A Refined Operational Semantics for ACT-R

Investigating the Relations between Different ACT-R Formalizations

Daniel Gall Thom Frühwirth

Ulm University

{daniel.gall,thom.fruehwirth}@uni-ulm.de

Abstract

The popular cognitive architecture ACT-R is used in many cognitive models to explain cognitive features of human-beings. It has a well-defined psychological theory but lacks a formalization of its underlying computational system. This lack allows for technical ad-hoc artifacts in the original reference implementation. More importantly, formal analysis of cognitive models is not possible without a well-defined semantics. In prior work we have defined an abstract operational semantics for ACT-R's production system that is suitable for model analysis. It abstracts from details like timings and conflict resolution methods. However, to describe the behavior of ACT-R implementations a more refined semantics is needed.

In this paper, we first introduce a new very abstract operational semantics for ACT-R that serves as formal base to compare different semantics. We define an improved version of the abstract semantics as an instance of our new very abstract semantics. Furthermore, we present a more refined operational semantics that also captures details from actual ACT-R implementations. We show that the refined semantics is sound w.r.t. the abstract semantics. This makes model analysis with the abstract semantics suitable for real-world ACT-R models.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Operational semantics; J.4 [*Social and Behavioral Sciences*]: Psychology; D.3.2 [*Language Classifications*]: Specialized application languages; I.6.8 [*Types of Simulation*]: Discrete event

Keywords ACT-R, cognitive modeling, production rule systems, formal operational semantics

1. Introduction

Adaptive Control of Thought – Rational (ACT-R) [7, 8] is a cognitive architecture that is very popular in computational cognitive modeling. It offers a unified psychological theory as well as a language and framework to implement psychological models about human cognition. Although it is well investigated from the psychological point of view, ACT-R lacks a formal foundation of its computational concepts. This inhibits computational analysis of cognitive models and leads to various different implementations with technical artifacts. To improve understanding of the fundamental concepts and to enable analysis of cognitive models and comparison of different implementations, a formal description of ACT-R's operational semantics is needed:

> "Unfortunately, in contrast to the precisely defined mathematics of stochastic optimal control, the ACT-R literature is marred by a lack of specificity and consistency. The ACT-R definition is distributed across a vast collection of journal articles and monographs, and no single text is sufficient to provide a complete definition. Important parts of the ACT-R definition vary from source to source, with no explanation as to why the change was made, or even an acknowledgement that the change had occurred at all." [25]

The need of a formalization is also indicated by the fact that there are several approaches to formalize ACT-R that we discuss in detail in section 6 [4, 5, 17, 21].

In prior work [17], we have presented an operational semantics of ACT-R that abstracts from details like timings, latencies and a concrete conflict resolution mechanism. This abstraction simplifies modeling and analysis while capturing the fundamental parts of the system. Calls to external modules are abstracted as state-dependent functions. This leads to a description of ACT-R's fundamental concepts that is independent from the exact instantiation and used modules. By representing rule conflicts as non-deterministic state transitions, our abstract operational semantics considers all possible state transitions. Hence, our description is independent from the used conflict resolution mechanism. This allows to exchange parts of the ACT-R theory and enables analysis of different implementations.

Independent from our work, a formal semantics of ACT-R is presented in [4]. This semantics already includes parts of the refinement we need for the continuation of our work and hence is valuable to validate and extend our previous efforts. However, it still leaves some parts open that are needed for the formalization of a concrete instantiation (i.e. a description of actual implementations) of ACT-R and lacks some definitions that we already have established in our abstract semantics and that we need for our analysis tools, for example the notions of matching and variable bindings.

In this paper, we first define a very abstract semantics in the spirit of the formalization according to Albrecht and Westphal [4]. It simplifies analysis of the relations between different operational semantics. We redefine our abstract semantics from [17] as an instance of the very abstract semantics and get rid of some inaccuracies and notational overload we have introduced there. As before, the abstract semantics is easy to analyze. This is further substantiated by the sound and complete translation scheme of ACT-R models to Constraint Handling Rules (CHR). This translation scheme enables the application of theoretical results of CHR to it, e.g. the decidable confluence test [13, section 5.2].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPDP '15, July 14–16, 2015, Siena, Italy.
Copyright © 2015 ACM 978-1-4503-3516-4/15/07…$15.00.
http://dx.doi.org/10.1145/2790449.2790517

We continue the work from [17] and introduce a more refined semantics that adds concepts like timing, latencies and conflict resolution. Those concepts have been omitted in our abstract semantics to simplify analysis. This gives an exact formal description of existing ACT-R implementations. We show soundness of the refined semantics w.r.t. the abstract operational semantics, i.e. that every derivation in the refined semantics is a valid derivation in the abstract semantics. This result paves the way for model analysis by making our simple abstract semantics usable for e.g. confluence and termination analysis.

Altogether, the contributions of this paper are

- a comparison and discussion of the recent work on ACT-R semantics,
- a very abstract operational semantics that serves as theoretical toolkit to analyze relations between different actual instantiations of ACT-R semantics,
- an improved version of our abstract operational semantics from [17],
- a refined operational semantics that continues our work to an analytical framework of ACT-R and
- a soundness result that relates our abstract semantics to the refined semantics. This makes the abstract semantics suitable for analysis.

2. Description of ACT-R

In this section, we describe ACT-R informally. For a detailed introduction to the system, we refer to [6–8, 24]. 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 modeling language learning [23] or improving human computer interaction by the predictions of a cognitive model [10]. The components of the ACT-R architecture even have been mapped to brain regions [6, 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 [24]. Computational cognitive models are described clearly and unambiguously since they are executed by a computer producing detailed simulations of human behavior [22]. 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 figure 1.

As shown in figure 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.

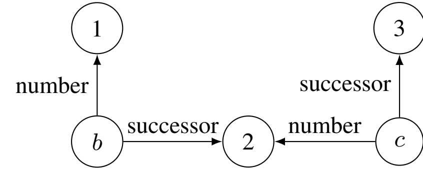

Figure 1. Two count facts with names b and c that model the counting chain 1, 2, 3.

