches a buffer, if the chunk in the tested buffer b has the specified type t and all slot-value pairs in P match the values of the chunk in b. Thereby, variables of the rule are bound to the actual values of the chunk. Values of a chunk in the buffers are always ground, i.e. they do not contain any variables. This is ensured by the previously mentioned condition in the syntax of a rule that the right hand side of a rule does not introduce new variables (see Section 2.2). Hence the chunks in the buffers stay ground.

If there is more than one matching rule, a conflict resolution mechanism that depends on the sub-symbolic layer chooses one rule that is applied. After a rule has been selected, it takes a certain time (usually 50 ms) for the rule to fire. I.e. actions are applied after this delay. During that time the procedural module is blocked and no rule can match.

The right hand side consists of actions $a(b, t, P)$, where $a \in A$ is an action symbol, b is a constant denoting a buffer and P is again a set of slot-value pairs. We have already explained the three types of actions (modifications, requests and clearings) roughly. In more detail, a modification overwrites only the slots specified in P with the values from P. A request clears the requested buffer and asks a module for a new chunk. It can take some time specified by the module (and often depending on sub-symbolic values) until the request is processed and the chunk is available. During that time, other rules still can fire, i.e. requests are executed in parallel. However, a module can only process one request for a buffer at the same time. Buffer clearings simply remove the chunk from a buffer. In the following, we disregard clearings in our definitions since they are easy to add.

As mentioned before, a rule can only consist of one action per buffer. This excludes race conditions and conflicts between actions. For instance, a modification is typically applied without any delay whereas a request typically takes some time and overwrites the chunk in the requested buffer.

This would lead to losing the modification that has been applied before. The modification cannot be applied after the request, as it is not clear that the requested chunk is of the same type as the chunk that has been in the buffer before. Hence, such rules are undesired. Note that a request can still result in different chunks as requests typically are not functional but non-deterministic. In implementations, they often include a noisy component that influences the result.

In the following example, a rule is defined and its behavior is explained informally.

Example 2.1 (production rule). We want to model the counting process of a little child that has just learned how to count from one to ten. We use the natural number chunks described in Section 2.1 as declarative knowledge. Furthermore, we have a goal chunk of another type g that memorizes the current number in a *current* slot. We now define a production rule, that increments the number in the counting process (and call this rule *inc*). We denote variables with capital letters in our examples. The left-hand side of the rule *inc* consists of two tests:

- $=(goal, g, \{(current, X)\})$ and
- $=(retrieval, succ, \{(number, X), (successor, Y)\})$.

This means that the rule tests if in the goal buffer there is a chunk of type g that has some number X (which is a variable) in the *current* slot. If this number X is also in the *number* slot of the chunk in the retrieval buffer, the test succeeds and the variable Y is bound to the value in the *successor* slot. The actions of the rule are:

- $=(goal, g, \{(current, Y)\})$ and
- $+(retrieval, succ, \{(number, Y)\})$.

The first action modifies the chunk in the goal. A modification cannot change the type, hence it has to be the same type symbol as in the test. The *current* slot of the goal chunk is adjusted to the successor number Y and the declarative module is asked for a chunk of type *succ* with Y in its *number* slot. This is called a retrieval request. After a certain amount of time, the declarative module will put a chunk with Y in its *number* and $Y + 1$ in its *successor* slot into the retrieval buffer and the rule can be applied again.

3 VERY ABSTRACT OPERATIONAL SEMANTICS

Our goal is to define an operational semantics that captures as many ACT-R implementations as possible and leaves room for extensions and modifications. Hence, we first give them a common theoretical foundation that is based on the formalization according to [3] – the very abstract operational semantics. It describes the fundamental concepts of a production rule system that operates on buffers and chunks like ACT-R. This work extends the definition from [3]. We compare our work with the work from [3] in Section 7.1. Later on, in Section 4, we define an instance of this very abstract semantics that is suitable for analysis of cognitive models.

An *ACT-R architecture* is a concrete instantiation of the very abstract semantics and defines general parts of the system that are left open by the very abstract semantics like the set of possible actions A, the effect of such an action or the selection process. In contrast to that, an *ACT-R model* defines model-specific instantiations of parts like the set of types $\mathbb{T}$ and the set of production rules Σ.

3.1 Chunk Stores

As described before in Section 2, ACT-R operates on a network of typed chunks that we call a *chunk store*. Therefore, we first define the notion of types:

Definition 3.1 (chunk types). Let 2^C be the power set of all constant symbols. A *typing function* $\tau : \mathbb{T} \to 2^C$ maps each type from the set of types $\mathbb{T} \subseteq C$ to a finite set of allowed slot names. Every type set $\mathbb{T}$ must contain a special type $\mathsf{chunk} \in \mathbb{T}$ with $\tau(\mathsf{chunk}) = \emptyset$.

A chunk store is defined over a set of types and a typing function. We abstract from chunk names as they do not add any information to the system. In fact, chunks are defined as unique, immutable entities with a type and connections to other chunks:

Definition 3.2 (chunk store). A *chunk store* Δ is a set of elements of the form $c{::}(t, val)$ where $c \in C$ is a unique chunk identifier, $t \in \mathbb{T}$ 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.

Every chunk store Δ must contain a chunk $\mathtt{nil}{::}(\mathsf{chunk}, \emptyset)$. Each chunk store Δ has an *identifier function* $id_\Delta : \Delta \to C, c{::}(t, val) \mapsto c$ that returns the chunk identifier of the chunk. The inverse of id is defined as follows:

$$id_\Delta^{-1}(x) := \begin{cases} c & \text{if } id_\Delta(c) = x, \\ \mathtt{nil} & \text{otherwise.} \end{cases}$$

The set of all chunk stores over all constant symbols C is denoted by $\mathcal{D}$. Chunk identifiers can be omitted, if they do not play a role or are clear from the context, i.e. a chunk $c{::}(t, val)$ can be referred to as (t, val).

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.

Chunk identifiers are unique within one chunk store. Note that two chunks are only considered equivalent, if they have the same chunk identifier, type and value functions (in that case they are the indistinguishable in the set and therefore treated as one and the same chunk). Hence, a chunk store can contain multiple elements with the same values that still are unique entities representing different concepts. This can be seen in the following example. We model our well-known example from Fig. 1 as a chunk store.

Example 3.3 (chunk store of natural numbers). The chunk store from Fig. 1 can be modeled as follows. Note that in the examples, we use chunk identifiers to define the connections in the slots (the *val* functions) instead of the whole chunk definition for the sake of brevity.

- The set of types is $\mathbb{T}_{3.3} = \{number, succ\}$.
- The typing function $\tau_{3.3} : \mathbb{T} \to 2^C$ is defined as $\tau_{3.3}(number) = \emptyset$ and $\tau_{3.3}(succ) = \{number, successor\}$.

  Note that chunks of type *number* have no slots. A *number* chunk represents the mental concept of the number. To distinguish those chunks on the model level and give them a semantic meaning, their identifiers are number symbols. Hence, the chunk with identifier 1 will represent the mental concept of number 1. However, for the model itself, the identifier does not play a semantic role. The semantics comes only from the connections to other *number* chunks.
- We have the following chunks in our store $\Delta_{3.3}$:
  - the unique entities with identifiers $1, 2, 3$ that are defined as $(number, \emptyset)$,
  - $b :: (succ, val_b)$ with $val_b(s) = \begin{cases} 1 & \text{if } s = number \\ 2 & \text{if } s = successor \end{cases}$
  - $c :: (succ, val_c)$ with $val_c(s) = \begin{cases} 2 & \text{if } s = number \\ 3 & \text{if } s = successor \end{cases}$

From the definition of a chunk store Δ, we can derive a graph (Δ, E) where for each slot-value pair $val(s) = d$ of a chunk $c::(t, val) \in \Delta$ there is an edge $(c, d) \in E$ with label s. In the graphical representation, we can label vertices with the chunk identifiers. We can apply this to Example 3.3 to derive the graph illustrated in Fig. 1 on page 3.

Sometimes we want to build chunk stores that only have a few chunks in them that refer to chunks in other chunk stores in their slots. We call this concept a *partial chunk store.*

Definition 3.4 (partial chunk store). A *partial chunk store* with reference to a chunk store Δ, denoted as $\Delta^{\subset}$, is a set of elements of the form $c::(t, val)$ where $c \in C$ is a unique chunk identifier, $t \in \mathbb{T}$ is a chunk type and $val : \tau(t) \to \Delta \cup \Delta^{\subset}$ is a function that maps each slot of the chunk (determined by the type t) to another chunk from chunk store $\Delta \cup \Delta^{\subset}$. Every chunk in the partial chunk store $\Delta^{\subset}$ has a unique identifier that is disjoint from the identifiers in Δ. The function $id_{\Delta^{\subset}} : \Delta^{\subset} \to C$ returns the chunk identifier for each chunk in $\Delta^{\subset}$.

The set of all partial chunk stores that refer to a chunk store Δ is denoted as $\mathcal{D}_{\Delta}^{\subset}$.

Example 3.5 (partial chunk store). Let $\Delta_{3.3}$ be the chunk store from Example 3.3. We define $\Delta_{3.5}$ as a partial chunk store that refers to $\Delta_{3.3}$. It contains the chunk $x ::_{\Delta_{3.5}} (succ, val_x)$ with the following slots:

$$val_x := \begin{cases} 2 & \text{if } s = number \\ 3 & \text{if } s = successor \end{cases}$$

We define an operation that merges two (partial) chunk stores. In an abstract way it can be considered as a special set union that merges two elements of a chunk store, if they have the same chunk identifiers. However, since there are many different implementations of ACT-R, we do not want to limit our formulation to this special type of set union, but define a more general operator $\circ$. For the general understanding of the paper and the proofs it is sufficient to think of it as set union that maintains uniqueness of chunk identifiers and might merge some chunks to one.

Definition 3.6 (chunk merging). Let $\mathcal{D}$ be the set of all chunk stores, $\Delta \in \mathcal{D}$ a chunk store and $\mathcal{D}_{\Delta}^{\subset}$ the set of all partial chunk stores that refer to Δ. Then $\circ : (\mathcal{D} \cup \mathcal{D}_{\Delta}^{\subset}) \times (\mathcal{D} \cup \mathcal{D}_{\Delta}^{\subset}) \to (\mathcal{D} \cup \mathcal{D}_{\Delta}^{\subset})$ the *chunk merging operator* that merges two (partial) chunk stores to one (partial) chunk store.

We require the following properties for $\circ$. For all chunk stores $\Delta \in \mathcal{D}$ and all (partial) chunk stores $\Delta', \Delta'', \Delta''' \in \mathcal{D} \cup \mathcal{D}_{\Delta}^{\subset}$:

(1) The chunk merge operator is only defined for two (partial) chunk stores that have disjoint chunk identifiers, except for syntactically identical chunks. This means that for all $c' \in \Delta'$ and $c'' \in \Delta''$ with $id_{\Delta'}(c') = id_{\Delta''}(c'')$, it must hold that $c' = c''$, i.e. both chunks have same types and value functions.
(2) The merging is closed, i.e. if Δ' and Δ'' are (partial) chunk stores, then $\Delta' \circ \Delta''$ is a (partial) chunk store.
(3) $(\Delta' \circ \Delta'') \circ \Delta''' = \Delta' \circ (\Delta'' \circ \Delta''')$, i.e. $\circ$ is *associative.*
(4) $\emptyset$ is the *neutral element*, i.e. $\Delta' \circ \emptyset = \emptyset \circ \Delta' = \Delta'$.
(5) If $c' \in \Delta'$, then $c' \in \Delta' \circ \Delta''$ and $id_{\Delta'}(c') = id_{\Delta' \circ \Delta''}(c')$, i.e. $\Delta' \subseteq \Delta' \circ \Delta''$. This means that all elements of the left chunk store Δ' are also part of the merged chunk store.
(6) There is a mapping $map_{\Delta', \Delta''} : \Delta' \cup \Delta'' \to \Delta' \circ \Delta''$ that maps chunks from the original chunk stores to the merged chunk store. For all chunks $c := i::(t, val) \in \Delta' \cup \Delta''$: If there is a chunk $c' \in \Delta \circ \Delta'$ in the merged chunk store with the same identifier $id_{\Delta' \circ \Delta''}(c') = i$, then $c = c'$. This means that the mapping function represents the actual merging of the chunk store and just returns the new identifiers for the chunks that have been removed due to the merging. All other chunks remain untouched.

Note that this also means that $map_{\Delta',\Delta''}(c) = c$ if $c \in \Delta'$, i.e. the chunks from Δ' remain untouched by the merging (c.f. Axiom 5).

From the axioms it is clear that $(\mathcal{D} \cup \mathcal{D}_\Delta^\subset, \circ)$ is a monoid, since the structure is closed under $\circ$, associative and has a neutral element.

The simplest definition of a chunk merging operator is the set union ($\cup$). For chunk stores that obey Axiom 1 (i.e. that have disjoint identifiers except for syntactic equivalent chunks), set union is closed, associative and has neutral element $\emptyset$. It also obeys Axiom 5. The mapping function is defined as the identity function as no elements are lost in the merging process.

Example 3.7 (set union as $\circ$). Let $\Delta := \{c_1, c_2\}$ and $\Delta' := \{c_2, c_3\}$ with $id_\Delta(c_i) = i$ for $i = 1, 2$ and $id_{\Delta'}(c_i) = i$ for $i = 2, 3$. Then $\Delta \circ \Delta' = \{c_1, c_2, c_3\}$. Due to the same identifiers, types and value functions of the two appearances of c_2, the merged store only keeps one version of c_2. The mapping just returns the original chunks from both stores.

In most implementations, chunks that have the same structure are merged to one chunk regardless of identifier, i.e. for a chunk store Δ and two partial chunk stores $\Delta_1^\subset$ and $\Delta_2^\subset$ that refer to Δ: If $c_1{::}(t, val) \in \Delta_1^\subset$ and $c_2{::}(t, val) \in \Delta_2^\subset$, then c_1 and c_2 is merged to c_1 in $\Delta_1^\subset \circ \Delta_2^\subset$. This means that only c_1 is kept in the merged store. The mapping function returns

$$map_{\Delta_1^\subset,\Delta_2^\subset}(c_1) = c_1 \text{ and}$$
$$map_{\Delta_1^\subset,\Delta_2^\subset}(c_2) = c_1.$$

3.2 States

ACT-R states are tuples that consist of several parts:

- a chunk store,
- a cognitive state that defines which chunks from the chunk store are currently in which buffer. Those chunks from the chunk store are the only ones that are visible to the production system, since it can only match chunks in buffers.
- A conjunction of additional information that can be used to store information representing the sub-symbolic level, and
- a time component.

In the following the missing parts of an ACT-R state are defined formally.

Definition 3.8 (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 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. The set of all 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.

ACT-R adds a sub-symbolic level to the symbolic concepts that have been defined so far. The sub-symbolic level adds implicit information to the symbolic structures that account for neural processes usually associated with neural network models of cognition [34]. This distinguishes ACT-R from pure symbolic architectures like SOAR [34].

To gather information from the sub-symbolic layer, we add a component to the state that contains (sub-symbolic) additional information needed to calculate sub-symbolic values. This information can be altered by an abstract function as can be seen in Section 3.3. The information will be expressed as conjunctions of predicates from first-order logic. The additional information is also used to manage data used in ACT-R's modules.

Additionally, modules other than the procedural module hold their data in the additional information.

ACT-R states are defined as follows:

Definition 3.9 (very abstract state). A *very abstract state* is a tuple $\langle \Delta; \gamma; \upsilon; t \rangle$ where γ is a cognitive state in the sense of Definition 3.8, υ is a conjunction of ground, atomic first order predicates (called *additional information*), $t \in \mathbb{R}_0^+$ is a time. The state space is denoted with $\mathcal{S}_{va}$.

The chunk store Δ contains a type that is denoted by a constant and a valuation function that connects slot names (constants) to other elements from Δ. The cognitive state γ connects buffer names (constants) with chunks from Δ and a delay in $\mathbb{R}_0^+$.

