ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Faculty of Engineering
and Computer Science
Institute of Software Engineering
and Compiler Construction

A Rule-Based Implementation of ACT-R Using Constraint Handling Rules

Master Thesis
University of Ulm

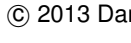
Daniel Gall
daniel.gall@uni-ulm.de

Reviewers:
Prof. Dr. Dr. Thom Frühwirth
Prof. Dr. Slim Abdennadher

Consultant:
Prof. Dr. Dr. Thom Frühwirth

2013

“A Rule-Based Implementation of ACT-R Using Constraint Handling Rules”
September 5, 2013



Typesetting: PDF-LaTeX 2ε
Graphics created with: TikZ
LaTeX Template: Guido de Melo
Printed by: Kommunikations- und Informationszentrum (kiz), University of Ulm

Abstract

Computational Cognitive Modeling is a research field at the interface of computer science and the cognitive sciences. It enables researchers to build detailed cognitive models upon a cognitive architecture which provides some general assumptions about human cognition to simulate human behaviour. By conducting the same experiments with humans and an executable computational cognitive model, the plausibility of a model can be verified.

ACT-R is a cognitive architecture which is widely used in the field of computational cognitive modeling. It is a production-rule system whose models are expressed by a set of declarative knowledge elements – the data – and a set of rules. The rules match the data and, if applied, have effects on the data. *Constraint Handling Rules (CHR)* is a high-level rule-based formalism and programming language which offers a lot of analysis tools for its programs.

This work first presents some fundamental aspects of ACT-R and then introduces a translation of ACT-R models to CHR programs. The implementation of ACT-R using Constraint Handling Rules shows that cognitive models can be expressed very elegantly in CHR. Since CHR provides a lot of analysis tools and its rules have a defined declarative semantics, this approach may support the verification of cognitive models. Additionally, the translation of the models is automated by a compiler, so it is not compulsory for modelers to learn a whole new language and legacy models may be translated to CHR.

Contents

1 Introduction

Computational psychology or *computational cognitive modeling* is a research field in the cognitive sciences. Among the other approaches – mathematical and verbal-conceptual modeling – "computational modeling appears to be the most promising approach in many ways and offers the flexibility and the expressive power that no other approaches can match" [Sun08, p. vii]. It explores human cognition by implementing detailed computational models that enable computers to execute them and simulate human behaviour [Sun08, p. 3]. By conducting the same experiments with humans and with simulations of the suggested underlying cognitive models, the plausibility of models can be checked and models can be improved gradually. This approach is illustrated in figure 1.1. Furthermore, an important benefit of computational models is that they are – as a matter of principle – detailed and have a clear semantics, in order to be executed by a computer. Hence, the problem of being imprecise about a model definition, as it may appear with verbal modeling, can be overcome.

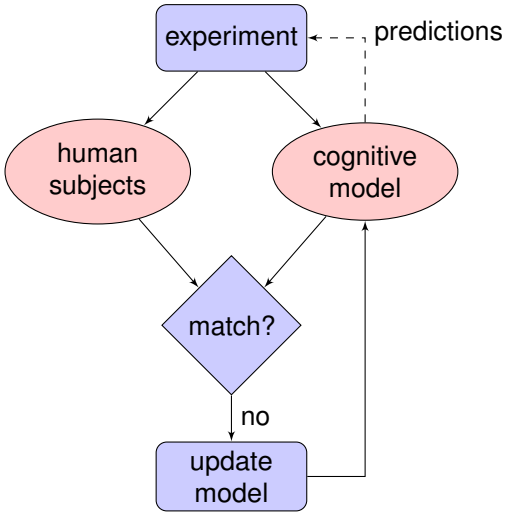


Figure 1.1: A typical workflow in computational cognitive modeling. After a model has been created, an experiment is designed to test the predictions of the model. The experiment is performed by humans and the cognitive model. Afterwards the results are checked and the model can be adapted. [Abo]

However, psychology is experiencing a movement towards specialization [And+04], i.e. there are a lot of independent, highly specialized fields that lack a more global view, which impedes cognitive modeling where a very detailed and complete view is necessary for execution:

> "In 1972, [. . . it] seemed to me, also, that the cognitive revolution was already well in hand and established. Yet, I found myself concerned about the theory psychology was developing. [. . .] I tried to make that point by noting that what psychologists mostly did for theory was to go from dichotomy to dichotomy." [New90, pp. 1 sq.]