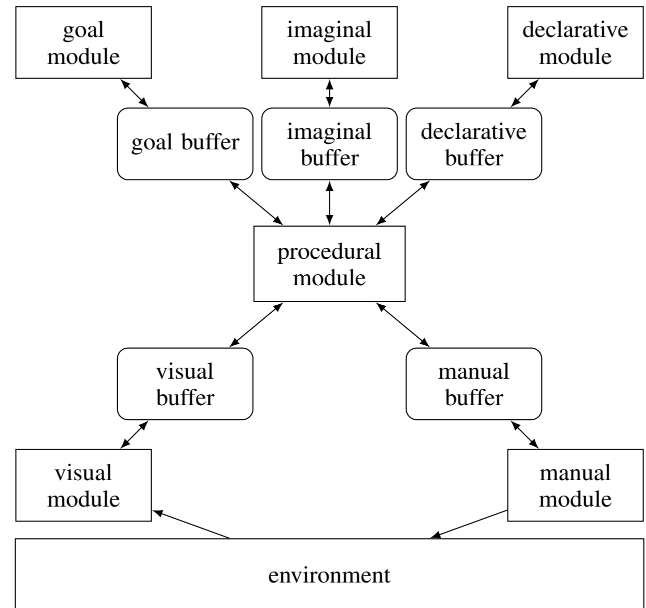

Figure 2. Modular architecture of ACT-R. This illustration is inspired by [24] and [8].

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, latencies, 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 is dependent 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. Similarly 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 [17]. 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.

The syntax of ACT-R is defined over two possibly infinite, disjoint sets of (constant) symbols $\mathcal{C}$ and variable symbols $\mathcal{V}$. An ACT-R model consists of a set of types $\mathbb{T}$ 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 the buffer $b \in \mathcal{C}$ and $P \subseteq \mathcal{C} \times (\mathcal{C} \cup \mathcal{V})$ is a set of slot-value pairs (s, v) where $s \in \mathcal{C}$ and $v \in \mathcal{C} \cup \mathcal{V}$. This means that only the values in the slot-value pairs can consist of both constants and variables. The right-hand side $R \subseteq \mathcal{A}$ of a rule is a finite set of actions where $\mathcal{A} = \{a(b, t, P) \mid a \in A, b \in \mathcal{C}, t \in \mathcal{C} \text{ and } P \subseteq \mathcal{C} \times (\mathcal{C} \cup \mathcal{V})\}$. I.e. 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. Usually, the action symbols are defined as $A := \{=, +, -\}$ for modifications, requests and clearings respectively.

We define the function $vars$ that 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.

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. 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.

We now give an example rule and informally explain its behavior.

Example 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:*

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

The first action modifies the chunk in the goal. A modification cannot change the type, that is why we just add an anonymous variable denoted by the underscore symbol in the type specification. 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 a refined semantics that extends our abstract semantics by sub-symbolic details. To compare the refined and abstract semantics, we first give them a common theoretical foundation that

is based on the formalization according to Albrecht and Westphal – 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 [4]. We compare our work with the work from [4] in section 6.2.

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 rules Σ. Figure 4 summarizes what is defined by the architecture and the model.

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 1 (chunk types)**.** *A* typing function $\tau : \mathbb{T} \to 2^{\mathcal{C}}$ *maps each type from the set* $\mathbb{T} \subseteq \mathcal{C}$ *to a finite set of allowed slot names.*

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 2 (chunk store)**.** *A* chunk store *is a multi-set of tuples* (t, val) *where* $t \in \mathbb{T}$ *is a chunk type and* $val : \tau(t) \to \Delta$ *is a function that maps each slot of the chunk (determined by the type* t*) to another chunk. We denote the chunk store of all possible chunks with* Δ*. For a chunk* $c = (t, val)$*, the following functions are defined:*

- $type(c) = t$ *and*
- $slots(c) = val$.

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. Note that a chunk store can contain multiple elements with the same values that still are unique entities representing different concepts. We will see this in the following example: We model our well-known example from figure 1 as a chunk store:

Example 2 (chunk store of natural numbers)**.** *The chunk store from figure 1 can be modeled as follows:*

- *The set of types is* $\mathbb{T}_2 = \{number, succ\}$.
- *The typing function* $\tau_2 : \mathbb{T} \to 2^{\mathcal{C}}$ *is defined as* $\tau(number) = \emptyset$ *and* $\tau(succ) = \{number, successor\}$.
- *We have the following chunks in our store* Δ_2*:*
  - *the unique entities* $1, 2, 3$ *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* $val_b(s) = \begin{cases} 2 & \text{if } s = number \\ 3 & \text{if } s = successor \end{cases}$

3.2 States

We first define the individual parts of an ACT-R state. The notion of a *cognitive state* defines which chunks are currently in which buffer and therefore visible to the production system that can only match chunks in buffers.

Definition 3 (cognitive state)**.** *A* cognitive state γ *is a function* $\mathbb{B} \to \Delta \times \mathbb{R}_0^+$ *that maps each buffer to a chunk and a delay. The set of cognitive states is denoted as* Γ*, whereas* Γ_{part} *denotes the set of* partial cognitive states*, i.e. cognitive states that are partial functions and do not necessarily map each buffer to a chunk. We define the following functions to access the individual parts of a cognitive state* γ*: If* $\gamma(b) = (c, d)$ *for an arbitrary buffer* b*, then*

- $chunk(\gamma(b)) = c$ *and*
- $delay(\gamma(b)) = d$.

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.

ACT-R adds a sub-symbolic level to the symbolic concepts that have been defined so far and that distinguish it from other production rule systems. To gather information from the sub-symbolic layer, we add the concept of (sub-symbolic) parameter valuations that hold the additional information needed to calculate sub-symbolic values. These valuations can be altered by an abstract function as we will see in section 3.3. We define a parameter valuation as follows:

Definition 4 (parameter valuation)**.** *A* parameter valuation *is a function* $\upsilon : \Upsilon \to \mathcal{C}$ *with a domain specified by the set of allowed parameters* Υ *and the target set* $\mathcal{C}$*, the set of constants.*

The set of allowed parameter valuations Υ is defined by the concrete architecture. We give an example for parameter valuations:

Example 3 (parameter valuations)**.** *We want to memorize the last application of each rule. Therefore, we introduce a new parameter by defining the set of allowed parameter valuations as* $\Upsilon_3 = \{t_r \mid r \in \Sigma\}$*. For the rule* r^* *that has been applied at simulation time* 1.5*, the valuation* υ *in a state is* $\upsilon(t_{r^*}) = 1.5$.