The additional information component υ is a conjunction of ground, atomic predicates over a signature Υ and the time is also a number. The set of allowed predicates for additional information is denoted as Υ and must contain at least *true*. Additionally, the allowed additional information Υ is closed under conjunction. Additional information holds data of ACT-R's modules as well as sub-symbolic information. It can be considered as a bag of structured information that can be used to store values needed for the computation process as in a database. The notation of a conjunction is of only syntactical nature at first, as we do not define a logical reading for the expressions. The semantics of additional information comes from its use in the operational semantics for instance in the rule selection function or the interpretation of actions as it will be defined in Section 3.3. However, there is a close relation to additional information and built-in constraints in the translation of ACT-R to CHR as shown in Section 5 that is made explicit by this syntactic choice. Nevertheless, it is always possible to think of additional information as a (multi-)set of relational data represented as first-order predicates.

Note that in an initial state, the chunk store contains at least the chunks that appear in the rules. Additionally, a very abstract state cannot contain variables from $\mathcal{V}$, but is only compound from terms, sets and functions over constants from $\mathcal{C}$.

We continue our running example by defining a very abstract state with one of the chunks defined in Example 3.3.

Example 3.10 (ACT-R states). We want to model the counting process of a little child that has learned the sequence of the natural numbers from one to ten as declarative facts and can retrieve those facts from declarative memory. Therefore, we add a chunk of type g with a *current* slot that memorizes the current number in the counting process.

The following state has a chunk of type g in the goal buffer that has the current number 1. The retrieval buffer is currently retrieving the chunk b with number 1 and successor 2. The retrieval is finished in one second as denoted by the delay. Fig. 3 illustrates the state. The formal definition is:

- In addition to Example 3.3, we add a chunk type g with a *current* slot that takes track of the current number in the counting process.
  - $\mathbb{T}_{3.10} = \mathbb{T}_{3.3} \cup \{g\}$ where $\mathbb{T}_{3.3}$ is the set of types from Example 3.3.
  - $\tau_{3.10}(t) = \begin{cases} \{current\} & \text{if } t = g \\ \tau_{3.3}(t) & \text{otherwise.} \end{cases}$
- We define an initial goal chunk c_g of this new type g as $c_g := (g, val_{goal})$ where

$$val_{goal}(current) = 1.$$

- This chunk and the number chunks from Example 3.3 are part of a chunk store

$$\Delta_{3.10} := \Delta_{3.3} \uplus \{c_g\}.$$

- The initial state in this example is defined as $\sigma_0 := \langle \Delta_{3.10}; \gamma_0; \mathit{true}; 1 \rangle$. It is a very abstract ACT-R state with the chunk store $\Delta_{3.10}$, a cognitive state γ_0, empty additional information and with current time 0.
- The cognitive state γ_0 of the very abstract state σ_0 is defined as follows:
  - $\gamma_0(\mathit{goal}) = (c_g, 0)$, i.e. the *goal* buffer holds the initial goal chunk c_g. It has the delay 0, i.e. it is directly available to the procedural system.
  - $\gamma_0(\mathit{retrieval}) = (b, 1)$ where b is defined as in Example 3.3. The delay 1 indicates that this chunk is not yet available to the procedural system. This represents the situation where the chunk b has been requested from the declarative module and the request is still processed.

The current time in the state is 1. The delay of the goal buffer is 0, i.e. its chunk is visible to the procedural system. The delay of the retrieval buffer is 1, i.e. the chunk will be visible when the global time of the state is 2.

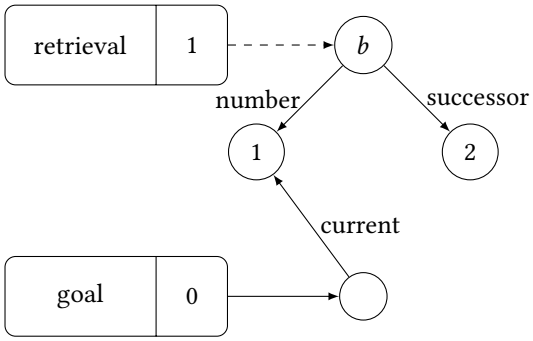

Fig. 3. Visual representation of the very abstract state defined in Example 3.10. The dashed arrow signifies that the chunk in the retrieval buffer is not yet visible (as indicated by the delay right of the buffer's name).

In Example 3.10, the additional information component was empty. This is not completely accurate, since the declarative module has its own chunk store – called *declarative memory* – that is a subset of the global chunk store. To resolve requests, only the chunks in this separate chunk store are considered. In the following example, additional information for the declarative memory is defined.

Example 3.11 (additional information). Chunks in the declarative memory can be represented by predicates of the form $\mathit{dmchunk}(c, t)$ where c is a chunk identifier and t a type. The slot connections can be represented by predicates of the form $\mathit{dmslot}(c, s, c')$ that models the connection of slot s in chunk c to chunk c'.

The declarative memory consisting of the chunks b, 1 and 2 can be represented as follows:

$$\mathit{dmchunk}(1, \mathit{number}) \wedge \mathit{dmchunk}(2, \mathit{number}) \wedge$$
$$\mathit{dmchunk}(b, \mathit{succ}) \wedge \mathit{dmslot}(b, \mathit{number}, 1) \wedge \mathit{dmslot}(b, \mathit{successor}, 2).$$

In practice, the chunks in the declarative memory are complemented with some sub-symbolic information, namely *activation levels* that influence the latency of a retrieval request and that can model learning and forgetting.

3.3 Operational Semantics

In this section, the state transition system of the very abstract semantics is defined. First of all, the set of applicable rules has to be defined. Therefore, a selection function S is introduced that is defined by the architecture and maps a state to a set of applicable rules and the variable bindings implied by the application of the rule.

Definition 3.12 (selection function). Let $\Theta(\mathcal{V}, C)$ be the set of possible ground substitutions over variables $\mathcal{V}$ and constants C that replace every variable from $\mathcal{V}$ with at most one value from C. The set $2^{\Sigma \times \Theta(\mathcal{V}, C)}$ denotes the power set of all tuples (r, θ) where $r \in \Sigma$ is a rule and $\theta \in \Theta(\mathcal{V}, C)$ is a ground substitution of variables from $\mathcal{V}$ with constants from C as described before.

A *selection function* is a function $S : \mathcal{S}_{va} \to 2^{\Sigma \times \Theta(\mathcal{V}, C)}$ that maps a state to a set of pairs (r, θ) where $r \in \Sigma$ is a production rule and $\theta \in \Theta(\mathcal{V}, C)$ is a substitution of variables from $\mathcal{V}$ with constants from C, such that all variables from the rule r are substituted, i.e. $dom(\theta) = vars(r)$.

The function S usually defines a notion of matching and makes sure that only rules that match visible information in the buffers can fire (i.e. only chunks that are not delayed are considered). In implementations the set of applicable rules typically is restricted to none or one element. The reason is that the set of syntactically matching rules is filtered using sub-symbolic information to get rid of non-determinism. Abstract definitions of ACT-R can allow more than one rule to be applicable.

To define the modification of a state by a transition, we define interpretation functions of actions that determine the possible effects of an action.

Definition 3.13 (interpretation of actions). Let $\mathcal{S}_{va}$ be the set of all very abstract states and $2^{\mathcal{D}^{\subseteq}_{\Delta}}$ the power set of all partial chunk stores that refer to Δ. An *interpretation of an action* is a function $I : \mathcal{A} \times \mathcal{S}_{va} \to 2^{\mathcal{D}^{\subseteq}_{\Delta} \times \Gamma_{\text{part}} \times \Upsilon}$ that maps a tuple of an action and a very abstract state to a set of tuples. Each tuple consists of a partial chunk store, a partial cognitive state and additional information. The following conditions must hold: $(\Delta^*, \gamma^*, \upsilon^*) \in I(\alpha, \sigma)$ if

(1) $I(\alpha, \sigma) \neq \emptyset$, i.e. the interpretation of an action has at least one effect,
(2) $I(\alpha, \langle \Delta, \gamma, \mathit{true} \wedge \upsilon, t \rangle) = I(\alpha, \langle \Delta, \gamma, \upsilon, t \rangle)$, i.e. *true* is the neutral element of additional information and has no effect on the processing of actions,
(3) the resulting chunk store Δ^* is a partial chunk store that refers to Δ and whose chunk identifiers are disjoint from Δ,
(4) the co-domain of γ^* is $\Delta^* \times \mathbb{R}_0^+$, i.e. the cognitive state can only refer to chunks in the resulting chunk store Δ^*, and
(5) if the action α has the buffer b in its scope, i.e. $\alpha := (b, t, p)$, then the resulting partial cognitive state γ^* has only b in its domain, i.e. $dom(\gamma^*) = \{b\}$.

An interpretation maps each state and action of the form $a(b, t, P)$ – where $a \in A$ is an action symbol, $b \in C$ a constant denoting a buffer, $t \in C$ a type, and $P \subseteq C \times (C \cup \mathcal{V})$ is a set of slot-value pairs – to a tuple $(\Delta^*, \gamma^*, \upsilon^*)$. Thereby, Δ^* is a partial chunk store that refers to Δ, γ^* is a partial cognitive state, i.e. a partial function that assigns only the buffer b from the action to a chunk and a delay. The partial cognitive state γ^* 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 the additions to the sub-symbolic level induced by the action.

Note that the interpretation of an action can return more than one possible effect that can be chosen non-deterministically. This is used in the abstract semantics where due to the lack of sub-symbolic information all possible effects have to be considered. For example, the declarative module can find more than one chunk matching the retrieval request. Usually, by comparing

activation levels of chunks, one chunk will be returned. However, in the abstract semantics all matching chunks are possible. In the refined semantics, we restrict the selection to one possible effect as proposed by the ACT-R reference manual [8]. This is achieved by defining an interpretation function that returns a set of at most one effect.

Example 3.14 (interpretation of an action). In this example we define a *neutral effect*. We will see later that if this effect is applied to a state, the state does not change modulo time.

Let $\alpha := a(b, t, p)$ be our action that produces the neutral effect and $\sigma := \langle \Delta; \gamma; v; t \rangle$ an ACT-R state. We define

$$I(\alpha, \sigma) := \{(\{c\}, \gamma', \mathit{true})\}$$

where $\gamma'(b) := (c, d)$ for a buffer b with $\gamma(b) = (c, d)$ for some chunk $c \in \Delta$ and some delay $d \in \mathbb{R}_0^+$. The partial cognitive state γ' has the domain $\{b\} \subseteq \mathbb{B}$ and is undefined for all other buffers.

Intuitively, this action returns a chunk store with only one chunk c that has the same type and value as the chunk $\gamma(b)$ that has been in buffer b in the state σ. The resulting partial cognitive state just links b to this new chunk c. No additional information (*true*) is added.

This is a valid interpretation, since $I(\alpha, \sigma) \neq \emptyset$ and $\Delta^* := \{c\}$ is a valid partial chunk store that refers to Δ. Therefore $\mathit{val} : \tau(t) \to \Delta$ is a valid value function.

To combine interpretations of all actions of a rule, we first define how two interpretations can be combined. Therefore, we introduce the following set operator, that combines two sets of sets:

Definition 3.15 (combination operator $\sqcup$ for effects). Let e, f be effects of two actions, i.e. results of an interpretation function of an action in a state σ with chunk store Δ, i.e. $e \in I(\alpha', \sigma)$ and $f \in I(\alpha'', \sigma)$. Let $e := (\Delta', \gamma', v')$ and $f := (\Delta'', \gamma'', v'')$ where Δ', Δ'' are partial chunk stores with disjoint chunk identifiers that refer to Δ, $\gamma' : \mathbb{B}' \to \Delta \circ \Delta' \times \mathbb{R}_0^+, \gamma'' : \mathbb{B}'' \to \Delta \circ \Delta'' \times \mathbb{R}_0^+$ are partial cognitive states with disjoint domains, i.e. $\mathbb{B}', \mathbb{B}'' \subseteq \mathbb{B}$ and $\mathbb{B}' \cap \mathbb{B}'' = \emptyset$ and v is a conjunction of first-order predicates. Then the *combination* of the effects e and f w.r.t. a chunk store Δ is defined as

$$e \sqcup f := (\Delta' \circ \Delta'', \gamma, v' \wedge v'')$$

where $\gamma : \mathbb{B}' \cup \mathbb{B}'' \to (\Delta' \circ \Delta'') \times \mathbb{R}_0^+$ with

$$\gamma(b) := \begin{cases} \mathit{map}_{\Delta', \Delta''}(\gamma'(b)) & \text{if } b \in \mathit{dom}(\gamma'), \\ \mathit{map}_{\Delta', \Delta''}(\gamma''(b)) & \text{if } b \in \mathit{dom}(\gamma''). \end{cases}$$

The intuition behind this definition is that two effects of an action, i.e. two triples of chunk store, cognitive state and additional information, are merged to one effect that combines them. Hence, we get a merged partial cognitive state that has the combined buffer-chunk mappings of the two original cognitive states. This is possible, since the domains of the partial cognitive states are required to be disjoint.

The partial chunk stores are merged to one partial chunk store referring to the same total chunk store. Note that from the definition of $\circ$, the chunk store of the first effect is a subset of the merged store. However, the second chunk store might have lost some members. The mapping function assigns every chunk from the second store one from the merged store.

The merged additional information is a conjunction of the additional information from both effects.

The combination is well-defined: $\Delta' \circ \Delta''$ exists since Δ' and Δ'' are partial chunk stores that refer to the same chunk store Δ. Additionally, they have disjoint identifiers because they are results of an interpretation function (see Definition 3.13) and therefore their merging cannot fail due to different chunks with same identifiers.

The cognitive state is valid, since it has the combined domain of γ' and γ'' and just maps the chunks from those original partial cognitive states to their versions in the merged chunk store by *map*, i.e. the co-domain of γ is just the merge product of the co-domains of γ' and γ'' and keeps their connections of buffers to chunks.

The definition of combinations of effects can be lifted to sets of effects by the following definition. Let E and F be two sets of effects in some state σ with chunk store Δ, then their combination is defined as all possible pairwise combination of their elements:

$$E \sqcup F := \{e \sqcup f \mid e \in E \wedge f \in F\}.$$

Since the interpretation of an action is possibly non-deterministic, i.e. might have more than one effect triple, the combination of such sets of effect triples is a set that combines each effect from the first set with each effect from the second set. This leads to a set of combined effects from which the transition system will be able to choose one non-deterministically. However, since every effect set is required to have at least one effect, the same applies for their combination.

The interpretation function $I : \Sigma \times \times \mathcal{S}_{va} \to 2^{\mathcal{D} \times \Gamma_{\text{part}} \times \Upsilon}$ that maps a rule to all its possible effects (i.e. chunk store, cognitive state and additional information) in a given state is defined as follows:

Definition 3.16 (interpretation of rules). In a state $\sigma := \langle \Delta; \gamma; v; t \rangle$, a rule $r := L \Rightarrow R$ is interpreted by an interpretation function $I : \Sigma \times \mathcal{S}_{va} \to 2^{\mathcal{D}_\Delta^\subseteq \times \Gamma_{\text{part}} \times \Upsilon}$. It maps a rule and a state to a set of tuples of partial chunk stores, partial cognitive states and additional information, i.e. a set of all possible effects of the rule in the given state. The interpretation function is defined as follows: $I(r, \sigma)$ applies the function $apply_r$ to all tuples in the result set when combining the individual actions of the rule:

- $apply : \mathcal{D}_\Delta^\subseteq \times \Gamma_{\text{part}} \times \Upsilon \to \mathcal{D}_\Delta^\subseteq \times \Gamma_{\text{part}} \times \Upsilon$ is a function that applies some more effects at the end of the rule application and is defined by the architecture. It maps a tuple of a partial chunk store, a partial cognitive state and a conjunction of additional information to another tuple of the same kind.
- For all actions $\alpha \in R$, the resulting chunk stores are disjoint (i.e. chunk identifiers are renamed apart).
- The interpretation has the result

  $$I(r, \sigma) = apply_r(\bigsqcup_{\alpha \in R} I(\alpha, \sigma))$$

  where the combination operator $\sqcup$ refers to Δ and the *apply* function is applied to each member of the combination set. Hence, all possible effects of the rules are combined and each of the resulting partial cognitive states is then modified by the *apply* function that is defined by the architecture.