Newell suggested in his book from 1990 [New90], that for developing consistent models of cognition it is necessary to create a theory that tries to put all those highly specialized components together [And+04, p. 1036]. He therefore introduced the term *cognitive architecture* [And07, p. 5], which today can be defined as "a specification of the structure of the brain at a level of abstraction that explains how it achieves the function of the mind" [And07, p. 7]. A cognitive architecture provides the ability to create models for specific cognitive tasks [TLA06, p. 29] by offering "representational formats together with reasoning and learning mechanisms to facilitate modeling" [TLA06, p. 29]. Nevertheless, a cognitive architecture should also constrain modeling – ideally it should only allow "cognitive models that are cognitively plausible" [TLA06, p. 29]. The relation of cognitive models and architecture is illustrated in figure 1.2.

Adaptive Control of Thought-Rational (ACT-R) is a cognitive architecture, that "is capable of interacting with the outside world, has been mapped onto brain structures, and is able to learn to interact with complex dynamic tasks" [TLA06, p. 29], so its theory is well-investigated. It also is one of the most popular cognitive architectures in the field [RT05] and provides an implementation that allows modelers to execute their models by a computer and hence offers a platform for computational cognitive modeling as defined above.

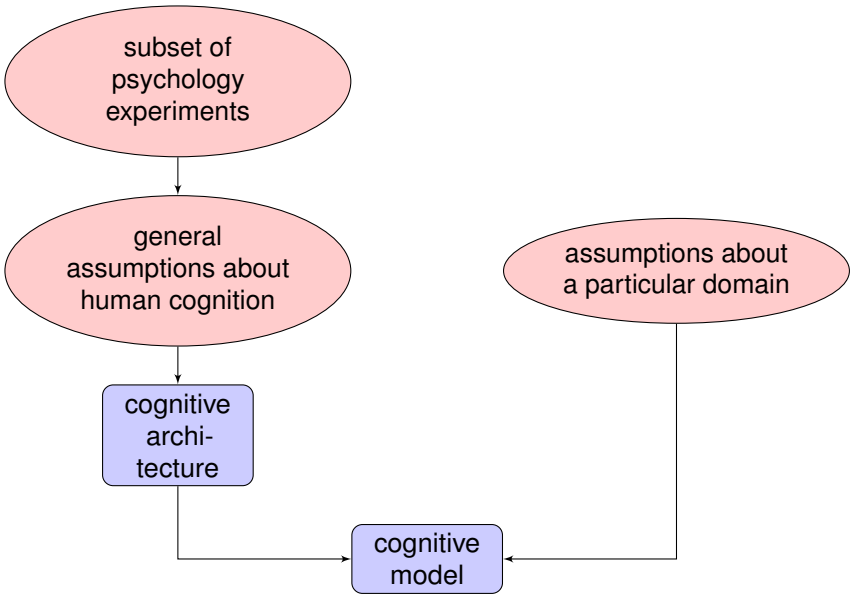


Figure 1.2: Cognitive models usually are built upon a cognitive architecture by adding domain specific knowledge to the context of the architecture. The cognitive architecture contains general knowledge derived from psychological experiments. [Abo]

1.1 Motivation and Goal

This work is the first step towards a full-featured implementation of ACT-R using Constraint Handling Rules and hence towards a computational cognitive modeling platform based entirely on logical rules. Constraint Handling Rules (CHR) is a high-level rule-based programming language with several very well-defined operational and declarative semantics [Frü09, pp. 49 sqq.], so the meaning of a CHR program is logically defined. Additionally, there are a lot of methods to analyze CHR programs [Frü09, pp. 96 sqq.] and there are a lot of helpful properties of CHR programs, like the anytime and online property [Frü09, pp. 83 sqq.]. The declarativity of CHR programs does not affect the efficiency, so every algorithm which can be implemented efficiently in an imperative language can also be implemented efficiently in CHR and the constant slow-down of CHR compared to a C program is very low using high-optimizing compilers [Frü09, p. 94].