Note that we will use rule application times similarly when defining the sub-symbolic information in the refined semantics (c.f. section 5.1).

We now define ACT-R states as follows:

Definition 5 (very abstract state)**.** *A* very abstract state *is a tuple* $\langle \gamma; \upsilon; t \rangle^{\mathbb{V}}$ *where* γ *is a cognitive state in the sense of definition 3,* $\upsilon : \Upsilon \to \mathcal{C}$ *is a parameter valuation function (called* additional information*),* $t \in \mathbb{R}_0^+$ *is a time and* $\mathbb{V}$ *is a set of variable bindings. The state space is denoted with* $\mathcal{S}_{va}$.

A set of variable bindings $\mathbb{V} \subseteq \mathfrak{V}$ consists of equality relations over constants and variables ($\mathcal{C} \cup \mathcal{V}$) that define the values of variables in the operational semantics. The symbol $\mathfrak{V}$ denotes the set of all variable bindings over $\mathcal{C} \cup \mathcal{V}$.

We continue our running example by defining a very abstract state with one of the chunks defined in example 2.

Example 4. *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. Figure 3 illustrates the state. The formal definition is:*

- $\mathbb{T}_4 = \mathbb{T}_2 \cup \{g\}$ *where* $\mathbb{T}_2$ *is the set of types from example 2.*
- $\tau_4(t) = \begin{cases} \{current\} & \text{if } t = g \\ \tau_2(t) & \text{otherwise.} \end{cases}$
- $\Delta_4 = \Delta_2 \cup \{(g, val_{goal})\}$ *where* $val_{goal}(current) = X$ *and* $X \in \mathcal{V}$.
- $\sigma_0 = \langle \gamma_0; \emptyset; 0 \rangle^{\mathbb{V}}$
- $\gamma_0(goal) = ((g, val_{goal}), 0)$

- $\gamma_0(retrieval) = (b, 1)$ *where b is defined as in example 2.*
- $\mathbb{V} = \{X = 1\}$

Note that we have introduced a variable X that is bound to chunk 1 in $\mathbb{V}$.

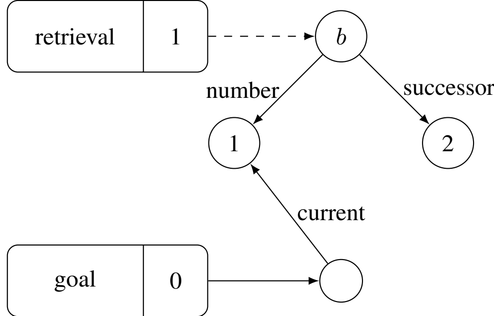

Figure 3. Visual representation of the very abstract state defined in example 4. The dashed arrow signifies that the chunk in the retrieval buffer is not yet visible (as indicated by the delay right of the buffer's name).

3.3 Operational Semantics

We now define the state transition system of the very abstract semantics. As in every production rule system, we first define how matching rules are chosen. Therefore, we introduce a selection function S that is defined by the architecture and maps a state to a set of matching rules and the variable bindings implied by the application of the rule.

Definition 6 (selection function)**.** *A* selection function *is a function $S : \mathcal{S}_{va} \to 2^{\Sigma \times 2^{\mathfrak{V}}}$ that maps a state to a set of pairs $(r, \mathbb{V})$ where $r \in \Sigma$ is a production rule and $\mathbb{V} \subseteq \mathfrak{V}$ is a set of variable bindings.*

For actual implementations of ACT-R, the result of S is usually restricted to sets with zero or one element, but for abstract definitions there can also be more than one rule. The function S usually defines a notion of matching and makes sure that only rules can fire that match visible information in the buffers, i.e. chunks that are not delayed by a time greater than zero.

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 7 (interpretation of actions)**.** *An* interpretation of actions *is a function $I : \mathcal{A} \times \mathcal{S}_{va} \to 2^{\Gamma_{\text{part}} \times \mathcal{C}^{\Upsilon}}$ where $\mathcal{C}^{\Upsilon}$ denotes the set of all functions $\Upsilon \to \mathcal{C}$.*

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

Note that the interpretation of an action can return more than one possible effect. 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 [9].

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 8 (combination operator $\sqcup$ for effects)**.** *If A and B are sets of tuples (f, g) where f and g are functions, then their* combination *is defined as*

$$A \sqcup B := \{(f \cup f', g \cup g') \mid (f, g) \in A \text{ and } (f', g') \in B\}.$$

In the following example, we combine two sets of effects, i.e. sets of tuples of partial cognitive states and parameter valuations:

Example 5 (combination of effects)**.** *For two sets of effects $E = \{(\gamma_1, \upsilon_1), (\gamma_2, \upsilon_2)\}$ and $E' = \{(\gamma_1', \upsilon_1')\}$ the combination is*

$$E \sqcup E' = \{(\gamma_1 \cup \gamma_1', \upsilon_1 \cup \upsilon_1'), (\gamma_2 \cup \gamma_1', \upsilon_2 \cup \upsilon_1')\}.$$

The result still is a valid set of effects, since each element of the combination is still a tuple of functions.

We now define the interpretation function $I : \Sigma \to 2^{\Gamma_{\text{part}} \times \Upsilon}$ that maps a rule to all its possible effects to the cognitive state and additional information.

Definition 9 (interpretation of rules)**.** *A rule $r := L \Rightarrow R$ is interpreted by an interpretation function $I : \Sigma \times \mathcal{S}_{va} \to 2^{\Gamma_{\text{part}} \times \Upsilon}$ that 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 : $\Gamma_{\text{part}} \times \Upsilon \to \Gamma_{\text{part}} \times \Upsilon$ that applies some more effects at the end of the rule application and is defined by the architecture,*
- *The interpretation has the result*

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

where the apply function is applied to each member of the combination set. Hence, all possible effects of the rules are combined such that the corresponding cognitive states and valuations belong together as shown in example 5 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 [9]. For the additional information in υ the concrete architecture has to ensure this property.