The *apply* function can apply additional changes to the state that are not directly defined by its actions. For instance, it can change some sub-symbolic values that depend on the rule application like the utility of the rule itself. Note that by definition of the ACT-R syntax it is ensured that each of the γ_{part} in the combination of the individual actions is still a function, since only one action per buffer is allowed as defined in Section 2.2.

It is important to mention, that there are two types of non-determinism in the interpretation of a rule:

(1) The first non-determinism comes from the non-deterministic nature of interpretations of an action. Each action can lead to different results (depending on their definition). This is why all interpretation functions have power sets of effects as co-domain.
(2) The second type of non-determinism comes from the definition of the combination operator that merges chunk stores by using the chunk merging operator $\circ$. Since $\circ$ is not required

to be commutative, the result of the merged chunk stores may vary. This leads to possibly differing chunk identifiers. It is possible to abstract from this kind of non-determinism by introducing the concept of (graph) isomorphism on chunk stores.

The operational semantics is defined as the state transition system $(\mathcal{S}_{va}, \rightarrowtail)$:

Definition 3.17 (very abstract operational semantics). In the *very abstract operational semantics* of ACT-R, the transition relation $\rightarrowtail: \mathcal{S}_{va} \times \mathcal{S}_{va}$ over very abstract states is defined as follows:

Apply For a rule r the following transitions are possible:

$$\frac{(r,\theta) \in S(\sigma), (\Delta^*, \gamma^*, \upsilon^*) \in I(r\theta, \sigma)}{\sigma := \langle \Delta; \gamma; \upsilon; t\rangle \rightarrowtail^{r}_{\textbf{apply}} \langle \Delta \circ \Delta^*; \gamma'; \upsilon \wedge \upsilon^*; t'\rangle}$$

where

- $\gamma' : \mathbb{B} \to \Delta \circ \Delta^*$,
  $\gamma'(b) := \begin{cases} (map_{\Delta,\Delta^*}(c), d) & \text{if } \gamma^*(b) = (c,d) \text{ is defined} \\ (map_{\Delta,\Delta^*}(c), d \ominus \delta) & \text{otherwise, if } \gamma(b) = (c,d), \end{cases}$
- $x \ominus y := \begin{cases} x - y & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$ for two numbers $x, y \in \mathbb{R}_0^+$, and
- $t' = t + \delta$ for a delay $\delta \in \mathbb{R}_0^+$ defined by the concrete instantiation of ACT-R.

When applying the rule, the resulting partial chunk store Δ^* is merged with the chunk store Δ from the state. Hence, $\Delta \circ \Delta^* = \Delta \circ \Delta^*_{\alpha_1} \circ_\Delta \ldots \circ_\Delta \Delta^*_{\alpha_n}$ for all actions α_i on the right-hand side of the rule. Note that $\Delta \subseteq \Delta \circ \Delta^*$, i.e. all chunks in Δ also appear in the merged chunk store with preservative chunk identifiers by Definition 3.6 of the chunk merging.

The partial cognitive state that comes from the interpretation of the rule replaces all positions in the original cognitive state where it is defined, otherwise the original cognitive state remains untouched. Note that for all buffers $b \in \mathbb{B}$ and $b \notin dom(\gamma^*)$ with $\gamma(b) = (c,d)$ we can also write $\gamma'(b) := (c, d \ominus \delta)$ instead of $\gamma'(b) := (map_{\Delta,\Delta^*}(c), d \ominus \delta)$ since the chunk merging guarantees that $\Delta \subseteq \Delta \circ \Delta^*$ with preservative identifiers as mentioned before.

The delays are taken from the partial cognitive state γ^* or are reduced by a constant amount that models progression of time.

When it is clear from the context, we just use $\rightarrowtail^r$ to denote that the transition applies rule r.

No Rule

$$\frac{C(\sigma)}{\sigma := \langle \Delta; \gamma; \upsilon; t\rangle \rightarrowtail_{\textbf{no}} \langle \Delta; \gamma'; \upsilon; \vartheta(\sigma)\rangle}$$

where

- $C \subseteq \mathcal{S}_{va}$ is a side condition in form of a logical predicate,
- $update : \mathcal{S}_{va} \to \Gamma_{\text{part}}$ a function that describes how the cognitive state should be transformed,
- $\vartheta : \mathcal{S}_{va} \to \mathbb{R}_0^+$ a function that describes the time adjustment in dependency of the current state, and
- $\gamma'(b) := \begin{cases} update(\sigma)(b) & \text{if defined} \\ \gamma(b) & \text{otherwise} \end{cases}$ is the updated state.

An *apply* transition applies a rule that satisfies the conditions of the selection function S by overwriting the cognitive state γ with the result from the interpretations of the actions of rule r. Thereby, one possible combination of all effects of the actions is considered. Note that the transition is also possible for all other combinations. Only the buffers with a new chunk are overwritten,

Table 1. Parameters of the very abstract semantics that must be defined by the architecture or the cognitive model respectively.

Architecture		Model	
C	set of constants	$\mathbb{T}$	set of types
$\mathcal{V}$	set of variables	τ	typing function
$\mathbb{B}$	set of buffers	Σ	set of rules
A	set of action symbols	σ_S	start state
δ	rule delay		
Υ	allowed additional information		
$\circ$	chunk merging operator		
S	rule selection function		
I	interpretation functions		
update	transformation of cognitive state after no rule transition		
ϑ	progress of time after no rule transition		

the others keep their contents. The same applies for parameters: They keep their value except for those where v^* defines a new value. Additionally, the rule application can take a certain time δ that is defined by the architecture. Time is forwarded by δ, i.e. the time in the state is incremented by δ and the delays in the cognitive state that determine when a chunk becomes visible to the system are decremented by δ (with a minimal delay of 0).

The *no rule* transition defines what happens if there is no rule applicable, but there are still effects of e.g. requests that can be applied. This means that there are buffers $b \in \mathbb{B}$ with $\gamma(b) = (c, d)$ and $d > 0$, i.e. information that is not visible to the production rule system. In that case there are no possible transitions in the original semantics. We generalized this case in our definition of the *no rule* transition that allows state transitions without rule applications. It ensures that if a side condition $C(\sigma)$ defined by the ACT-R instantiation, the cognitive state is updated according to the function *update* and the current time of the system is set to a specified time $\vartheta(\sigma)$. Both functions are also defined by the concrete architecture. This makes new information visible to the production system and hence new rules might fire. In typical ACT-R implementations, the side condition $C(\sigma)$ is that $S(\sigma) = \emptyset$, i.e. that no rule is applicable, and $\vartheta(\sigma) := t + d^*$ where σ has the time component t and d^* is the minimum delay in the cognitive state of σ. This means that time is forwarded to the minimal delay in the cognitive state and makes for instance pending requests visible to the production rule system. It can be interpreted like if the production rule system waits with the next rule application until there is new information present that leads to a rule matching the state. This behavior coincides with the specification from the ACT-R reference implementation [8]. If no transition is applicable in a state σ, i.e. there is no matching rule and no invisible information in σ, then σ is a final state and the computation stops.

The definition of our very abstract semantics leaves parts to be defined by the actual architecture and the model. Table 1 summarizes what has to be defined by an architecture and a model.

In the following, it is shown that the neutral effect from Example 3.14 is a neutral element of the rule application except for the time component.

Example 3.18 (neutral element of rule applications). Let $\sigma := \langle \Delta; \gamma; v; t \rangle$ be an ACT-R state and $r := L \Rightarrow \{\alpha\}$ an ACT-R rule with only one action $\alpha := a(b^*, t^*, P^*)$. Let $\gamma(b^*) := (c^*, d^*)$.

We define the neutral effect from Example 3.14 $e := I(\alpha, \sigma) := \{(\{c^*\}, \gamma^*, \mathit{true})\}$ as the only effect of α. By definition of the neutral effect, $\gamma^*(b^*) := (c^*, d^*)$, i.e. the same chunk and delay as in the original cognitive state γ and undefined for all other inputs. Then

$$I(r, \sigma) := \{(\{c^*\}, \gamma^*, \mathit{true})\}.$$

Let $\sigma \rightarrowtail^r \sigma'$. Then $\sigma' := \langle \Delta \circ \{c^*\}; \gamma'; \upsilon \wedge \mathit{true}; t' \rangle$. The follow-up cognitive state γ' is

$$\gamma'(b) := \begin{cases} (map_{\Delta, \{c^*\}}(c), d) & \text{if } \gamma^*(b) = (c, d) \text{ is defined} \\ (map_{\Delta, \{c^*\}}(c), d \ominus \delta) & \text{otherwise, if } \gamma(b) = (c, d). \end{cases}$$

This can be reduced to

$$\gamma'(b) := \begin{cases} (c^*, d) & \text{if } b = b^* \\ (c, d \ominus \delta) & \text{otherwise, if } \gamma(b) = (c, d), \end{cases}$$

Since c^* has the same type and value function, $\gamma'(b)$ can be considered equivalent to $\gamma(b)$ modulo delays for all $b \in \mathbb{B}$. The action only adds the predicate *true* to the conjunction of additional information which is ignored when actions are processed, i.e. effectively no information is added. Hence, σ is equivalent to σ' modulo delays and the time component that comes from the pure rule application. of an

4 ABSTRACT SEMANTICS AS INSTANCE OF THE VERY ABSTRACT SEMANTICS

The *abstract semantics* is defined as an instance of the very abstract semantics. It is suitable for the analysis of procedural core of cognitive models because it abstracts from timings and conflict resolution by leaving parts of the transition system to non-deterministic choices and still giving room for extensions and modifications. The idea is to define the minimal core of all implementations of ACT-R's procedural system disregarding parameter choices, timings, sub-symbolic information and module configuration. The abstract semantics captures all possible state transitions the procedural system can make.

4.1 Definition of the Abstract Semantics

Since it is a central part of the procedural system of ACT-R, we first define the notion of matchings:

Definition 4.1 (matching). A buffer test $\beta := =(b, t, P)$ for a buffer $b \in \mathbb{B}$ testing for a type $t \in \mathbb{T}$ and slot-value pairs $P \subseteq C \times (C \cup \mathcal{V})$ *matches* a state $\sigma := \langle \Delta; \gamma; \upsilon; x \rangle$, written $\beta \sqsubseteq \sigma$, if and only if there is a substitution $\theta : \mathcal{V} \to C$ such that $\gamma(b) = (c::(t, \mathit{val}), 0)$ and for all $(s, v) \in P : id_\Delta(\mathit{val}(s)) = v\theta$. We define $\mathit{Bindings}(\beta, \sigma) := \theta$ as the function that returns the smallest substitution that satisfies the matching $\beta \sqsubseteq \sigma$.

This definition can be extended to rules: A rule $r := L \Rightarrow R$ matches a state σ, written as $r \sqsubseteq \sigma$, if and only if for all buffer tests $t \in L$ match σ. The function $\mathit{Bindings}(r, \sigma)$ returns the smallest substitution that satisfies the matching $r \sqsubseteq \sigma$.

This means that a buffer test matches a state, if the tested buffer contains a chunk of the tested type and all slot tests hold in the state, i.e. the variables in the test can be substituted by values consistently such that they match the values from the state. The values in the test are denoted by the identifiers of the chunks. 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.

We give the architectural parameters that are left open in the very abstract semantics:

States We set the time in every state to $t := 0$ (or any other constant) because abstract states are not timed. Hence, each abstract state is a tuple $\langle \Delta; \gamma; \upsilon; 0 \rangle$ where $\gamma \in \Gamma$ is a cognitive state. We sometimes project an abstract state $\langle \Delta; \gamma; \upsilon; 0 \rangle$ to $\langle \Delta; \gamma; \upsilon \rangle$ for the sake of brevity.

Selection Function The rule selection in the abstract semantics is simply defined as $S_{abs}(\sigma) := \{(r, Bindings(r, \sigma)) \mid r \in \Sigma \wedge r \sqsubseteq \sigma\}$. Hence we select all matching rules in state σ and replace the variables from the rules by their actual values from the matching, since in the state transition system the substitution θ is applied to the rule when calculating the effects.

Effects For a state $\sigma = \langle \Delta; \gamma; \upsilon; 0 \rangle$ with $\gamma(b) = (i_b::(t, val_\gamma), d)$, the interpretation function I_{abs} for actions $A := \{=, +\}$ in the abstract semantics is defined as follows:

- $I_{abs}(= (b, t, P), \sigma) = \{(\Delta^*, \gamma^*, true)\}$ for modifications
  where
  - $c := i_c::(t, val_b)$ for a fresh id $i_c \in C$,
  - $\Delta^* := \{c\}$,
  - $\gamma^*(b) := (c, 0)$, and
  - the new slot values are:

$$val_b(s) := \begin{cases} id_\Delta^{-1}(\upsilon) & \text{if } (s, \upsilon) \in P \\ val_\gamma(s) & \text{otherwise.} \end{cases}$$

    This means that a modification creates a new chunk that modifies only the slots specified by P and takes the remaining values from the chunk that has been in the buffer. Note that the type cannot be modified, since the resulting chunk always has the type derived from the chunk that has previously been in the buffer. Modifications are deterministic, i.e. that there is only one possible effect. The slot-value function val_b is well-defined, since it appears in a partial chunk store that references Δ, it has Δ as co-domain by Definition 3.4 of a partial chunk store.
    If the action contains a slot-value pair (s, υ) that modifies s to a chunk that was not existent in the original state σ, this chunk is not magically constructed, since we do not know its type or values. Instead, we map this slot to `nil`. This comes from the definition of id_Δ^{-1}, which is `nil` for υ, since the chunk referenced by υ does not exist in Δ.
- $(\Delta^*, \gamma^*, \upsilon^*) \in I_{abs}(+(b, t, P), \sigma)$ for requests
  if
  - $request_b : \mathbb{T} \times 2^{C \times (C \cup \mathcal{V})} \times \Upsilon \to 2^{\Delta \times \mathbb{R}_0^+ \times \Upsilon}$ 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 (c, d, υ) of a chunk c, delay d and parameter valuation function υ.
  - For all $(c^*, d^*, \upsilon^*) \in request_b(t, P, \upsilon)$ we set $\gamma^*(b) := \begin{cases} (c^*, 1) & \text{if } d^* > 0 \\ (c^*, 0) & \text{otherwise,} \end{cases}$
    and $\Delta^* := \{c^*\}$.

  Note that the additional information in the result of the request is directly added to the result of the interpretation function to update the internal state of the requested module.
- The function *apply* from Definition 3.16 that adds additional changes to the state when a rule is applied is defined as the identity function, i.e. no changes to the state are introduced by the rule application itself but only by its actions.

Rule Application Delay The delay of a rule application is set to $\delta := 0$, since the abstract semantics does not care about timings.

No Rule Transition In the *no rule* transition, there are three parameters to be defined by the actual ACT-R instantiation: The side condition C, the state update function *update* and the time adjustment function ϑ. We define them for a state $\sigma := \langle \gamma; \upsilon; t \rangle$ as follows:

- $\sigma \in C$ if and only if there is a $b^* \in \mathbb{B}$ such that $\gamma(b^*) = (c, d)$ with $d > 0$, i.e. there is a buffer with a chunk that is not visible to the system. Those are the cases where there is a pending request. This means that the *no rule* transition is possible as soon as there is at least one pending request. We call the buffer of one such request b^*.
- $[update(\sigma)](b^*) := (c, 0)$ if $\gamma(b^*) = (c, d)$ and $d > 0$ for one $b^* \in \mathbb{B}$. This means that one pending request is chosen to be applied (the one appearing in C). Since this is a rule scheme and b^* can be chosen arbitrarily, the transition is possible for all assignments of b^*. This coincides with the original definition of our abstract semantics where one request is chosen from the set of pending requests.
- The function ϑ that determines how the time is adjusted after a chunk has been made visible is defined as $\vartheta(\sigma) := t$, i.e. the time is not adjusted.

Table 2 shows the parameters of the abstract semantics that have to be defined by the architecture.