We now define the operational semantics as the state transition system $(\mathcal{S}_{va}, \mapsto)$:

Definition 10 (very abstract operational semantics)**.** *The transition relation $\mapsto : \mathcal{S}_{va} \times \mathcal{S}_{va}$ in the* very abstract operational semantics *of ACT-R is defined as follows:*

Apply *For a fresh variant $r' := r[\bar{x}/\bar{y}]$ of rule r with $vars(r) = \bar{x}$, the following transitions are possible:*

$$\frac{(r', \mathbb{V}^*) \in S(\sigma), (\gamma_{\text{part}}, \upsilon^*) \in I(r')}{\sigma := \langle \gamma; \upsilon; t \rangle^{\mathbb{V}} \mapsto \langle \gamma'; \upsilon'; t' \rangle^{\mathbb{V} \cup \mathbb{V}^*}}$$

where

- $\gamma'(b) = \begin{cases} \gamma_{\text{part}}(b) & \text{if defined} \\ (c, d \ominus \delta) & \text{otherwise, if } \gamma(b) = (c, d), \end{cases}$

- $x \ominus y = \begin{cases} x - y & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$ *for two numbers* $x, y \in \mathbb{R}_0^+$,
- $\upsilon'(p) = \begin{cases} \upsilon^*(p) & \text{if defined} \\ \upsilon(p) & \text{otherwise,} \end{cases}$ *and*
- $t' = t + \delta$ *for a delay* $\delta \in \mathbb{R}_0^+$ *defined by the concrete instantiation of ACT-R.*

We use $\mapsto^r$ *to denote that the transition applies rule* r.

No Rule

$$\frac{C(\sigma)}{\sigma := \langle \gamma; \upsilon; t \rangle^{\mathbb{V}} \mapsto \langle \gamma'; \upsilon; \theta(\sigma) \rangle^{\mathbb{V}}}$$

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,*
- $\theta : \mathcal{S}_{va} \to \mathbb{R}_0^+$ *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.*

We also write $\mapsto^{no}$ *to emphasize that the* no rule *transition is used.*

Note that chunks are immutable and therefore changes in the cognitive state always exchange the whole chunk in a modified buffer.

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, the others keep their contents. The same applies for parameters: They keep their value except for those where υ^* 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 $\theta(\sigma)$. 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 $\theta(\sigma) := t + d^*$ 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 [9]. 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.

Architecture	Model
$\mathcal{C}$: 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	
δ: rule delay	
Υ: allowed parameter valuations	
S: rule selection function	
I: interpretation functions	

Figure 4. Parameters of the very abstract semantics that must be defined by the architecture or the cognitive model respectively.

The definition of our very abstract semantics leaves parts to be defined by the actual architecture and the model. Figure 4 summarizes what has to be defined by an architecture and a model.

4. Abstract Semantics as Instance

We redefine our abstract semantics from [17] as an instance of the very abstract semantics from section 3. With the redefinition we correct some technicalities and reduce the syntactic overload of the original statement. The detailed changes are addressed in section 6.1.

4.1 Definition

First of all, we define the notion of matchings:

Definition 11 (matching)**.** *A buffer test* $=(b, ct, P)$ *for a buffer* $b \in \mathbb{B}$ *testing for a type* ct *and slot-value pairs* $P \subseteq \mathcal{C} \times (\mathcal{C} \cup \mathcal{V})$ matches *a state* $\sigma := \langle \gamma; \upsilon; t \rangle^{\mathbb{V}}$, *written* $=(b, ct, P) \sqsubseteq \sigma$, *if and only if the following holds:*

$$\mathbb{V} \to (\forall s \in \mathcal{C}, \forall v \in \mathcal{C} \cup \mathcal{V} : (s, v) \in P \to \exists b' \in \mathbb{B} : b' = b \wedge \gamma(b') = ((ct, val), 0) \wedge val(s) = v)$$

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*.*

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

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

States The states from definition 5 are defined upon a set of allowed parameter valuations Υ_{abs}. We set $\Upsilon_{abs} := \emptyset$ since the abstract semantics has no sub-symbolic layer. Additionally, 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 $\langle \gamma; \emptyset; 0 \rangle^{\mathbb{V}}$ where $\gamma \in \Gamma$ is a cognitive state.

Selection Function The rule selection in the abstract semantics is simply defined as $S_{abs}(\sigma) := \{(r, \{r \sqsubseteq \sigma\}) \mid r \in \Sigma \wedge r \sqsubseteq \sigma\}$. Hence we select all matching rules in state σ and bind the variables from the rules to their actual values from the matching.

Effects For a state $\sigma = \langle \gamma; \emptyset; 0 \rangle^{\mathbb{V}}$ the interpretation function I_{abs} for actions in the abstract semantics is defined as follows:

- $I_{abs}(=(b, t, P), \sigma) = \{(\gamma_p, \emptyset)\}$ for modifications where
  - $\gamma_p(b) := ((type(\gamma(b)), val_b), 0)$ and

- for the old slot-value pairs in the cognitive state γ that are defined as $old = slots\,(chunk\,(\gamma\,(b)))$ the new values are:

$$val_b(s) := \begin{cases} v & \text{if } (s, v) \in P \\ old(s) & \text{otherwise.} \end{cases}$$

  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 ($chunk(\gamma(\cdot))$). 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. Note that modifications are deterministic, i.e. that there is only one possible effect.

- $(\gamma_p, \emptyset) \in I_{abs}(+(b, t, P), \sigma)$ for requests if
  - $request_b : \mathbb{T} \times 2^{\mathcal{C} \times (\mathcal{C} \cup \mathcal{V})} \times \mathcal{S}_{va} \rightarrow 2^{\Delta \times \mathbb{R}_0^+ \times \mathcal{C}^{\Upsilon}}$ 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.
  - We assume that $(c_b, d_b, \upsilon_b) \in request_b(t, P, \sigma)$. Then $\gamma_p(b) := \begin{cases} (c_b, 1) & \text{if } d_b > 0 \\ (c_b, 0) & \text{otherwise,} \end{cases}$

  Note that the sub-symbolic information in the result of the request is discarded in the interpretation function of the abstract semantics.

- These definitions trivially ensure the functional character of the additional information (i.e. υ only maps at most one element to an element in the domain) as required by its definition: υ is the empty function.
- The function *apply* from definition 9 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 $\sigma := \langle \gamma; \emptyset; t \rangle^{\mathbb{V}}$ 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 $\theta(\sigma) := t$, i.e. the time is not adjusted.

We now extend our running example by a derivation in the abstract semantics:

Example 6 (abstract semantics)**.** *We begin with the state σ_0 from example 4. Then, the following derivations are possible. Note that we assume that variable bindings in $\mathbb{V}$ are directly applied to the state for the sake of readability.*

$$\sigma_0 \mapsto^{no} \langle \gamma_1; \emptyset; 0 \rangle^{\mathbb{V}}$$
$$\mapsto^{inc} \langle \gamma_2; \emptyset; 0 \rangle^{\mathbb{V}} =: \sigma_2$$

where

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

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 1 is applicable. If we assume that* $request_{retrieval}(succ, \{(number, 2)\}, \sigma) = (c_{req}, 1, \emptyset)$ *for all states σ where* $c_{req} = (succ, \{(number, 2), (successor, 3)\})$*, 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 figure 5. Note that in this state again the* no rule *transition is possible.*

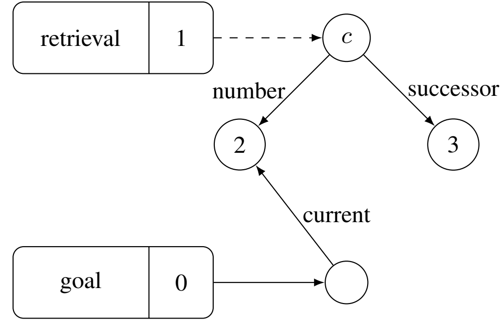

Figure 5. Visual representation of state σ_2 from example 6.

5. Refined Semantics as Instance

The abstract semantics does not use any sub-symbolic information or timings and is highly non-deterministic in the choice of rules and application of requests. It can be used for analysis, but to describe actual ACT-R implementations, we define a refined semantics in section 5.1. It coincides with our CHR implementation, where e.g. specific conflict resolution mechanisms can be plugged to the interfaces of the abstract system [15]. This is expressed by the abstract sub-symbolic functions that are parameters of the refined semantics and can be specified by the actual instantiation. In the reference implementation of ACT-R (and other implementations) the exchangeable conflict resolution mechanism is replaced by one fixed method. We have described some of the common mechanisms in [15]. They can be easily adopted in our refined semantics.

In section 5.2 we show that every derivation in the refined semantics is a valid derivation in the abstract semantics. This makes the latter suitable for analysis of ACT-R models without covering all the details of the refined semantics. Since the refined semantics describes typical ACT-R implementations, this is an important result.

5.1 Definition of the Semantics

We define the refined semantics as an instance of the very abstract semantics to make it easier comparable to the abstract semantics.

The architectural parameters from the definition in section 3 are defined as follows:

States There are many possibilities for parameters that can be stored by modules in the state. In our basic version of the refined semantics, we define sub-symbolic information for the typical ACT-R instantiation with a declarative module. Hence, the set of allowed parameter valuations Υ_{ref} contains the symbols $t_c^{a,i}$ where $c \in \Delta$ is a chunk and $i \in \mathbb{N}$ is an integer. Additionally, it contains the symbols $t_r^{u,i}$ where $r \in \Sigma$ is a rule and $i \in \mathbb{N}$ is an integer. The value $t_c^{a,i}$ is a time where chunk c has been used, i.e. retrieved. This value is needed to find the activation of the chunk. Analogously, $t_r^{u,i}$ is an application time of rule r needed to calculate the utility.

Selection Function We define the refined rule selection by using the selection function from the abstract semantics (see section 4.1): $S_{ref}(\sigma) := conf_res(S_{abs}(\sigma), \sigma)$. Usually, the conflict resolution function chooses at most one rule by a conflict resolution mechanism that is defined by the function $conf_res$.

Effects Requests can be defined for many different modules. Hence, the sub-symbolic information varies (hidden in the function $request_b$ defined by each module). We give the definition for the retrieval buffer. Hence, the interpretation function I_{ref} for actions is defined as follows:

- $I_{ref}(\alpha, \sigma) = I_{abs}(\alpha, \sigma)$ if α is a term with functor $=$, i.e. a modification. This means that we simply use the definition from the abstract semantics since the sub-symbolic layer is not affected by a modification.
- $I_{ref}(+(b, t, P), \sigma) = \{(\gamma_p, \upsilon)\}$ for requests if
  - $sel_req_b(request_b(t, P, \sigma), \sigma) = (c_b, d_b, \upsilon)$ where
  - $sel_req_b : 2^{\Delta \times \mathbb{R}_0^+} \times \mathcal{S}_{va} \to \Delta \times \mathbb{R}_0^+$ is a function defined by the architecture.

  Then

  - $\gamma_p(b) := (c_b, d_b)$ and
  - $\upsilon'(t_{c_b}^{a,i}) := t$ for a fresh integer $i \in \mathbb{N}$ if $b = retrieval$. Other modules can define their values analogously.

  The request is calculated by the same procedure as in the abstract semantics (by function $request_b$). However, one individual result is selected and applied after a delay d_b specified by the requested buffer. During that delay time, the buffer appears to be empty for the production rule system. The selection function is defined by the requested module. Additionally, sub-symbolic information is adjusted. For instance, if the retrieval buffer is requested, the sub-symbolic information is adjusted: A new retrieval time for the requested chunk is added. This information is needed to calculate its activation. Note that no sub-symbolic information is overwritten, since a new integer i is introduced to mark the parameter. Additionally, υ' is a (partial) function since it only is defined for the retrieval buffer. Hence, the definition of additional information as a function is not harmed.
- For a rule $r \in \Sigma$, the function $apply$ from definition 9 is defined as follows:

  $apply_r(\gamma_{\text{part}}, \upsilon) := (\gamma_{\text{part}}, \upsilon \cup \{(t_r^{u,i}, t)\})$ for an integer i.

  This means that with the rule application, the application time of rule r is memorized. This information is needed to calculate rule utilities.

Rule Application Delay The delay of a rule application is set to $\delta := 0.5$ seconds as defined by the ACT-R reference manual [9].