Table 2. Parameters of the abstract semantics that must be defined by the architecture.

	Architecture
C	set of constants
$\mathcal{V}$	set of variables
$\mathbb{B}$	set of buffers
A	set of action symbols
δ	rule delay
Υ	allowed additional information
$\circ$	chunk merging operator
$request_b$	result of requests

We summarize the transition scheme of the abstract semantics:

Rule transition

$$\frac{r \sqsubseteq \sigma \wedge \theta = Bindings(r, \sigma) \wedge (\Delta^*, \gamma^*, \upsilon^*) \in I(r\theta, \sigma)}{\sigma := \langle \Delta; \gamma; \upsilon \rangle \rightarrowtail^r_{\textbf{apply}} \langle \Delta \circ \Delta^*; \gamma'; \upsilon \wedge \upsilon^* \rangle}$$

where $\gamma' : \mathbb{B} \to \Delta \circ \Delta^*$,

$$\gamma'(b) := \begin{cases} (map_{\Delta, \Delta^*}(c), d) & \text{if } \gamma^*(b) = (c, d) \text{ is defined} \\ (map_{\Delta, \Delta^*}(c), d) & \text{otherwise, if } \gamma(b) = (c, d). \end{cases}$$

Again, since $\Delta \subseteq \Delta \circ \Delta^*$ with preservative chunk identifiers, we can also write

$$\gamma'(b) := \begin{cases} (map_{\Delta, \Delta^*}(c), d) & \text{if } \gamma^*(b) = (c, d) \text{ is defined} \\ (c, d) & \text{otherwise, if } \gamma(b) = (c, d). \end{cases}$$

No rule transition

$$\frac{\gamma(b^*) = (c^*, d^*) \wedge d^* > 0}{\sigma := \langle \Delta; \gamma; \upsilon; t \rangle \rightarrowtail_{\textbf{no}} \langle \Delta; \gamma'; \upsilon \rangle}$$

$$\text{where } \gamma'(b) := \begin{cases} (c^*, 0) & \text{if } b = b^* \\ \gamma(b) & \text{otherwise.} \end{cases}$$

The running example is extended by a derivation in the abstract semantics:

Example 4.2 (abstract semantics). We begin with the state $\sigma_0 = \langle \Delta_{3.10}; \gamma_0; \mathit{true}; 1 \rangle$ from Example 3.10. It is visualized in Fig. 3. We extend the state with additional information as described in Example 3.11. Thereby, the same chunks as in the global chunk store Δ. Additionally, we change the time to 0 according to the definition of the abstract semantics. This results in the following state:

$$\sigma_0' = \langle \Delta_{3.10}; \gamma_0; DM_0; 0 \rangle$$

where DM_0 is the declarative memory containing all chunks from Δ. Then, the following derivations are possible:

$$\begin{aligned} \sigma_0' &\rightarrowtail^{no} \langle \gamma_1; DM_0; 0 \rangle \\ &\rightarrowtail^{inc} \langle \gamma_2; DM_0; 0 \rangle =: \sigma_2 \end{aligned}$$

where

- $\gamma_1(\mathit{retrieval}) = (b, 0)$ (and $\gamma_1(\mathit{goal}) = \gamma_0(\mathit{goal})$ as in σ_0),
- $\gamma_2(\mathit{retrieval}) = (c, 1)$ and
- $\gamma_2(\mathit{goal}) = ((g, \mathit{val}_{g_2}, 0)$ where $\mathit{val}_{g_2}(\mathit{current}) = 2$.

In σ_0 no rule is applicable, but there is a pending request whose result is not visible for the production system. Hence, we can apply the *no rule* transition which makes the chunk b visible. Then the rule *inc* from Example 2.1 is applicable.

If we assume that $\mathit{request}_{\mathit{retrieval}}(\mathit{succ}, \{(\mathit{number}, 2)\}, v) = (c_{\mathit{req}}, 1, \emptyset)$ for all additional information v where $c_{\mathit{req}} = (\mathit{succ}, \{(\mathit{number}, 2), (\mathit{successor}, 3)\})$, i.e. a chunk of type *succ* with the number 2 in the *number* slot and 3 in the *successor* slot, we reach the state σ_2 that is illustrated in Fig. 4.

Note that in this state σ_2 the *no rule* transition is possible again. This transition just chooses one invisible chunk non-deterministically and makes it visible. It can always be applied non-deterministically. This means that if there was another rule applicable in σ_2, either the rule can be applied or the no rule transition is used.

The additional information does not change in the rewriting process. This comes from the definition of the *no rule* transition that never changes the additional information component. Furthermore, modifications do not change the additional information according to their definition. Requests to the declarative memory do not change additional information as well in the setting of our example as we only consider symbolic chunk structures in the declarative memory. As no new chunks are created or added to the declarative memory in a retrieval request, no additional information is added.

In general, requests can add additional information according to their definition. If sub-symbolic information of the structures in the declarative memory is considered, then a request to it will add information about the retrieval of a chunk to consider it when calculating activation levels. Since we abstract from those features, a retrieval request does not add any additional information.

4.2 Discussion

The abstract semantics formalizes the symbolic part of the ACT-R architecture. This means, that the procedural system and its interaction with the modules are in focus. Additionally, it captures all transitions that could be possible in an ACT-R model without the knowledge of actual values of sub-symbolic information. By concentrating on the core elements of ACT-R, our work makes analysis

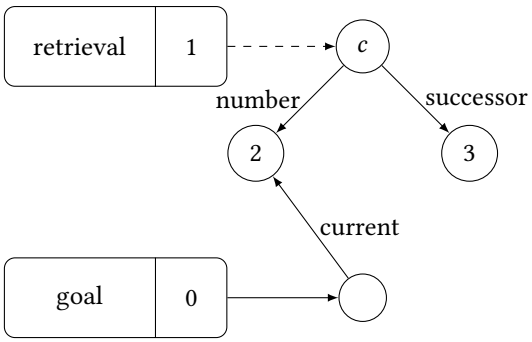

Fig. 4. Visual representation of state σ_2 from Example 4.2.

of cognitive models possible. Future work could deal with the expected behavior of sub-symbolic components. Sub-symbolic information often has noisy components, for instance the activation level of a chunk in declarative memory has a noisy component [34]. Additionally, sub-symbolic information can change by parameter adaptations. Those parameters are often adapted repeatedly in the modeling process to match data from experiments.

In [18], a refined semantics is presented. It has been designed to thoroughly describe the core of the reference implementation according to the reference manual [8]. It is shown that the refined semantics is sound w.r.t. the abstract semantics, i.e. every computation of the refined semantics is a computation of the abstract semantics. This means that the abstract semantics captures all computations possible in ACT-R implementations regardless of actual sub-symbolic components.

Since the semantics of ACT-R and its reference implementation is only given informally (c.f. [5, 8, 34]), it is not possible to formally verify our definition of the semantics. We closely followed the descriptions in the reference manual and designed the semantics with the reference implementation in mind. The soundness result for the refined semantics supports the close relation of our formalization and implementations. In [15], an older version of our ACT-R implementation that is closely related to the formalization has been evaluated to match the results of typical ACT-R implementations with different conflict resolution mechanisms.

Furthermore, in [2, 3] a formal semantics for ACT-R is presented that has emerged independently from our work (c.f. Section 7.1). Our very abstract semantics is an improved version of this semantics. Since the abstract semantics can be expressed in terms of the very abstract semantics, it is shown that it is a valid ACT-R formalization in terms of the semantics from [2, 3].

Our semantics can be easily extended by new modules by changing the signature of the allowed additional information and by adding an interpretation of the corresponding request in the interpretation function of actions (c.f. Definition 3.13). The very abstract and abstract semantics ignore different conflict resolution mechanisms and represent rule conflicts by non-determinism. By specifying a different selection function (c.f. Definition 3.17) than in the abstract semantics (c.f. Section 4.1), different conflict resolution mechanisms can be represented. Since the selection function of the abstract semantics simply defines matching of production rules, it can be easily extended to filter the matching rules according to some conflict resolution mechanism. This has been shown in [18], where the refined semantics is defined as another instance of the very abstract semantics that is sound w.r.t. the abstract semantics.

5 TRANSLATION OF ACT-R MODELS TO CONSTRAINT HANDLING RULES

In this section we show how to translate an ACT-R model to a Constraint Handling Rules (CHR) program. This is one of the main contributions of this paper. The translation is the first that matches the current operational semantics of ACT-R. The proof of the soundness and completeness of the translation w.r.t. the abstract operational semantics of ACT-R is a new contribution of this paper.

Therefore, we first give a quick introduction to CHR and its basic concepts we need for definitions and proofs in Section 5.1. Then we introduce a normal form of ACT-R rules that simplifies the translation process and proofs in Section 5.2. In Section 5.4 we show the translation of ACT-R states to CHR states and in Section 5.5 we define the translation scheme for rules. Finally, in Section 6 our translation of ACT-R models to CHR programs is proven sound and complete w.r.t. the abstract operational semantics of ACT-R.

5.1 Constraint Handling Rules (CHR)

Before we define the translation scheme of ACT-R models to CHR program, we first recapitulate syntax and semantics of CHR briefly. For an extensive introduction to CHR, its semantics, analysis and applications, we refer to [13]. We use the latest definition of the state transition system of CHR that is based on state equivalence [25]. This semantics allows for more elegant proofs and has been proven to be equivalent to the canonical so-called very abstract semantics of CHR. The definitions from those canonical sources are now reproduced. We assume the reader to be familiar with the common concepts of first-order predicate logic like predicate symbols, function symbols, predicates, functions, constants, variables, terms and formulas.

The syntax of CHR is defined over a set of variables $\mathcal{V}$, a set of function symbols (with arities) Φ and a set of predicate symbols with arities Π that is disjointly composed of *CHR constraint symbols* and *built-in constraint symbols*. The set of constraint symbols contains at least the symbols $=/2$, $true/0$ and $false/0$.

For a constraint symbol $c/n \in \Pi$ and terms $t_1, \ldots, t_n$ over $\mathcal{V}$ and Φ for $1 \leq i \leq n$, $c(t_1, \ldots, t_n)$ is called a *CHR constraint*, if c/n is a CHR constraint symbol or a *built-in constraint* if c is a built-in constraint respectively. CHR states are defined as follows.

Definition 5.1 (CHR state). A *CHR state* is a tuple $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$ where the *goal* $\mathbb{G}$ is a multi-set of constraints, the *built-in constraint store* $\mathbb{B}$ is a conjunction of built-in constraints and $\mathbb{V}$ is a set of *global variables*.

All variables occurring in a state that are not global are called *local* and the local variables that are only used for built-in constraints are called *strictly local variables*.

CHR states can be modified by rules that together form a CHR program.

Definition 5.2 (CHR program). A *CHR program* is a finite set of rules of the form

$$r \ @ \ H_k \setminus H_r \Leftrightarrow G \mid B_c, B_b$$

where r is an optional rule name, the heads H_k and H_r are multi-sets of CHR constraints, the guard G is a conjunction of built-in constraints and the body is a multi-set of CHR constraints B_c and a conjunction of built-in constraints B_b. 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*.

If $H_k = \emptyset$, the rule is called a *simplification rule* and we write

$$r \ @ \ H_r \Leftrightarrow G \mid B_c, B_b.$$

Conversely, if $H_r = \emptyset$, the rule is called a *propagation rule* and we write

$$r \ @ \ H_k \Rightarrow G \mid B_c, B_b.$$

Informally, a rule is applicable, if the head matches constraints from the store $\mathbb{G}$ and the guard holds, i.e. is a consequence of the built-in constraints $\mathbb{B}$. 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_c, B_b and G are added.

To define the operational semantics formally, we first have to define state equivalence over CHR states following the work in [25]. Therefore, we assume a constraint theory $\mathcal{CT}$ for the interpretation of the built-in constraints in Π.

Definition 5.3 (state equivalence of CHR states). Equivalence between CHR states is the smallest equivalence relation $\equiv$ over CHR states that satisfies the following conditions:

Equality as substitution

$$\langle \mathbb{G}; X{=}t \wedge \mathbb{B}; \mathbb{V} \rangle \equiv \langle \mathbb{G}[X/t]; X{=}t \wedge \mathbb{B}; \mathbb{V} \rangle.$$

Transformation of the constraint store If $\mathcal{CT} \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$ where $\bar{s}, \bar{s}'$ are the strictly local variables of $\mathbb{B}, \mathbb{B}'$, respectively, then:

$$\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \equiv \langle \mathbb{G}; \mathbb{B}'; \mathbb{V} \rangle.$$

Omission of non-occurring global variables

$$\langle \mathbb{G}; \mathbb{B}; \{X\} \cup \mathbb{V} \rangle \equiv \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle.$$

Equivalence of failed states

$$\langle \mathbb{G}; \mathit{false}; \mathbb{V} \rangle \equiv \langle \mathbb{G}'; \mathit{false}; \mathbb{V}' \rangle.$$

The operational semantics is now defined by the following transition scheme over equivalence classes over CHR states i.e. $[\sigma] ::= \{\sigma' \mid \sigma' \equiv \sigma\}$

Definition 5.4 (operational semantics of CHR). For a CHR program the state transition system over CHR states and the rule transition relation $\mapsto$ is defined as the following transition scheme:

$$\frac{r \ @ \ H_k \setminus H_r \Leftrightarrow G \mid B_c, B_b}{[\langle H_k \uplus H_r \uplus \mathbb{G}; G \wedge \mathbb{B}; \mathbb{V} \rangle] \mapsto^r [\langle H_k \uplus B_c \uplus \mathbb{G}; G \wedge B_b \wedge \mathbb{B}; \mathbb{V} \rangle]}$$

Thereby, we assume that r is a variant of a rule in the program such that its local variables are disjoint from the variables occurring in the representative of the pre-transition state.

We may just write $\mapsto$ instead of $\mapsto^r$ if the rule r is clear from the context.

5.2 Set Normal Form

To simplify the translation scheme, we assume the ACT-R production rules to be in *set normal form*. The idea is that each buffer test contains each slot exactly once. In the general ACT-R syntax it is possible to specify more than one slot-value pair for each slot or none at all. If we assume set-normal form, we can reduce the cases to consider in the translation and soundness and completeness proofs.

In the following, the set-normal form is defined formally and it is shown that every ACT-R production rule can be transformed to set-normal form with same semantics.

Definition 5.5 (set normal form). An ACT-R rule $r := L \Rightarrow R$ is in *set normal form*, if and only if for all buffer tests $=(b, t, P) \in L$ and $s \in \tau(t)$ there is exactly one $v \in \mathcal{V} \cup \mathcal{C}$ such that $(s, v) \in L$.

Theorem 5.6 (set normal form). *For all ACT-R models with rules Σ, there is a set of rules Σ' with the following properties: For all rules $r \in \Sigma$ with $r := L \Rightarrow R$ that are applicable in at least one state there is a rule $r' \in \Sigma'$ with $r' := L' \Rightarrow R$ in set-normal form such that for all ACT-R states σ, σ' it holds that $\sigma \rightarrowtail^r \sigma'$ if and only if $\sigma \rightarrowtail^{r'} \sigma'$ (in the abstract semantics).*

The set Σ' that contains the rules of Σ in set-normal form can be constructed as follows. All rules in Σ that are not applicable in any state do not appear in Σ'. For all remaining rules $r \in \Sigma$, we first transform r to a rule r' that is in set-normal form. For every buffer test $=(b, t, P) \in L$ there is a test $=(b, t, P'\theta) \in L'$ for a substitution θ. Thereby, P' has the following slot-value pairs:

- For all $s \in \tau(t)$ where there is exactly one $v \in \mathcal{V} \cup \mathcal{C}$ such that $(s, v) \in P$, $(s, v) \in P'$.
- For all $s \in \tau(t)$ where there is no $v \in \mathcal{V} \cup \mathcal{C}$ such that $(s, v) \in P$, there is a slot-value pair $(s, V) \in P'$ for a fresh variable $V \in \mathcal{V}$.
- For all $s \in \tau(t)$ where there is are $v_1, \ldots, v_n \in \mathcal{V} \cup \mathcal{C}$ such that $(s, v_i) \in P$ for $1 \leq i \leq n$ we produce a substitution $\theta_b := \{v_1/v, \ldots, v_n\}$ and add a slot-value pair $(s, v) \in P'$ for a $v \in \mathcal{V} \cup \mathcal{C}$. There are the following cases:
  (1) $v_1, \ldots, v_n \in \mathcal{V}$: Then $v \in \mathcal{V}$ is a fresh variable. Intuitively, we just introduce a new variable and replace all other variables by this new variable.
  (2) $v_1, \ldots, v_n \in \mathcal{C}$: Then $v = v_1$. Intuitively, since r is applicable in at least one state, $v_1 = v_2 = \cdots = v_n$. The slot-value pairs are redundant and therefore we only add one of them.
  (3) $v_1, \ldots, v_n \in \mathcal{V} \cup \mathcal{C}$: W.l.o.g., $v_1 \in \mathcal{C}$. Then $v = v_1$.

We define the substitution θ as the composition of all substitutions θ_b. The rule r' is obviously in set normal form.

The proof idea is that if $r \sqsubseteq \sigma$ all variables or constants that appear in the same slot test in the same buffer have the same values. If there is a buffer test $=(b, t, P)$ with two slot-value pairs $(s, v) \in P$ and (s, v'), then $v\theta = v'\theta$ for $\theta = \mathit{Bindings}(r, \sigma)$. Hence, we can replace all occurrences of v and v' by v and remove (s, v') from P. This is exactly what our construction of the set-normal form does.

5.3 Notational Aspects

We use the following symbols:

- Lists are denoted by enumerating their elements, e.g. $[a, b, c]$ for the list with elements a, b and c. [] denotes the empty list. We use $[H|T]$ for a list with head element H and tail list T. We also use the following notation for list comprehension: $[x : x \in M \wedge p(x)]$ is the list with all elements x from a set M that satisfy $p(x)$.
- The list concatenation is denoted with ++.
- For a function $f : A \to B$ with finite domain A, we define $[\![f]\!] := [(a, b) : a \in A \wedge b \in B \wedge f(a) = b]$ sorted by an order on A and B, i.e. the (sorted) enumerative list notation of the function f. It can be understood as the list representation of the relational representation of f as a set of tuples $f \subseteq A \times B$.

5.4 Translation of States

To translate an ACT-R state to CHR, we have to define translations for the individual components of such a state.

Definition 5.7 (translation of chunk stores). A (partial) chunk store Δ can be translated to a first order term as follows:

$$[\mathit{chunk}(c, t, [\![\mathit{val}]\!]) : c::(t, \mathit{val}) \in \Delta]$$

Thereby, $[\![f]\!]$ denotes the explicit relational notation of the function f as a sorted list of tuples and $[x : p(x)]$ is the list comprehension as defined in Section 5.3.

We denote the translation of a (partial) chunk store Δ with $\mathit{chr}(\Delta)$.

Each chunk in a chunk store is translated to a term *chunk*/3 that is member of a list. Note that we do not have defined the order of chunks in the list that represents the chunk store. We consider all permutations of such chunk store lists as equivalent, since for the proofs the actual choice of order will not make any difference, since we always can choose just the translation with the "correct" order. Where necessary, we comment on that issue in our proofs in Section 6. The slot-value pairs in the *chunk* terms are sorted as defined in $[\![\cdot]\!]$.

The cognitive state γ will be represented by *gamma*/3 constraints that map a buffer $b \in \mathbb{B}$ to a chunk identifier and a delay. Since additional information is already represented as logical predicates, we represent them as built-in constraints in the CHR store. We can now define the translation of an abstract state.

Definition 5.8 (translation of abstract states). An abstract ACT-R state $\sigma := \langle \Delta; \gamma; \upsilon \rangle$ can be translated to the following CHR state:

$$\langle \{delta(chr(\Delta))\} \\ \uplus \{gamma(b, id_\Delta(c), d) \mid b \in \mathbb{B}, c \in \Delta \wedge \gamma(b) = (c, d)\}; \upsilon; \emptyset \rangle.$$

We denote the translation of an ACT-R state σ by $chr(\sigma)$.

The chunk store is represented by a *delta* constraint that contains the translated chunk store as defined in Definition 5.7. Hence, a valid translation of an ACT-R state can only contain exactly one *delta* constraint.

For every buffer of the given architecture, a constraint *gamma* with buffer name, chunk id and delay is added to the state. Since γ is a total function, every buffer has exactly one *gamma* constraint. Additionally, the chunk identifier in the *gamma* constraint must appear in exactly one *chunk* constraint because the co-domain of gamma refers to Δ and the chunk identifiers are unique.

Additional information is used directly as built-in constraints.

5.5 Translation of Rules

In our translation scheme, ACT-R rules are translated to corresponding CHR rules.

5.5.1 Auxiliary Functions for Variable Names. To manage relations between newly introduced variables, we define some auxiliary functions. The functions all produce variable names from the set of variables $\mathcal{V}$ for a set of arguments that are from the set of constants C (or a subset of it) and are applied during the translation, i.e. they do not appear in the generated CHR code.

Definition 5.9 (variable functions). Let $\mathcal{V}, C$ be the set of variables and constants of an ACT-R architecture respectively, $\mathbb{B} \subset C$ the set of buffers of this architecture and $\mathcal{V}_i \subset \mathcal{V}$ for $i = 1, \ldots, 6$ are disjoint subsets of the set of variables. Then the following auxiliary functions are defined:

Chunk variable function $CVar : \mathbb{B} \to \mathcal{V}_1, b \mapsto C_b$ that returns a fresh, unique variable C_b^1 for each buffer b. It identifies the chunk of a particular buffer in the translation.

Delay variable function $DVar : \mathbb{B} \to \mathcal{V}_2, b \mapsto D_b$ that returns a fresh, unique variable D_b for each buffer b. It identifies the delay of a particular buffer in the translation.

Result variable functions $ResStore, ResId, ResDelay : \mathbb{B} \to \mathcal{V}_i$ for $i = 3, 4, 5$ are defined as $b \mapsto X_b^i$ and return a fresh, unique variable X_b^i for each buffer b. They are needed to memorize the results of an action.

Merge variable function $MergeId : \mathbb{B} \times \mathcal{V}_6, (b, s) \mapsto V_{b,s}$ that returns a fresh, unique variable X_b^6 for each buffer b. It is needed to memorize the new chunk identifier after merging the chunk in b with the existing chunk store.

Cognitive state variable function $CogState : 2^{\mathbb{B}} \rightarrow 2^{\mathbb{B}\times\mathcal{V}_1}, CogState(B) \mapsto [(b, CVar(b)) : b \in B]$ (where the list is sorted by the b). The function returns a set of buffer-variable pairs for a set of buffers that connects each buffer with the chunk variable of its chunk. When equating $CVar(b)$ with $id_\Delta(\gamma(b))$ for all buffers $b \in \mathbb{B}$, $CogState(\mathbb{B})$ can be considered as a pattern for the cognitive state.

The functions start with capital letters to emphasize that they represent CHR variables (or tuples of CHR variables in the case of *CogState*) in the translated rules.

5.5.2 Built-in Constraints for Actions. We define some built-in constraints that are needed for the translation. The idea is that they calculate the results of actions, chunk merging and chunk mapping as defined in the operational semantics of ACT-R. For actual instantiations of the abstract semantics (i.e. with a defined set of actions and chunk merging and mapping mechanisms), it has to be shown that their CHR implementations obey the properties that we define in the following.

Definition 5.10 (action built-ins). Let $\alpha := a(b, t, P)$ be an action and $\sigma := \langle\Delta; \gamma; v\rangle$ an ACT-R state. Let $D := chr(\Delta)$ be the CHR representation of the chunk store in σ and $G := [\![\gamma]\!]$ be the enumerative list representation of the cognitive state γ. Then the *action* constraint is defined as follows:

$$action(\alpha, D, G, D_{res}, C_{res}, E_{res}) \wedge v \leftrightarrow D_{res} = (\Delta^*, \gamma^*, v^*) \in I(\alpha, \sigma) \wedge chr(\Delta^*) \wedge \gamma^*(b) := (c_b^*, d_b^*) \wedge C_{res} = id_{\Delta^*}(c_b^*) \wedge E_{res} = d_b^* \wedge v^*$$

This means that the *action* constraint is equivalent to the situation where the triple $(\Delta^*, \gamma^*, v^*) \in I(\alpha, \sigma)$ is a result of the interpretation of action α in the ACT-R state σ with $\gamma^*(b) := (c_b^*, d_b^*)$ and the result variables $D_{res}, C_{res}, E_{res}$ are syntactically equivalent to the translations of their ACT-R counterparts.

Intuitively, the *action* built-in constraint represents a function that gets the action α, CHR representations of the constraint store Δ and the cognitive state γ as input and returns by the help of the additional information v the CHR representation of $I(\alpha, \sigma)$. The constraint theory of the *action* constraint is well-defined, since in Definition 3.13, γ^* has the domain $\{b\}$ and co-domain $\Delta^* \times \mathbb{R}_0^+$, hence $\gamma^*(b)$ is defined. Since $c_b^* \in \Delta^*$, $id_{\Delta^*}(c_b^*)$ is defined.

5.5.3 Built-in Constraints for Chunk Merging. In the abstract semantics, chunk stores are merged by the operator $\circ$. In the following, built-in constraints are defined that implement $\circ$ and the corresponding mapping function *map*.

Definition 5.11 (merge built-in). For a set of chunk stores $\{\Delta_1, \ldots, \Delta_n\}$ with $D_i := chr(\Delta_i)$ for $i = 1, \ldots, n$, the built-in $merge/2$ is defined as follows:

$$merge([D_1, D_2, \ldots D_n], D) \leftrightarrow D = chr(\Delta_1 \circ \Delta_2 \circ \ldots \circ \Delta_n).$$

Definition 5.12 (map built-in). For two CHR representations of chunk stores $D := chr(\Delta)$ and $D' := chr(\Delta')$, the built-in constraint $map/4$ is defined as:

$$map(D, D', C, C') \leftrightarrow C' = id_{\Delta\circ\Delta'}(map_{\Delta,\Delta'}(id_\Delta^{-1}(C))) \text{ if } chunk(C, T, P) \text{ in } D \text{ or}$$
$$C' = id_{\Delta\circ\Delta'}(map_{\Delta,\Delta'}(id_{\Delta'}^{-1}(C))) \text{ otherwise.}$$

The main difference of this definition from the definition of the function *map* from the abstract semantics is that the built-in constraint operates on chunk identifiers whereas the function operates directly on chunks. Note that in the case that the chunk with identifier C appears in neither in D nor in D', C' is bound to `nil` by definition of *id* (see Definition 3.2).

5.5.4 List Operations. We use the built-in constraint $in/2$ to denote that a term is member of a list.

Definition 5.13 (member of a list). For a term c and a list l, the constraint c *in* l holds, iff there is a term c' that is member of l and $c = c'$.

Note that variables in c are bound to the values in l by this definition.

5.5.5 Translation Scheme for Rules. We can now define the translation scheme for rules.

Definition 5.14 (translation of rules). An ACT-R rule in set-normal form $r := L \Rightarrow R$ can be translated to a CHR rule of the following form:

$$
\begin{aligned}
& r \mathbin{@} \{delta(D)\} \uplus \{gamma(b, CVar(b), DVar(b)) \mid b \in \mathbb{B}\} \\
& \quad \Leftrightarrow \\
& \qquad \bigwedge_{=(b,t,P)\in L} (chunk(CVar(b), t, P) \ in \ D \wedge DVar(b){=}0) \mid \\
& \qquad \{delta(D^*)\} \\
& \quad \uplus \{gamma(b, MergeId(b), ResDelay(b)) \mid a(b, t, P) \in R\} \\
& \quad \uplus \{gamma(b, CVar(b), DVar(b)) \mid a(b, t, P) \notin R\}, \\
& \qquad \bigwedge_{\alpha=a(b,t,P)\in R} action(\alpha, D, CogState(\mathbb{B}), ResStore(b), ResId(b), ResDelay(b)) \\
& \quad \wedge \ merge([ResStore(b) : a(b, t, P) \in R], D')\} \\
& \quad \wedge \ merge([D, D'], D^*) \uplus \\
& \quad \wedge \bigwedge_{a(b,t,P)\in R} map(D, D', ResId(b), MergeId(b)).
\end{aligned}
$$

Note that ACT-R constants and variables from $\mathcal{C}$ and $\mathcal{V}$ are implicitly translated to corresponding CHR variables.

We denote the translation of a rule r by $chr(r)$ and the translation of an ACT-R model Σ that is a set of ACT-R rules by $chr(\Sigma)$. Thereby, $chr(\Sigma) := \{chr(r) \mid r \in \Sigma\}$.

The intuition behind the translation can be described as follows:

The CHR rule tests the state for a $delta/1$ constraint representing the chunk store and $gamma/3$ constraints that come from the buffer tests of the rule. In the $gamma/3$ constraints a variable for the chunk identifier is introduced. In the guard, the built-in constraint $in/2$ checks, if the chunk store represented as a list contains a term $chunk/3$ with the same type and slot-value pairs as specified in the buffer tests. The connection to the buffer is realized by the same variable for the chunk identifier (through the variable function *CVar*). Since the rule is in set-normal form, the buffer tests are already completed (i.e. all slots are tested) and represented as a sorted list of slot-value pairs as in the state.

In the body of the rule, the built-in constraint calculates the result of each action from the right-hand side of the ACT-R rule. The resulting chunk stores are merged to one store D' by the built-in constraint *merge*. Note that the order of merging is not specified by the translation scheme (as it is not specified by the ACT-R abstract semantics).

The built-in constraint $map/4$ implements the *map* function of the ACT-R semantics and gives access to the possibly modified chunk identifier of all elements in D'. By definition of *map*, only chunk identifiers in D' are modified by merging. The resulting chunk identifiers are bound to a variable specified by the variable function $MergeId/1$.

The resulting chunk store *delta* and *gamma* constraints for all buffers are added. If the buffers have been modified, the *gamma* constraints points to the resulting chunk of the action, if not it shows to the chunk that has been in the buffer before.

Example 5.15 (translation of rules). In this example, the rule $inc : L \Rightarrow R$ from Example 2.1 is translated to CHR. Let $\alpha_{\mathrm{g}} \in R$ and $\alpha_{\mathrm{r}} \in R$ be the actions for the **g**oal and the **r**etrieval buffer, respectively.

The following rule is the result of the translation $chr(inc)$. For readability, multi-set brackets are omitted and built-in constraints and CHR constraints are mixed and separated by comma in the body.

$$
\begin{aligned}
&inc \ @ \ delta(D),\\
&\quad gamma(goal, C_{\mathrm{g}}, E_{\mathrm{g}}), gamma(retrieval, C_{\mathrm{r}}, E_{\mathrm{r}})\\
&\Leftrightarrow\\
&\quad chunk(C_{\mathrm{g}}, g, [(current, X)]) \ in \ D \wedge E_{\mathrm{g}}{=}0 \wedge\\
&\quad chunk(C_{\mathrm{r}}, succ, [(number, X), (successor, Y)]) \ in \ D \wedge E_{\mathrm{r}}{=}0 \mid\\
&\quad action(\alpha_{\mathrm{g}}, D, [(goal, C_{\mathrm{g}}), (retrieval, C_{\mathrm{r}})], RD_{\mathrm{g}}, RC_{\mathrm{g}}, RE_{\mathrm{g}}),\\
&\quad action(\alpha_{\mathrm{r}}, D, [(goal, C_{\mathrm{g}}), (retrieval, C_{\mathrm{r}})], RD_{\mathrm{r}}, RC_{\mathrm{r}}, RE_{\mathrm{r}}),\\
&\quad merge([RD_{\mathrm{g}}, RD_{\mathrm{r}}], D'),\\
&\quad merge([D, D'], D^*),\\
&\quad map(D, D', RC_{\mathrm{g}}, M_{\mathrm{g}}),\\
&\quad map(D, D', RC_{\mathrm{r}}, M_{\mathrm{r}}),\\
&\quad delta(D^*),\\
&\quad gamma(goal, M_{\mathrm{g}}, RE_{\mathrm{g}}),\\
&\quad gamma(retrieval, M_{\mathrm{r}}, RE_{\mathrm{r}}).
\end{aligned}
$$

5.6 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:

$$no \ @ \ gamma(B, C, D) \Leftrightarrow D > 0 \mid gamma(B, C, 0)$$

This transition is possible for all requests that are pending (i.e. that have a delay $D > 0$). Hence, the system chooses one request non-deterministically.