No Rule Transition For a state $\sigma := \langle \gamma; \upsilon; t \rangle^{\mathbb{V}}$ we define the parameters of the *no rule* transition as follows:

- We first define $d^* := \min\{d \mid d > 0, b \in \mathbb{B}, \gamma(b) = (c, d)\}$.
- Then the condition C is defined as: $\sigma \in C$ if and only if $S_{ref}(\sigma) = \emptyset$ and d^* exists, i.e. a pending request is only applied explicitly if no rule can be applied and execution is stuck. Otherwise, if there are applicable rules, then the requests will be applied implicitly by the progress of time. Additionally, a minimal delay $d^* > 0$ must exist, i.e. there is invisible information in the cognitive state.
- For all buffers $b \in \mathbb{B}$ the update function is defined as $[update(\sigma)](b) := (c, d - d^*)$ if $\gamma(b) = (c, d)$.
- $\theta(\sigma) := t + d^*$, i.e. the current time is forwarded by the minimum delay in the current state.

5.2 Soundness of the Refined Semantics

In this section, we show that each derivation of the refined semantics is also a derivation in the abstract semantics (if only the cognitive state is regarded). This result is important to allow for using the abstract semantics to reason about actual models in the refined semantics.

Therefore, we define a state transfer function that maps each refined state an abstract state:

Definition 12 (state transfer function)**.** *The function trans : $\mathcal{S}_{ref} \to \mathcal{S}_{abs}$ that transfers refined to abstract states is defined as $trans(\langle \gamma; \upsilon; t \rangle^{\mathbb{V}}) = \langle \gamma'; \emptyset; 0 \rangle^{\mathbb{V}}$ where $\gamma'(b) = (c_b, 1)$ if and only if $\gamma(b) = (c_b, d)$ and $d > 0$ and otherwise $\gamma'(b) = \gamma(b)$. The function can also be applied on partial state descriptions, e.g. tuples of a cognitive state and additional information.*

This definition drops the sub-symbolic information and the timing when transforming the state. Additionally, chunks in cognitive states are divided into two classes – one where the chunk is visible to the system (delay is zero), and one where it is not (otherwise). Intuitively this is reasonable because with the lack of a notion of time, the delay only affects the visibility. Note that the transfer function is not injective, since each of the abstract states can have been transferred from multiple refined states. Hence, it has no inverse function which is not necessary to show soundness.

First, we show that rules are applicable in both transition systems:

Lemma 1 (applicability)**.** *For all refined states σ and rules r the following holds: If r is applicable in σ, then it is also applicable in $trans(\sigma)$.*

Proof. The rule application transition from definition 10 has the condition that there is a tuple $(r, \mathbb{V}^*) \in S(\sigma)$. For the refined semantics, this selection function S is defined as $S_{ref}(\sigma) = conf_res(S_{abs}(\sigma), \sigma)$ where $conf_res$ is the conflict resolution function that selects a sub-set of rules from a given set of rules (by using information from a given state). The set of rules the conflict resolution function can choose from is defined by S_{abs}, the selection function of the abstract semantics. □

After having shown that applicable rules in the refined semantics are also applicable in the abstract semantics, we show that the effects of the rules lead to the same cognitive state:

Lemma 2 (effects of rules)**.** *The interpretation functions of the abstract (I_{abs}) and the refined semantics (I_{ref}) are equivalent w.r.t. the transformation function trans. This means that if $I_{ref}(r) = \{(\gamma_{ref}, \upsilon_{ref})\}$ for a request r then there is some $(\gamma_{abs}, \emptyset) \in I_{abs}(r)$ such that $trans(\gamma_{ref}, \upsilon_{ref}) = (\gamma_{abs}, \emptyset)$.*

Proof. For modifications the proposition holds trivially, since $I_{ref}(\alpha, \sigma) = I_{abs}(\alpha, \sigma)$ for all terms α with functor $=$.

The definitions for requests compare as follows: Let the interpretation of a request in the refined semantics be $I_{ref}(+(b, t, P), \sigma) = \{(\gamma_{ref}, \upsilon_{ref})\}$ where the cognitive state and sub-symbolic information are defined as

- $\gamma_{ref} = (c_{ref}, d_{ref})$ where
- $sel_req(request_b(t, P, \sigma)) = (c_{ref}, d_{ref}, \upsilon_{ref})$.

Since $sel_req(request_b(t, P, \sigma)) \subseteq request_b(t, P, \sigma)$, there is an element $(\gamma_{abs}, \emptyset) \in I_{abs}(+(b, t, P), \sigma)$ where $\gamma_{abs} = (c_{abs}, d_{abs})$ with $c_{ref} = c_{abs}$. The delay $d_{abs} = 0$ if and only if $d_{ref} = 0$, otherwise $d_{abs} = 1$. This is the same definition as in the transition function. The sub-symbolic information of the abstract state is always $\emptyset$, in the transfer function and the definition of the interpretation of a request itself. Hence, the effects of a rule application are equivalent. □

Lemma 3 (soundness of rule applications)**.** *For all refined states σ and rules r the following holds: If $\sigma \mapsto^{r}_{ref} \sigma'$ then $trans(\sigma) \mapsto^{r*}_{abs} trans(\sigma')$*

Proof. According to lemma 2, the effects of rules are sound. It remains to show that the composition of the resulting state is sound w.r.t. the transfer function. The only difference in the both semantics is the parameter δ: In the abstract semantics it is zero, in the refined semantics it can be greater that zero. For the effects of rules, lemma 2 shows that the delay does not matter for the effects of the rules, since the transfer function reduces it for the requested buffers.

For the remaining buffers b, the very abstract semantics defines that $\gamma_b = (c, d \ominus \delta)$. This means, that no new information other than the one in a modified or requested state can become visible in the abstract semantics. In contrast to this, the refined semantics can make other information visible. Let σ^{ref} be the state in the refined semantics after the application of the rule r and V the set of buffers that become visible in the refined but not in the abstract semantics. Then for every buffer $b \in V$ the *no rule* transition can be applied in the abstract semantics (since the only condition is that there is invisible information). This information is then visible and leads to an equivalent state w.r.t. the transfer function, i.e. that the resulting state σ^{no} after applying all necessary no rule transitions is equivalent to $trans(\sigma^{ref})$. □

Note that a one to one correspondence of transitions in the abstract and refined semantics is not possible, since it can be necessary to use the *no rule* transition in the abstract semantics to achieve the same result of a rule application in the refined semantics. For soundness, this is allowed.