6 SOUNDNESS AND COMPLETENESS OF THE TRANSLATION

In this section, we show that our translation is sound and complete w.r.t. the abstract semantics of ACT-R. This means that every transition that is possible in the abstract ACT-R semantics is also possible in CHR and vice versa.

The first step is to show that the results of the built-in constraints in the body of a translated ACT-R rule are equivalent to the interpretation of the right-hand side of the ACT-R rule. Therefore, we use that by definition the built-in constraints *action*, *merge* and *map* are equivalent to their

ACT-R counterparts. Additionally, we have to show that the combination of the results of individual actions leads to the same result as the built-in constraints. We use induction to show that in the following lemma.

Lemma 6.1 (equivalence of effects). *For an ACT-R rule $r := L \Rightarrow R$ in set-normal form, a state $\sigma := \langle \Delta, \gamma, v, t \rangle$ and $D = chr(\Delta)$. Let $I(r, \sigma) := (\Delta^*, \gamma^*, v^*)$ with $\bigwedge_{b \in dom(\gamma^*)} \gamma^*(b) = (c_b^*, d_b^*)$. Then the following two propositions are equivalent:*

(1)

$$\bigwedge_{\alpha_b \in R} action(\alpha_b, D, G, ResStore(b), ResId(b), ResDelay(b)) \wedge v \wedge merge([ResStore(b) : a(b,t,P) \in R], D^*) \wedge merge([D, D^*], D') \wedge \bigwedge_{\alpha_b \in R} map(D, D^*, ResId(b), MergeId(b))$$

(2)

$$D^* = chr(\Delta^*) \wedge D' = chr(\Delta \circ \Delta^*) \wedge \bigwedge_{b \in dom(\gamma^*)} (MergeId(b) = id_{\Delta \circ \Delta^*}(map_{\Delta, \Delta^*}(c_b^*)) \wedge ResDelay(b) = d_b^*) \wedge v^* \wedge v$$

Proof. We use induction over the number of actions in R.

base case: $|R| = 1$ Let $r := L \Rightarrow R$ be an ACT-R rule in set-normal form with one action $\alpha \in R$ and $\sigma := \langle \Delta, \gamma, v, t \rangle$. Let $D = chr(\Delta)$ and $G = [\![\gamma]\!]$. Let $I(r, \sigma) := (\Delta^*, \gamma^*, v^*)$ with $\bigwedge_{b \in dom(\gamma^*)} \gamma^*(b) = (c_b, d_b)$ and $I(\alpha, \sigma) := (\Delta_\alpha, \gamma_\alpha, v_\alpha)$ with $(c_\alpha, d_\alpha) = \gamma_\alpha(b)$.

We start with

$$action(\alpha, D, G, D_{res}, C_{res}, E_{res}) \wedge v \wedge merge([D_{res}], D^*) \wedge merge([D, D^*], D') \wedge map(D, D^*, C_{res}, C'_{res}).$$

First of all, we reduce the *action* constraint by use of Definition 5.10:

$$action(\alpha, D, G, D_{res}, C_{res}, E_{res}) \wedge v \leftrightarrow D_{res} = chr(\Delta_\alpha) \wedge C_{res} = id_{\Delta_\alpha}(c_\alpha) \wedge E_{res} = d_\alpha \wedge v \wedge v_\alpha.$$

By Definition 5.11, we can reduce the *merge* constraints to

$$D^* = chr(chr^{-1}(D_{res})) \wedge D' = chr(chr^{-1}(D) \circ chr^{-1}(D^*))$$

which is by definition of $D := chr(\Delta)$ from the assumptions and $D_{res} = chr(\Delta_\alpha)$ from the last step equivalent to

$$D^* = chr(\Delta_\alpha) \wedge D' = chr(\Delta \circ \Delta_\alpha).$$

Since $C_{res} = id_{\Delta_\alpha}(c_\alpha)$ and therefore a *chunk* term with identifier C_{res} appears in D^* (and not in D), we can now reduce the *map* built-in by Definition 5.12 and get together with the definitions of D and D^* to

$$map(D, D^*, C_{res}, C'_{res}) \leftrightarrow C'_{res} = id_{\Delta \circ \Delta_\alpha}(map_{\Delta, \Delta_\alpha}(id_{\Delta_\alpha}^{-1}(C_{res}))).$$

Since we have that $C_{res} = id_{\Delta_\alpha}(c_\alpha)$, this is equivalent to

$$C'_{res} = id_{\Delta \circ \Delta_\alpha}(map_{\Delta,\Delta_\alpha}(c_\alpha)).$$

By Definition 3.16, we have that for one action $\Delta_\alpha = \Delta^*$ and $c_\alpha = c^*$. Hence, this is equivalent to

$$C'_{res} = id_{\Delta \circ \Delta^*}(map_{\Delta,\Delta^*}(c^*)).$$

All in all, this proves the proposition for $|R| = 1$.

induction step: $|R| \to |R| + 1$ Let $\sigma := \langle \Delta; \gamma; \upsilon \rangle$ be an ACT-R state and $D := chr(\Delta)$ the CHR representation of the chunk store and $G := [\![\gamma]\!]$ the enumerative list representation of the cognitive state.

Let $r' := L \Rightarrow R'$ with $R' := R \cup \alpha$ be a rule that has been constructed from a rule $r := L \Rightarrow R$. Let $I(r, \sigma) := (\Delta^*, \gamma^*, \upsilon^*)$ with $\bigwedge_{b \in dom(\gamma^*)} \gamma^*(b) = (c_b^*, d_b^*)$ be the interpretation of the smaller rule r with $|R|$ actions.

Let $I(r', \sigma) := (\Delta^{**}, \gamma^{**}, \upsilon^{**})$ with $\bigwedge_{b \in dom(\gamma^{**})} \gamma^{**}(b) = (c_b^{**}, d_b^{**})$ be the interpretation of the rule r' that has $|R| + 1$ actions..

We begin with

$$\bigwedge_{\alpha_b \in R'} action(\alpha_b, D, G, D_b, C_b, E_b) \wedge \upsilon \wedge merge([D_b : a(b,t,P) \in R'], D^{**}) \wedge merge([D, D^{**}], D'') \wedge \bigwedge_{\alpha_b \in R'} map(D, D^{**}, C_b, C''_b).$$

We need to apply the induction hypothesis. Therefore, we split the conjunctions and lists and get the equivalent formula

$$\bigwedge_{\alpha_b \in R} action(\alpha_b, D, G, D_b, C_b, E_b) \wedge action(\alpha_b, D, G, D_\alpha, C_\alpha, E_\alpha) \wedge \upsilon \wedge merge([D_b : a(b,t,P) \in R] + +[D_\alpha], D^{**}) \wedge merge([D, D^{**}], D'') \wedge \bigwedge_{\alpha_b \in R} map(D, D^{**}, C_b, C''_b) \wedge map(D, D^{**}, C_\alpha, C'_\alpha).$$

By definition of *merge* (c.f. Definition 5.10) and associativity of $\circ$ and neutral element $[] = chr(\emptyset)$, we can split the merging as follows: We first merge the actions in R to D^* and then merge D^* with the result chunk of action α to D^{**}:

$$\bigwedge_{\alpha_b \in R} action(\alpha_b, D, G, D_b, C_b, E_b) \wedge action(\alpha, C_\alpha, T_\alpha, P_\alpha, D_\alpha) \wedge \upsilon \wedge merge([D_b : a(b,t,P) \in R], D^*) \wedge merge([D^*, D_\alpha], D^{**}) \wedge merge([D, D^{**}], D'') \wedge \bigwedge_{\alpha_b \in R} map(D, D^{**}, C_b, C''_b) \wedge map(D, D^{**}, C_\alpha, C'_\alpha).$$

We can now introduce an intermediate result chunk store that merges the original store D with the results from r, i.e. D^*, to a chunk store D'. We introduce some auxiliary variables

C'_b that map the chunk identifiers of the intermediate chunk store D^* to the resulting chunk store D^*:

$$\bigwedge_{\alpha_b \in R} action(\alpha_b, D, G, D_b, C_b, E_b) \wedge action(\alpha, C_\alpha, T_\alpha, P_\alpha, D_\alpha) \wedge v \wedge$$
$$merge([D_b : a(b,t,P) \in R], D^*) \wedge$$
$$merge([D, D^*], D') \wedge$$
$$\bigwedge_{\alpha_b \in R} map(D, D^*, C_b, C'_b) \wedge$$
$$merge([D^*, D_\alpha], D^{**}) \wedge$$
$$merge([D, D^{**}], D'') \wedge$$
$$\bigwedge_{\alpha_b \in R} map(D, D^{**}, C_b, C''_b) \wedge map(D, D^{**}, C_\alpha, C'_\alpha).$$

We can now apply the induction hypothesis:

$$\begin{aligned}(*) := \\ & v \wedge v^* \wedge \\ & D^* = chr(\Delta^*) \wedge D' = chr(\Delta \circ \Delta^*) \wedge \\ & \bigwedge_{b \in dom(\gamma^*)} (C'_b = id_{\Delta \circ \Delta^*}(map_{\Delta, \Delta^*}(c^*_b)) \wedge E_b = d^*_b) \wedge \\ & merge([D^*, [chunk(C_\alpha, T_\alpha, P_\alpha)]], D^{**}) \wedge \\ & merge([D, D^{**}], D'') \wedge \\ & \bigwedge_{\alpha_b \in R} map(D, D^{**}, C_b, C''_b) \wedge map(D, D^{**}, C_\alpha, C'_\alpha).\end{aligned}$$

Thereby, it holds by definition of $I(r, \sigma)$ that $dom(\gamma^*) = \{b \mid \alpha_b \in R\}$.

If we apply Definition 5.10 to the remaining *action* constraint of action α, we get

$$action(\alpha, D, G, D_\alpha, C_\alpha, E_\alpha) \wedge v \leftrightarrow D_\alpha = chr(\Delta_\alpha) \wedge C_\alpha = id_{\Delta_\alpha}(c_\alpha) \wedge E_\alpha = d_\alpha \wedge v \wedge v^*.$$

Hence, we have that

$$merge([D^*, D_\alpha, D^{**}) \leftrightarrow D^{**} = chr(\Delta^* \circ \Delta_\alpha).$$

Thus, D^{**} is the CHR version of the merging of the results from the actions in R merged with the results from α. By Definition 3.16 of the interpretation of rules, we have that

$$I(r', \sigma) = (\Delta^* \circ \Delta_\alpha, \gamma' \cup \gamma_\alpha, v^* \wedge v_\alpha).$$

This is equivalent to

$$I(r', \sigma) = (\Delta^{**}, \gamma^{**}, v^{**})$$

by definition of r, r' and the interpretation of rules (Definition 3.16).

The last *merge* constraint can be reduced to:

$$merge([D, D^{**}], D'') \leftrightarrow D'' = chr(\Delta \circ \Delta^{**})$$

The *map* constraints can be reconnected which yields

$$\bigwedge_{\alpha_b \in R} map(D, D^{**}, C_b, C_b'') \wedge map(D, D^{**}, C_\alpha, C_\alpha') \leftrightarrow \bigwedge_{\alpha_b \in R'} map(D, D^{**}, C_b, C_b'')$$

By Definition 3.17, for all defined buffers $\gamma^{**}(b) = map_{\Delta,\Delta^{**}}(c_b)$ and we have assumed that $\gamma^{**}(b) = c_b^{**}$. By Definition 5.12, this yields

$$C_b'' = id_{\Delta \circ \Delta^{**}}(map_{\Delta,\Delta^{**}}(id_{\Delta^{**}}^{-1}(C_b))).$$

$id_{\Delta^{**}}^{-1}(C_b)$ exists, since the co-domain of γ^{**} is $\Delta^{**} \times \mathbb{R}_0^+$ by Definition 3.4 and the C_b are identifiers for the chunks in γ^{**}.

If we apply all this to the conjunction in $(*)$, we get

$$D^{**} = chr(\Delta^{**}) \wedge D'' = chr(\Delta \circ \Delta^{**}) \wedge$$
$$\bigwedge_{b \in dom(\gamma^{**})} (C_b'' = id_{\Delta \circ \Delta^{**}}(map_{\Delta,\Delta^{**}}(c_b^{**})) \wedge E_b = d_b^{**}) \wedge v^{**} \wedge v$$

□

The next lemma proves that rule application transitions are sound.

Lemma 6.2 (soundness of rule applications). *For all ACT-R rules r and ACT-R states $\sigma, \sigma' \in \mathcal{S}_{abs}$: if $\sigma \rightarrowtail^r \sigma'$ then $chr(\sigma) \mapsto^r chr(\sigma')$.*

Proof. Let $r := L \Longrightarrow R$ and $\sigma := \langle \Delta; \gamma; v; 0 \rangle$. Since $r \sqsubseteq \sigma$, we know that for every buffer test $\beta := =(b, t, P)$ there is a substitution θ such that $\gamma(b) = (c::(t, val), 0)$ and for all $(s, v) \in P$: $id_\Delta(val(s)) = v\theta$.

Let $\rho := chr(\sigma)$ the CHR translation of the ACT-R state σ. By Definition 5.8, we have that

$$\rho \equiv \langle \{delta([chunk(i_c, t, [\![val]\!]) \ : \ c \in \Delta \wedge c = i_c::(t, val)])\}$$
$$\uplus \{gamma(b, id_\Delta(c), d) \mid b \in \mathbb{B} \wedge c \in \Delta \wedge \gamma(b) = (c, d)\}; v; \emptyset \rangle$$

In the next step we split the state to only concentrate on the parts we are interested in for the rule application, i.e. *gamma* constraints for buffers that occur in a test. We do the same by splitting the representation of the constraint store to the chunks that occur in buffers and all other chunks. This is possible, since $chr(\Delta)$ does not define an order on the *chunk* terms in the list in the *delta* constraint.

$$\rho \equiv \langle \{delta([chunk(i_c, t, [\![val]\!]) \ : \ =(b, t', P) \in L \wedge \gamma(b) = (c, 0) \wedge c = i_c::(t, val)]{+}{+}\mathbb{D})\}$$
$$\uplus \{gamma(b, id_\Delta(c), d) \mid =(b, t, P) \in L \wedge \gamma(b) = (c, d)\} \uplus \mathbb{G}; v; \emptyset \rangle$$

Note that by now we only have rewritten the CHR state without requiring that the tests we refer to match.

Due to the fact that $r \sqsubseteq \sigma$, we can apply this knowledge to the translated state:

$$\rho \equiv \langle \{delta([chunk(i_c, t, [\![val]\!]) \ : \ =(b, t, P) \in L \wedge \gamma(b) = (c, 0) \wedge c = i_c::(t, val)$$
$$\wedge \ \text{for all } (s, v) \in P : id_\Delta(val(s)) = v\theta]{+}{+}\mathbb{D})\}$$
$$\uplus \{gamma(b, id_\Delta(c), 0) \mid =(b, t, P) \in L \wedge \gamma(b) = (c, 0)\} \uplus \mathbb{G}; v; \emptyset \rangle$$

Since r is in set-normal form, we can assume that there is only one test for each buffer in L. Hence, we find a *gamma* constraint in ρ for each such test without violating Definition 5.8, that only produces one *gamma* constraint for each buffer.

Due to set-normal form of r and hence totality of P w.r.t. the domain of val, it is clear that $P\theta = [\![val]\!]$. Hence, we can reduce the state to:

$$\rho \equiv \langle \{delta([chunk(id_\Delta(c), t, P\theta) \ : \ =(b,t,P) \in L \wedge \gamma(b) = (c,0)]{+}{+}\mathbb{D})\} \\ \uplus \{gamma(b, id_\Delta(c), 0) \mid =(b,t,P) \in L \wedge \gamma(b) = (c,0)\} \uplus \mathbb{G}; v; \emptyset\rangle$$

We introduce a substitution with fresh variables to replace the chunk identifiers in the state, i.e. $\theta' := \{CVar(b)/id_\Delta(c) \mid \gamma(b) = c\}$.

$$\rho \equiv \langle \{delta([chunk(CVar(b)\theta', t, P\theta) \ : \ =(b,t,P) \in L]{+}{+}\mathbb{D})\} \\ \uplus \{gamma(b, CVar(b)\theta', 0) \mid =(b,t,P) \in L\} \uplus \mathbb{G}; v; \emptyset\rangle$$

Let Θ be the conjunction of syntactic equality constraints that can be derived from the substitution $\theta \cup \theta'$, i.e. each substitution $x/t \in \theta \cup \theta'$ appears in Θ as $x = t$. By Definition 5.3, we can move the substitution to the built-in store:

$$\rho \equiv \langle \{delta([chunk(CVar(b), t, P) \ : \ =(b,t,P) \in L]{+}{+}\mathbb{D})\} \\ \uplus \{gamma(b, CVar(b), 0) \mid =(b,t,P) \in L\} \uplus \mathbb{G}; \Theta \uplus v; \emptyset\rangle$$

We introduce a fresh variable D and add $D{=}[chunk(CVar(b), t, P) \ : \ =(b,t,P) \in L]{+}{+}\mathbb{D}$ to the built-in store, which leads to the equivalent state:

$$\rho \equiv \langle \{delta(D)\} \uplus \{gamma(b, CVar(b), 0) \mid =(b,t,P) \in L\} \uplus \mathbb{G}; \\ \Theta \uplus D{=}[chunk(CVar(b), t, P) \ : \ =(b,t,P) \in L]{+}{+}\mathbb{D} \uplus v; \emptyset\rangle$$

By Definition 5.13, we have that $D{=}d \leftrightarrow \{chunk(CVar(b), t, P) \ in \ D \mid =(b,t,P)\}$. We can replace the corresponding built-ins by Definition 5.3:

$$\rho \equiv \langle \{delta(D)\} \uplus \{gamma(b, CVar(b), 0) \mid =(b,t,P) \in L\} \uplus \mathbb{G}; \\ \Theta \uplus \{chunk(CVar(b), t, P) \ in \ D \mid =(b,t,P)\} \uplus v; \emptyset\rangle$$

Let $chr(r) := H \Leftrightarrow G \mid B_c, B_b$. By Definition 5.14, we have that

$$\rho \equiv \langle H \uplus \mathbb{G}; \Theta \uplus G \uplus v; \emptyset\rangle$$

By Definition 5.4, $chr(r)$ is applicable in $\rho \equiv chr(\sigma)$ and $\rho \mapsto^r \rho'$ with

$$\rho' \equiv \langle B_c \uplus \mathbb{G}; \Theta \wedge G \wedge B_b \wedge v; \emptyset\rangle$$

Due to the Definition 5.14 of $chr(r)$, we have that

$$\begin{aligned} B_b := & \bigwedge_{\alpha = a(b,t,P) \in R} (action(\alpha, ResId(b), restype(b), resslots(b), ResDelay(b))) \ \wedge \\ & merge([[chunk(ResId(b), restype(b), resslots(b))] : a(b,t,P) \in R], D') \wedge \\ & merge([D, D'], D^*) \wedge \\ & \bigwedge_{a(b,t,P) \in R} (map(D, D', ResId(b), MergeId(b))) \\ B_c := & \{delta(D^*)\} \uplus \\ & \{gamma(b, MergeId(b), ResDelay(b)) \mid a(b,t,P) \in R\} \end{aligned}$$

Since r is in set-normal form, all buffers appearing in L also appear in R. Hence, all $gamma$ constraints removed by $chr(r)$ are added in B_c. There is exactly one $delta$ constraint in ρ'. It remains to show that the $delta$ and $gamma$ constraints are the ones describing the state σ'.

By Lemma 6.1, we have that for all $I(r,\sigma) \ni (\Delta^*, \gamma^*, \upsilon^*)$ with $\gamma^*(b) = (c_b^*, d_b^*)$ for all $b \in dom(\gamma^*)$

$$B_b \leftrightarrow D^* = chr(\Delta^*) \wedge D' = chr(\Delta \circ \Delta^*) \wedge \bigwedge_{b \in dom(\gamma^*)} (MergeId(b) = id_{\Delta \circ \Delta^*}(map_{\Delta,\Delta^*}(c_b^*)) \wedge ResDelay(b) = d_b^*) \wedge \upsilon^* \wedge \upsilon$$

Finally, we get by Definition 5.8 that $\rho' \equiv chr(\sigma')$. Hence, the translation of rule applications is sound w.r.t. the abstract operational semantics of ACT-R. □

We have now shown that our translation is sound regarding rule applications, i.e. a rule that can be applied in the abstract semantics of ACT-R can also be applied in the translated CHR program and leads to the same result. The state transition system of ACT-R has a second type of transitions: the *no-rule transitions*, that can be applied if there is a buffer with a delay > 0, i.e. an invisible chunk. In that case, the no-rule transition allows us to make this chunk visible to the production system by setting the delay to zero. In the following it is shown that our translation is sound and complete regarding the no-rule transition.

Lemma 6.3 (soundness and completeness of the no-rule transition). *For an ACT-R model M, ACT-R states σ and σ' and their CHR counterparts $chr(M)$, $chr(\sigma)$ and $chr(\sigma')$ the following propositions are equivalent:*

(1) $\sigma \rightarrowtail_{\mathbf{no}} \sigma'$ in the model M
(2) $chr(\sigma) \mapsto^{no} chr(\sigma')$

Proof. We start with proposition 1. Let $\sigma := \langle \Delta; \gamma; \upsilon \rangle$ and $\sigma' := \langle \Delta; \gamma'; \upsilon \rangle$. In the abstract semantics, the no-rule transition is defined as

$$\frac{\gamma(b^*) = (c^*, d^*) \wedge d^* > 0}{\langle \Delta; \gamma; \upsilon \rangle \rightarrowtail_{\mathbf{no}} \langle \Delta; \gamma'; \upsilon \rangle}$$

where $\gamma'(b) = \begin{cases} (c^*, 0) & \text{if } b = b^*, \\ \gamma(b) & \text{otherwise.} \end{cases}$

The *no-rule* transition in CHR is represented by the following CHR rule in $chr(M)$:

$$no \ @ \ gamma(B, C, D) \Leftrightarrow D > 0 \mid gamma(B, C, 0)$$

Since the *no-rule* transition is applicable in σ for some arbitrary but fixed b^*, it holds that

$$\gamma(b^*) = (c^*, d^*) \wedge d^* > 0.$$

Therefore, $chr(\sigma)$ must have the following form by Definition 5.8:

$$chr(\sigma) \equiv \langle gamma(B, C, D) \uplus \mathbb{G}; B{=}b^* \wedge C{=}id_\Delta(c^*) \wedge D{=}d^* \wedge \mathbb{C}; \emptyset \rangle.$$

Since $d^* > 0$, this is equivalent to

$$chr(\sigma) \equiv \langle gamma(B, C, D) \uplus \mathbb{G}; D > 0 \wedge B{=}b^* \wedge C{=}id_\Delta(c^*) \wedge D{=}d^* \wedge \mathbb{C}; \emptyset \rangle$$

and therefore the rule *no* can be applied to $chr(\sigma)$ by Definition 5.4. This leads to the state ρ' with

$$\rho' \equiv \langle gamma(b^*, c^*, 0) \uplus \mathbb{G}; \mathbb{C}; \emptyset \rangle.$$

By Definition 5.8, we have that $\rho' \equiv chr(\sigma)$.

The other direction is analogous. □

Lemma 6.4 (completeness of rule applications). *Let ρ, ρ' be CHR states that have been translated from ACT-R states, i.e. there are ACT-R states σ, σ' such that $\rho := chr(\sigma)$ and $\rho' := chr(\sigma')$. Let r be a CHR rule translated from an ACT-R rule s. If $\rho \mapsto_r \rho'$ then $\sigma \rightarrowtail_s \sigma'$.*

Proof. The CHR rule $r := H \Leftrightarrow G \mid B_c, B_b$ has been translated from an ACT-R rule s. Let $s := L \Rightarrow R$ be an ACT-R rule in set normal form. Then r has the following form by Definition 5.14:

$$
\begin{aligned}
& delta(D) \uplus \{gamma(b, CVar(b), DVar(b)) \mid b \in \mathbb{B}\} \\
\Leftrightarrow \\
& \bigwedge_{=(b,t,P)\in L} chunk(CVar(b), t, P)\ in\ D \wedge DVar(b){=}0 \mid \\
& \{delta(D^*)\} \uplus \\
& \{gamma(b, MergeId(b), ResDelay(b)) \mid a(b,t,P) \in R\} \uplus \\
& \{gamma(b, CVar(b), DVar(b)) \mid a(b,t,P) \notin R\}, \\
& \bigwedge_{\alpha=a(b,t,P)\in R} (action(\alpha, D, CogState(\mathbb{B}), ResStore(b), ResId(b), ResDelay(b))) \\
\wedge\ & merge([ResStore(b) : a(b,t,P) \in R], D') \\
\wedge\ & merge([D, D'], D^*) \\
\wedge\ & \bigwedge_{a(b,t,P)\in R} (map(D, D', ResId(b), MergeId(b)))
\end{aligned}
$$

where B_b are the built-in constraints of the body and B_c the CHR constraints.

Since r is applicable in ρ, by Definition 5.4, ρ must have the following form:

$$\rho \equiv \langle H \uplus \mathbb{G}; G \wedge \mathbb{C}; \mathbb{V} \rangle$$

Thereby, $\mathbb{G}$ is a multi-set of user-defined constraints and $\mathbb{C}$ a conjunction of built-in constraints. Since by definition of ACT-R states, $H \uplus \mathbb{G}$ must be ground and therefore there must be some built-in constraints Θ in $\mathbb{C}$ that bind those variables to the values from the state:

$$\rho \equiv \langle H \uplus \mathbb{G}; \Theta \wedge G \wedge \mathbb{C}'; \mathbb{V} \rangle$$

Θ is a conjunction of constraints of the form $V = c$ for a variable V and a term c, that binds the variables from H and G to the values from the state, i.e. the variables $CVar(b), DVar(b)$ (for all $b \in \mathbb{B}$) and the variables appearing in the P from the guard are bound to some values from the state. We denote θ as the substitution that follows from Θ. Furthermore, ρ must contain the additional information υ:

$$\rho \equiv \langle H \uplus \mathbb{G}; \Theta \wedge G \wedge \upsilon \wedge \mathbb{C}''; \mathbb{V} \rangle$$

In the following, the ACT-R state σ is constructed. Let $\sigma := \langle \Delta; \gamma; \upsilon' \rangle$. Thereby, $chr(\Delta) = D$ and for all $b \in \mathbb{B} : \gamma(b) = (id_\Delta^{-1}(CVar(b)), DVar(b))\theta$ by Definition 5.8. Additionally, $\upsilon' := \upsilon$. We can now apply the same equivalences as in Lemma 6.2 in reverse order and get

$$
\begin{aligned}
\rho \equiv \langle \{ & delta([chunk(i_c, t, [\![val]\!]) \ :\ {=}(b,t,P) \in L \wedge \gamma(b) = (c,0) \wedge c = i_c{::}(t, val) \\
& \wedge\ \text{for all } (s,v) \in P : id_\Delta(val(s)) = v\theta]{+}{+}\mathbb{D})\} \\
& \uplus \{gamma(b, id_\Delta(c), 0) \mid {=}(b,t,P) \in L \wedge \gamma(b) = (c,0)\} \uplus \mathbb{G}; \Theta \wedge G \wedge \upsilon \wedge \mathbb{C}''; \emptyset \rangle
\end{aligned}
$$

From there it is clear that the ACT-R rule $s \sqsubseteq \sigma$ by Definition 4.1. The rule s can be applied to σ according to Definition 3.17 which yields

$$\sigma' := \langle \Delta \circ \Delta^*; \gamma'; \upsilon \wedge \upsilon^* \rangle$$

where $(\Delta^*, \gamma^*, \upsilon^*) \in I(s\theta, \sigma)$ and $\gamma'(b) := \begin{cases} (map_{\Delta,\Delta^*}(c), d) & \text{if } \gamma^*(b) = (c,d) \text{ is defined} \\ (map_{\Delta,\Delta^*}(c), d) & \text{otherwise, if } \gamma(b) = (c,d). \end{cases}$

By Definition 5.4, the application of r in ρ leads to the state ρ' with

$$\rho' \equiv \langle B_c \uplus \mathbb{G}; G \wedge B_b \wedge \Theta \wedge v; \mathbb{V} \rangle.$$

By Lemma 6.1, we get that B_b is equivalent to

$$D^* = chr(\Delta^*) \wedge D' = chr(\Delta \circ \Delta^*) \wedge$$
$$\bigwedge_{b \in dom(\gamma^*)} (MergeId(b) = id_{\Delta \circ \Delta^*}(map_{\Delta,\Delta^*}(c^*)) \wedge ResDelay(b) = d_b^*) \wedge v^* \wedge v$$

Hence, ρ' and σ' correspond directly to each other, i.e. $\rho' \equiv chr(\sigma')$ and therefore rule transitions in the translation are complete w.r.t. the abstract operational semantics of ACT-R. □

Theorem 6.5 (soundness and completeness of translation). *For an ACT-R model M, ACT-R states σ and σ' and their CHR counterparts $chr(M)$, $chr(\sigma)$ and $chr(\sigma')$ the following propositions are equivalent:*

(1) $\sigma \rightarrowtail \sigma'$ in the model M
(2) $chr(\sigma) \mapsto chr(\sigma')$

Proof. This follows directly from Lemmas 6.2 to 6.4. □

7 RELATED WORK

We want to highlight the contribution of [3] that we have used as a starting point to improve our work by unifying and extending it by our needs. We discuss this line of work in detail in Section 7.1. In Section 7.2 we summarize other work related to this paper.

7.1 Formal Semantics According to Albrecht et al.

The formalization according to [3] has been developed independently from our work in [14, 16]. Our very abstract semantics is based on it. Our abstract semantics from [16] is expressed in terms of this very abstract semantics based on [3] in [18] to show its compliance with other approaches of formalizing ACT-R.

[3] basically defines a general production rule system that works on sets of buffers and chunks without specifying actual matching, actions and effects for the sake of modularity and reusability. We briefly summarize the differences between the semantics in [3] and our very abstract semantics. For details, we refer to the original papers. The nomenclature in this paper differs in some points from the original paper [3] to unify it with our previous work. We omit module queries for the sake of brevity.

The sets of buffers $\mathbb{B}$ and action symbols A are defined as in Section 2. For the sake of brevity, we have omitted the so-called buffer queries in our definition of the very abstract semantics. Queries are an additional type of test on the left-hand side of a rule. The very abstract semantics can be easily extended by queries. We have adopted the definition of chunk types, chunks and cognitive states from the formalization of [3], although the set of chunks in [3] should be a multi-set-like structure as Example 3.3 shows. However, we have reduced the definition in our very abstract semantics by omitting the notion of a *finite trace*, which is a sequence $\gamma_0, \gamma_1, \cdots \in \Gamma^*$ of cognitive states. Those traces are used to compute the effects of an action. This definition seems inaccurate as the information of a finite trace that only logs the contents of the buffers at each step does not suffice to calculate sub-symbolic information. In typical definitions the calculation of production rule utilities needs the times of all rule applications that are not part of the trace. In other implementations and instantiations of ACT-R, there can be more additional information that is needed for sub-symbolic calculations. That is why we have extended the states by a parameter valuation function that

abstracts from the information needed and leaves it to the architecture to define which information is stored.