We have shown that the *apply* transition is sound. We continue with the same result for the *no rule* transition:

Lemma 4 (soundness of no rule transition)**.** *For all refined states $\sigma, \sigma' \in \mathcal{S}_{ref}$: If $\sigma \mapsto^{no}_{ref} \sigma'$, then $trans(\sigma) \mapsto^{*no}_{abs} trans(\sigma')$. This means that if the* no rule *transition is possible in the refined semantics for σ with the resulting state σ', it is also possible for $trans(\sigma)$ with result $trans(\sigma')$ in the abstract semantics. The result is allowed to be reached after more than one step.*

Proof. We first show soundness of applicability of the *no rule* transition: The *no rule* transition is applicable in a state $\sigma := \langle \gamma; \upsilon; t \rangle^{\mathbb{V}}$ in the refined semantics if and only if $C_{ref}(\sigma)$ is true. This is the case if no rule is applicable (i.e. $S_{ref}(\sigma) = \emptyset$). Furthermore, a minimal delay $d^* := \min\{d \mid d > 0, b \in \mathbb{B}, \gamma(b) = (c, d)\}$ must exist. For the abstract semantics, C_{abs} is true, if and only if there is at least one delay $d > 0$ in the cognitive state (c.f. section 4.1). This is implied by C_{ref}. Hence, we have shown soundness of applicability.

It remains to show that the effects of the *no rule* transition are equivalent in both semantics. Hence, we compare the $update$ functions of both semantics. Let d^* be the minimal delay in σ as defined before. Then, $\left[update_{ref}(\sigma)\right](b) = (c, d - d^*)$ if $\gamma(b) = (c, d)$ for all buffers. In the abstract semantics, one buffer b^* is chosen in each transition whose content is made visible: $\left[update_{abs}(\sigma)\right](b^*) := (c, 0)$ for one b^*, if $\gamma(b^*) = (c, d)$ and $d > 0$.

In the refined semantics, more than one piece of information can be made visible at once by forwarding the time by the minimal delay, since the minimum is not necessarily unique. Let $B := \{b \in \mathbb{B} \mid \gamma(b) = (c, d^*)\}$ be the set of buffers with the minimal delay. By applying *no rule* to $trans(\sigma)$ with one buffer $b_1 \in B_\sigma$, the *no rule* transition stays applicable for another $b \in B - \{b_1\}$ in the subsequent state until we reach a state where $B = \emptyset$. Then we have reached a cognitive state $\sigma_{abs} = trans \sigma'$, since we have made all chunks visible that have the minimal delay. This means that there is a sequence of *no rule* applications in the abstract semantics that leads to the desired state $trans(\sigma')$.

Note that in σ' time has been adjusted by $\theta(\sigma)$. In the abstract semantics time is not updated and hence remains zero. We get the same result by applying $trans$ to σ. □

Again, a one to one correspondence of state transitions is not possible for the *no rule* transition, because only one buffer is made visible in the abstract semantics whereas multiple buffers can become visible by forwarding time in the refined semantics. For soundness, it suffices to show that every final state of the refined semantics can also be reached in the abstract semantics. This is the proposition of the following theorem:

Theorem 1 (soundness)**.** *For all states $\sigma, \sigma' \in \mathcal{S}_{ref}$: If $\sigma \mapsto^{*}_{ref} \sigma'$, then $trans(\sigma) \mapsto^{*}_{abs} trans(\sigma')$.*

Proof. The proposition follows directly from lemmas 1, 3 and 4. □

6. Related Work

We want to highlight two contributions that are particularly related to our work and influenced its results: our former definition of the abstract semantics and the semantics by Albrecht and Westphal. Hence, we discuss those two semantics in the following sections 6.1 and 6.2. In section 6.3 we summarize other work related to this paper.

6.1 Former Definition of the Abstract Semantics

In the definition of the abstract semantics from [17], the sets of buffers $\mathbb{B}$ and types of actions A are defined similarly to section 2. Chunk types have been omitted for the sake of brevity, but reintroduced in section 4 of this paper. The definition of chunk stores differs in two points:

- Chunks have unique names in [17] and chunk stores are therefore sets instead of multi-sets. However, the concepts are interchangeable and not crucial to the operational semantics.
- In [17], chunks can be modified in place. The definition in this paper (that has been derived from [4]) is more accurate to the reference implementation [9] and therefore has been introduced to our abstract semantics in section 4.

Otherwise, the relations Isa and $\mathrm{HasSlot}$ from [14] and [17] coincides with the definition of a chunk as a tuple with type and a slot-value function val.

The states in [17] do not define a notion of delay in the buffers. In definition of a cognitive state in the very abstract semantics, 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. A cognitive state roughly corresponds to a timed version of the Holds relation from the abstract semantics in [17]. There the delays have been modeled by non-deterministic transitions that can be applied between rule applications (since there is no notion of time in the abstract semantics) and by explicit clearing of requested buffers in the semantics. Both concepts can be transferred from one to another.

The notion of interpretations of actions has been addressed in [17] by the definition of add and delete lists for each type of actions that is similar to the very abstract semantics. However, add and delete lists are deterministic in [17]. The non-determinism that is made explicit in the interpretation functions (by returning a set of *possible* effects) has been reached by the definition of the interface to requests, the $request_b$ functions, that can be non-deterministic. This interface has been reintroduced in the definition of the abstract semantics in this paper, where it is part of the interpretation of a request.

6.2 Formal Semantics According to Albrecht and Westphal

The formalization according to Albrecht and Westphal [4] has been developed independently from our work in [14, 17]. Our very abstract semantics is based on it. Albrecht and Westphal basically define 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 Albrecht and Westphal semantics 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 [4] 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 Albrecht and Westphal formalization, although the set of chunks in [4] should be a multi-set as example 2 shows. However, we have reduced the definition in our very abstract semantics by omitting the notion of a *finite trace*, which is a sequence $\gamma_0, \gamma_1, \dots \in \Gamma^*$ 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 [4], 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 [4], 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 by Albrecht and Westphal.

The operational semantics in [4] 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 [4]. 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 Albrecht and Westphal 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.

6.3 Other Work

There are approaches of implementing ACT-R in other languages, for example a Python implementation [21] or (at least) two Java implementations [18, 20]. All those approaches do not concentrate on formalization and analysis, but only introduce new implementations. Stewart and West state that exchanging integral parts of the ACT-R reference implementation is difficult due to the need of an extensive knowledge of technical details [21]. 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 [5], 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 [4].

We describe an adaptable implementation of ACT-R using Constraint Handling Rules (CHR) in [14–16] 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.

7. Conclusion

In this paper, we have defined a *very abstract operational semantics* for ACT-R that extends the formalization according to Albrecht and Westphal. It is the common base to analyze other operational semantics since it leaves enough room for various ACT-R variants. We then have redefined our abstract semantics as an instance of the very abstract semantics. The abstract semantics simplifies analysis of ACT-R models by abstracting from details like timings, latencies, forgetting, learning and specific conflict resolution.

To show that our abstract semantics is suitable for analysis of real-world ACT-R models, we have defined a *refined operational semantics* that covers the details abstracted by the abstract semantics using the very abstract semantics. We have shown *soundness* of the refined w.r.t to the abstract semantics. This result paves the way to an *analytical framework* for ACT-R models.

For the future, we want to investigate how we can use our abstract semantics for analysis. Since we have shown a, w.r.t. to our abstract semantics, sound and complete *translation scheme* of ACT-R models to Constraint Handling Rules [17], we could apply theoretical results from the CHR world to ACT-R models. For instance, there is a decidable confluence test [1], a completion algorithm that fixes confluence [2], a test for operational equivalence [3] etc. whose application to cognitive models can be used for validation and quality improvements. There is also an automated confluence tester for CHR programs [19]. It has to be examined what has to be done to reasonably apply the analytical tools of CHR to cognitive models. For instance, confluence usually is too strict in practice since it includes states that can never be reached by the program or model. With the notion of observable confluence [11], only valid states that can be reached are considered, making confluence analysis applicable for practical use. Another interesting property of a cognitive model is its time complexity: Since humans are able to solve certain (subsets of) problems efficiently (even for larger problem instances), the corresponding cognitive model should also have a corresponding time complexity. This constrains the model space to only models of that time complexity. CHR offers methods to analyze the complexity of a program [12]. Those results could be lifted to ACT-R models to analyze their time complexity.

References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In G. Smolka, editor, *CP '97: Proc. Third Intl. Conf. Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 1997.

[2] S. Abdennadher and T. Frühwirth. On completion of Constraint Handling Rules. In M. J. Maher and J.-F. Puget, editors, *CP '98*, volume 1520 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, Oct. 1998. ISBN 3-540-65224-8.

[3] S. Abdennadher and T. Frühwirth. Operational equivalence of CHR programs and constraints. In J. Jaffar, editor, *CP '99*, volume 1713 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, Oct. 1999. ISBN 3-540-66626-5.

[4] R. Albrecht and B. Westphal. 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)*, volume 15 (Suppl. 1) of *Cognitive Processing*, pages 79–81. Springer, 2014.

[5] R. Albrecht, M. Gießwein, and B. Westphal. Towards formally founded ACT-R simulation and analysis. In *Proceedings of the 12th Biannual conference of the German cognitive science society (Gesellschaft für Kognitionswissenschaft)*, volume 15 (Suppl. 1) of *Cognitive Processing*, pages 27–28. Springer, 2014.

[6] J. R. Anderson. *How can the human mind occur in the physical universe?* Oxford University Press, 2007. ISBN 978-0-19-539895-3.

[7] J. R. Anderson and C. Lebiere. *The Atomic Components of Thought*. Lawrence Erlbaum Associates, Inc., 1998.

[8] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological Review*, 111 (4):1036–1060, 2004. ISSN 0033-295X.

[9] D. Bothell. *ACT-R 6.0 Reference Manual – Working Draft*. Department of Psychology, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

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

[11] G. J. Duck, P. J. Stuckey, and M. Sulzmann. Observable confluence for Constraint Handling Rules. In V. Dahl and I. Niemelä, editors, *ICLP '07*, volume 4670 of *Lecture Notes in Computer Science*, pages 224–239. Springer-Verlag, Sept. 2007. .

[12] T. Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. In A. D. Pierro and H. Wiklicky, editors, *QAPL '01: Proc. First Intl. Workshop on Quantitative Aspects of Programming Languages*, volume 59(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[13] T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009. ISBN 978-0-521-87776-3.

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

[15] D. Gall and T. Frühwirth. Exchanging conflict resolution in an adaptable implementation of ACT-R. *Theory and Practice of Logic Programming*, 14:525–538, 2014. ISSN 1475-3081. .

[16] D. Gall and T. Frühwirth. An adaptable implementation of ACT-R with refraction in Constraint Handling Rules. In N. Taatgen, M. van Vugt, J. Borst, and K. Mehlhorn, editors, *Proceedings of the 13th International Conference on Cognitive Modeling*, pages 61–66, 2015.

[17] D. Gall and T. Frühwirth. A Formal Semantics for the Cognitive Architecture ACT-R. volume 8981 of *Lecture Notes in Computer Science*. Springer, 2015. ISBN 978-3-319-17821-9. .

[18] jACT-R. The Homepage of jACT-R. URL `http://jactr.org/`.

[19] J. Langbein, F. Raiser, and T. Frühwirth. A state equivalence and confluence checker for CHR. In P. Van Weert and L. De Koninck, editors, *CHR '10*. K.U.Leuven, Department of Computer Science, Technical report CW 588, July 2010.

[20] D. Salvucci. ACT-R: The Java Simulation & Development Environment – Homepage. URL `http://cog.cs.drexel.edu/act-r/`.

[21] T. C. Stewart and R. L. West. Deconstructing and reconstructing ACT-R: exploring the architectural space. *Cognitive Systems Research*, 8(3): 227–236, 2007. ISSN 13890417.

[22] R. Sun. Introduction to computational cognitive modeling. In R. Sun, editor, *The Cambridge Handbook of Computational Psychology*, pages 3–19. Cambridge University Press, New York, 2008.

[23] N. A. Taatgen and J. R. Anderson. Why do children learn to say broke? a model of learning the past tense without feedback. *Cognition*, 86(2): 123155, 2002.

[24] N. A. Taatgen, C. Lebiere, and J. Anderson. Modeling paradigms in ACT-R. In *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation.*, pages 29–52. Cambridge University Press, 2006.

[25] J. Whitehill. Understanding ACT-R – an outsiders perspective, 2013. URL `http://arxiv.org/abs/1306.0125`.