In [3], effects of actions with action symbol $\alpha \in A$ are defined by an interpretation function $I_\alpha : \Pi \to 2^{\Gamma_{\text{part}} \times 2^\Delta}$ (we have omitted queries as stated before). Similarly to the very abstract semantics, it assigns to each finite trace the possible effects of an action. Effects are a partial cognitive state that overwrites the contents of the buffers as in the very abstract semantics and a set $C \subseteq \Delta$ that defines the chunks that are removed. In typical implementations of ACT-R, the chunks in C are moved to the declarative module which explains the need to define such a set. We have generalized this information by the notion parameter valuations that can be manipulated by an interpretation function. This enables us to abstract from the specific concept of moving chunks to declarative memory in our abstract semantics for example. Note that in [3], the combination of interpretation functions to a rule interpretation is only stated informally. Additionally, we have extended the domain of an action interpretation function to actions, i.e. terms over the actions symbols in A, and states instead of only action symbols, since more information is needed to calculate, like the parameters of the actions (i.e. the slot-value pairs) and information from the state.

The production rule selection function $S : \Pi \to 2^\Sigma$ maps a set of applicable rules to each finite trace. In the very abstract semantics we have extended the domain from traces to a whole state since again additional information might be needed to resolve rule conflicts. With parameter valuations, we abstract from the information that is actually needed and leave it to the architecture definition. Additionally, our definition of selection function adds the notion of variable bindings that are not considered in [3].

The operational semantics in [3] is defined as a labeled, timed transition system with the following transition relation $\rightsquigarrow$ over time-stamped cognitive states from $\Gamma \times \mathbb{R}_0^+$:

$$(\gamma, t)_\pi \overset{r,d,\omega}{\rightsquigarrow} (\gamma', t')$$

for a production rule $r \in \Sigma$, an execution delay $d \in \mathbb{R}_0^+$, a set of chunks $\omega \subseteq \Delta$ and a finite trace $\pi \in \Pi$, if and only if $r \in S(\pi, \gamma)$, i.e. r is applicable in γ, the actions of r according to the interpretation functions yield γ' and $t' = t + d$.

Note that the set of chunks ω has been used but never defined in the original paper [3]. We suspect that it represents an equivalent to the chunk store from our abstract semantics, i.e. the used subset of all possible chunks (which is how Δ is defined according to the paper). Although we consider it an integral part of ACT-R, the matching of rules – and particularly binding of variables by the matching – is completely hidden in S or even not defined. On the one hand this simplifies exchanging the matching, on the other hand the function S should then be defined slightly different to enable proper handling of variable bindings and conflict resolution as we discuss in Section 3.

In the original semantics according to [3] there is no definition of what happens if there is no rule applicable, but there are still effects of e.g. requests that can be applied. We have treated this case by adding the *no rule* transition to the very abstract semantics.

7.2 Other Work

The reference implementation of ACT-R is described in a technical document [8] that defines the operational semantics mostly verbally and determines various technical details that are important for this exact implementation but not the architecture itself. In [18], we have defined a semantics that describes the core of the reference implementation of ACT-R and show that every transition possible in this refined semantics is also possible in the abstract semantics. This shows that formal reasoning about our abstract semantics is meaningful to actual implementations.

There are approaches of implementing ACT-R in other languages, for example a Python implementation [31] or (at least) two Java implementations [22, 28]. All those approaches do not concentrate on formalization and analysis, but only introduce new implementations. In [31] it is stated that exchanging integral parts of the ACT-R reference implementation is difficult due to the need of an extensive knowledge of technical details. They propose an architecture that is more concise and reduced to the fundamental concepts (that they also identify in their paper). However, their work still lacks a formalization of the operational semantics.

In [2], the authors summarize the work on semantics in the ACT-R context. They also come to the conclusion that there are only new implementations available that sometimes try to formalize parts of the architecture, but no formal definition of ACT-R's operational semantics. The authors use this result as a motivation for their work in [3].

We describe an adaptable implementation of ACT-R using Constraint Handling Rules (CHR) in [14, 15, 17] that is based on our formalization. Due to the declarativity of CHR, the implementation is very close to the formalization and easy to extend. This has been proved by exchanging the conflict resolution mechanism (that is an integral part of typical implementations) with very low effort [15]. Even the integration of refraction, i.e. inhibiting rules to fire twice on the same (partial) state, has been exemplified and can be combined with other conflict resolution strategies. The translation presented there is close to both the core of the reference implementation and the abstract semantics whose abstract parts are defined such that the match the reference implementation and some of its extensions.

8 CONCLUSION

In this paper, we have defined a very abstract operational semantics for ACT-R that can serve as a common base to analyze other operational semantics since it leaves enough room for various ACT-R variants. We then have refined this semantics to an abstract semantics.

Similar to the very abstract semantics, the abstract semantics abstracts from details like timings, latencies, forgetting, learning and specific conflict resolution. However, it defines the matching of rules and the processing of actions as they are typically found in ACT-R implementations. Hence, the abstract semantics concentrates on an abstract version of the typical implementations of ACT-R's procedural system. This makes it possible to reason about the general transitions that are possible in many ACT-R implementations.

The proposed operational semantics and its instances are extensible by exchanging or extending the individual abstract components of the semantics as argued in Section 4.2. For instance, new modules can be represented by new allowed additional information with corresponding interpretation and conflict resolution can be introduced and adapted by replacing or extending the definition of the selection function.

We have defined a translation of ACT-R models to Constraint Handling Rules (CHR) that is sound and complete w.r.t. our abstract semantics of ACT-R. To the best of our knowledge, the abstract semantics together with the sound and complete translation to CHR is the first formal formulation of ACT-R that is suitable to implementation.

The translation can be used for automated model analysis, since the faithful embedding in CHR opens many possibilities to reason about cognitive models by applying theoretical results from the CHR world to ACT-R models. For instance, confluence is the property that a program always yields the same result for the same input regardless of the order rules are applied. In CHR, confluence has been studied extensively. Most importantly, there are a decidable confluence criterion for terminating programs [13] and a completion method [1] that introduces new rules to a program in order to make it confluent. Although the human mind is probably not confluent because there are

many competing strategies with different outputs for the same task, there are always sequences of rules in cognitive models that should not be interfered by any other rule. A confluence criterion helps identifying the parts of the model that are not confluent. This can improve model quality by allowing for controlled non-confluence where desired guaranteeing the rules in the rest of the program not interfering with each other.

However, in practice confluence usually is too strict. With the notion of invariant-based confluence [12] only valid states that can be reached are considered, making confluence analysis applicable for practical use. However, this comes at the cost of losing decidability of the confluence test in general. Based on our work in this paper, we have described a method to test cognitive models in ACT-R for confluence in [20]. For this purpose, the translation scheme of this paper is used to transform ACT-R models to CHR programs. Additionally, an ACT-R invariant is defined that restricts the CHR state space to only valid ACT-R state representations according to the scheme in this paper. It is shown that this invariant is actually maintained by the translated CHR rules. Furthermore, it is proven that the invariant does not increase the complexity of the confluence test. Hence, depending on the decidability of the module requests defined by the ACT-R architecture of interest, the confluence test on translated ACT-R models is decidable for terminating models.

The definition of the operational semantics is independent of CHR and hence directly available to all ACT-R users with a formal background. By implementing the translation w.r.t. the standard ACT-R modules and the reference architecture, the CHR implementation and the confluence test can be made available to cognitive modelers without CHR experience. A first step of this implementation can be found in [14, 15]. To adapt the implementation or extend the translation or the confluence test, some understanding of both ACT-R and CHR are required.

For the future, we want to investigate how the confluence test for CHR can be used in practice to prove other program properties by relaxing the confluence condition even further with the notion of confluence modulo equivalence [10, 11, 21]. A program is considered confluent modulo an arbitrary, user-defined equivalence relation, if all equivalent states can be joined to a pair of equivalent states (both w.r.t. the user-defined equivalence relation). This could be used to analyze if an ACT-R model always yields a certain class of chunks for the same input. For instance it could be interesting for the modeler to know if a certain buffer always contains a chunk of a certain chunk type or with a certain value in some slot at the end of a computation. By that method, models could guarantee certain properties on their final states improving explanatory power and quality of cognitive models.

To make predictions on the probability that a cognitive model has a certain result, we plan to use the CHR extension CHRiSM [29, 30] that allows to enrich CHR rules with probabilities. It supports probability computation and even an expectation-maximization learning algorithm that could be used for parameter learning of cognitive models.

REFERENCES

[1] Slim Abdennadher and Thom Frühwirth. 1998. On Completion of Constraint Handling Rules. In *CP '98* (Pisa, Italy) *(Lecture Notes in Computer Science)*, M. J. Maher and J.-F. Puget (Eds.), Vol. 1520. Springer-Verlag, 25–39.

[2] Rebecca Albrecht, Michael Gießwein, and Bernd Westphal. 2014. 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* 15 (Suppl. 1) (2014), 27–28.

[3] Rebecca Albrecht and Bernd Westphal. 2014. 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* 15 (Suppl. 1) (2014), 79–81.

[4] John R. Anderson. 2007. *How can the human mind occur in the physical universe?* Oxford University Press.

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

[6] John R. Anderson and Christian Lebiere. 1998. *The atomic components of thought.* Lawrence Erlbaum Associates, Inc.

[7] Hariolf Betz and Thom Frühwirth. 2005. *A linear-logic semantics for Constraint Handling Rules.* Springer Berlin Heidelberg, Berlin, Heidelberg, 137–151. DOI: http://dx.doi.org/10.1007/11564751_13

[8] Dan Bothell. *ACT-R 6.0 Reference Manual – Working Draft.* Department of Psychology, Carnegie Mellon University, Pittsburgh, PA.

[9] Michael D Byrne. 2001. ACT-R/PM and menu selection: Applying a cognitive architecture to HCI. *International Journal of Human-Computer Studies* 55, 1 (2001), 41–84.

[10] Henning Christiansen and Maja H. Kirkeby. 2015. Confluence Modulo Equivalence in Constraint Handling Rules. In *Logic-Based Program Synthesis and Transformation: 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, Maurizio Proietti and Hirohisa Seki (Eds.). Springer International Publishing, 41–58. DOI: http://dx.doi.org/10.1007/978-3-319-17822-6_3

[11] Henning Christiansen and Maja H. Kirkeby. 2017. On proving confluence modulo equivalence for Constraint Handling Rules. *Formal Aspects of Computing* 29, 1 (2017), 57–95. DOI: http://dx.doi.org/10.1007/s00165-016-0396-9

[12] Gregory J. Duck, Peter J. Stuckey, and Martin Sulzmann. 2007. Observable confluence for Constraint Handling Rules. In *ICLP '07* (Porto, Portugal) *(Lecture Notes in Computer Science)*, V. Dahl and I. Niemelä (Eds.), Vol. 4670. Springer-Verlag, 224–239. DOI: http://dx.doi.org/10.1007/978-3-540-74610-2_16

[13] Thom Frühwirth. 2009. *Constraint Handling Rules.* Cambridge University Press.

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

[15] Daniel Gall and Thom Frühwirth. 2014. Exchanging Conflict Resolution in an Adaptable Implementation of ACT-R. *Theory and Practice of Logic Programming* 14, 4-5 (2014), 525–538. DOI: http://dx.doi.org/10.1017/S1471068414000180

[16] Daniel Gall and Thom Frühwirth. 2015. A formal semantics for the cognitive architecture ACT-R.. In *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)*, Hirohisa Seki Maurizio Proietti (Ed.), Vol. 8981. Springer. DOI: http://dx.doi.org/10.1007/978-3-319-17822-6

[17] Daniel Gall and Thom Frühwirth. 2015. An adaptable implementation of ACT-R with refraction in Constraint Handling Rules. In *Proceedings of the 13th International Conference on Cognitive Modeling*, N.A. Taatgen, M.K. van Vugt, J.P. Borst, and K. Mehlhorn (Eds.). 61–66.

[18] Daniel Gall and Thom Frühwirth. 2015. 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 (PPDP '15)*. ACM, New York, NY, USA, 114–124. DOI: http://dx.doi.org/10.1145/2790449.2790517

[19] Daniel Gall and Thom Frühwirth. 2016. Translation of Cognitive Models from ACT-R to Constraint Handling Rules. In *Rule Technologies. Research, Tools, and Applications (Lecture Notes in Computer Science)*, Jose Julio Alferes, Leopoldo Bertossi, Guido Governatori, Paul Fodor, and Dumitru Roman (Eds.), Vol. 9718. Springer International Publishing, Cham, 223–237. DOI: http://dx.doi.org/10.1007/978-3-319-42019-6_15

[20] Daniel Gall and Thom Frühwirth. 2017. A decidable confluence test for cognitive models in ACT-R. In *Rules and Reasoning: International Joint Conference, RuleML+RR 2017, London, UK, July 12–15, 2017, Proceedings (Lecture Notes in Computer Science)*, Stefania Costantini, Enrico Franconi, William Van Woensel, Roman Kontchakov, Fariba Sadri, and Dumitru Roman (Eds.), Vol. 10364. Springer International Publishing, Cham, 119–134. DOI: http://dx.doi.org/10.1007/978-3-319-61252-2_9

[21] Daniel Gall and Thom Frühwirth. 2018. Confluence Modulo Equivalence with Invariants in Constraint Handling Rules. In *Functional and Logic Programming (Lecture Notes in Computer Science)*, John P. Gallagher and Martin Sulzmann (Eds.), Vol. 10818. Springer International Publishing, Cham, 116–131. DOI: http://dx.doi.org/10.1007/978-3-319-90686-7_8

[22] A.M Harrison. 2008. jACT-R : Making cognitive modeling portable. (2008). http://jactr.org/

[23] Julien Martin and François Fages. 2007. From business rules to constraint programs in warehouse management systems. In *Doctoral programme of the 13th Intl. Conf. on Princ. and Pract. of Constraint Programming*.

[24] Frank Raiser. 2007. *Graph transformation systems in CHR.* Springer Berlin Heidelberg, Berlin, Heidelberg, 240–254. DOI: http://dx.doi.org/10.1007/978-3-540-74610-2_17

[25] Frank Raiser, Hariolf Betz, and Thom Frühwirth. 2009. Equivalence of CHR states revisited. In *6th International Workshop on Constraint Handling Rules (CHR)*, F. Raiser and J. Sneyers (Eds.). KULCW, Technical report CW 555, 33–48.

[26] Frank Raiser and Thom Frühwirth. 2008. Towards term rewriting systems in Constraint Handling Rules. In *The 5th Workshop on Constraint Handling Rules*. 19.

[27] Frank Raiser and Thom Frühwirth. 2011. Analysing graph transformation systems through Constraint Handling Rules. *Theory Practice of Logic Programming* 11, 1 (Jan. 2011), 65–109. DOI: http://dx.doi.org/10.1017/S1471068410000438

[28] Dario Salvucci. 2015. ACT-R: The Java Simulation & Development Environment – Homepage. (2015). http://cog.cs.drexel.edu/act-r/
[29] Jon Sneyers, Wannes Meert, and Joost Vennekens. 2009. CHRiSM: Chance rules induce statistical models. In *Proceedings of the Sixth International Workshop on Constraint Handling Rules (CHR'09)*. 62–76.
[30] Jon Sneyers, Wannes Meert, Joost Vennekens, Yoshitaka Kameya, and Taisuke Sato. 2010. CHR (PRISM)-based probabilistic logic learning. *Theory and Practice of Logic Programming* 10, 4-6 (2010), 433–447.
[31] Terrence C. Stewart and Robert L. West. 2007. Deconstructing and reconstructing ACT-R: Exploring the architectural space. *Cognitive Systems Research* 8, 3 (2007), 227–236.
[32] Ron Sun. 2008. Introduction to computational cognitive modeling. In *The Cambridge Handbook of Computational Psychology*, Ron Sun (Ed.). Cambridge University Press, New York, 3–19.
[33] Niels A. Taatgen and John R. Anderson. 2002. Why do children learn to say "broke"? A model of learning the past tense without feedback. *Cognition* 86, 2 (2002), 123–155.
[34] Niels A. Taatgen, C. Lebiere, and J.R. Anderson. 2006. Modeling paradigms in ACT-R. In *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*. Cambridge University Press, 29–52.

Received February 2017; revised August 2017; accepted May 2018