Since models are expressed by production rules in ACT-R, the idea to implement such models in CHR seems likely, because, at a first glance, the semantics of a CHR rule does not seem to be very different from an ACT-R production rule, so a lot of work like

the *efficient* implementation of the matching process can be saved. Due to the clear semantics and the several analysis methods, the analysis and the inspection of the logical implications of a cognitive model is enhanced by this work. The testing of confluence and operational equality is decidable for CHR programs, so these properties can now be determined automatically for ACT-R models [Frü10, p. 4]. The declarative semantics facilitates the investigation of the correctness and the implications and predictions of a cognitive model. Furthermore, by translating the concepts of ACT-R to CHR, it is possible to compare those concepts with many other rule-based formalisms like term rewriting systems, functional programming or other production rule systems like OPS5 and to transfer ideas from one system to another[Frü09, pp. 141 sqq.].

In ACT-R, models are usually defined with ground variables. Since CHR provides the ability of executing abstract programs with unknown variables [Frü10, p. 4], this work allows to run ACT-R models where not all of the variables are known. Additionally, CHR programs work incrementally, so the execution of an ACT-R model can be stopped and intermediate results can be obtained by inspecting the constraint store. This property also allows to add new knowledge to the model while it is executed [Frü09, p. 176].

The aim of this work is to implement the basic concepts of ACT-R and stick as closely as possible to the theory using the original implementation as a reference. Hence, a goal is to find translation schemes of ACT-R production rules to CHR rules and the implementation of the underlying cognitive architecture in CHR. Thereby, the fundamental concepts of ACT-R are identified and distinguished from ad hoc artifacts as much as possible. However, the implementation is capable of executing original ACT-R models and therefore the fundamental, simplistic view of the theory is gradually refined in the work and each detail is motivated individually to show for which purposes it is necessary to introduce a technical detail.

1.2 Related Work

There are several implementations of the ACT-R theory in different languages. First of all, there is the official ACT-R implementation in Lisp [Acta] often referred to as the *vanilla* implementation. There are a lot of extensions to this implementation, which partly have been included to the original package in later versions like the ACT-R/PM extension that has been included in ACT-R 6.0 [Botb, p. 264]. The implementation comes with an experiment environment which offers a graphical user interface to load, execute and observe models which communicates through a network interface with the ACT-R core.

Stewart and West have built an implementation in Python, which also had the aim to simplify and harmonize parts of the ACT-R theory by finding the central components of the theory [SW06; SW07]. The authors describe another approach of implementing ACT-R without sticking too much to the classic implementations. For instance, the architecture has been reduced to only two components (the procedural and the declarative module which will be described in section 2) and build the rest of the architecture by using those two modules and combining them in different ways. Additionally, there is no possibility to translate traditional ACT-R models automatically to Python code. Hence, the focus of the work was different from the aims in this work.

Furthermore, there are two different implementations in Java: *jACT-R* [Jacb] and *ACT-R: The Java Simulation & Development Environment* [Javb]. The latter one is capable of executing original ACT-R models and offers an advanced graphical user interface. The focus of the project was to make ACT-R more portable with the help of Java, since Lisp's "extensibility for different task implementations and different hardware platforms has been lagging compared to more modern languages" [Java]. In jACT-R, the focus was to offer a clean and exchangeable interface to all the components, so different versions of the ACT-R theory can be mixed [Jaca] and models are defined using XML. Due to the modular design defining various interfaces which can be exchanged, jACT-R is highly adaptable to personal needs. However, since Java is the host language, there is no expected gain in declarativity and model analysis for both implementations.

There are approaches to implement psychological models using declarative and logic programming languages. In [PS07] Pereira and Saptawijaya present computational models for cognitive reasoning in the context of moral dilemmas using prospective logic programs. Balduccini and Girotto show in [BG10] how psychological knowledge can be formalized and reasoning over this knowledge can be achieved using answer set programming. However, both approaches are detached from a broadly distributed cognitive architecture like ACT-R. Nevertheless, in [BG10, p. 726] it is emphasized, that for psychological "theories of a more qualitative or logical nature [. . .] are not easy to formalize in" the way of neural-networks or similar approaches, but need a more abstract approach.

1.3 Overview

This work is divided in several parts. First of all, an important task was the identification of the fundamental parts of the ACT-R theory. Hence, a description of the theory is given in chapter 2. Chapter 3 gives a very brief introduction to Constraint Handling Rules. The result of the implementation of ACT-R in CHR is described in chapter 4, which first

formalizes some of the parts of the ACT-R theory before it suggests how to implement the afore described concepts in Constraint Handling Rules. Afterwards, the work of chapter 4 is demonstrated in some example models in section 5. Chapter 6 summarizes the work and gives an outlook to future work.

The work is accompanied by a CD with the source code and a digital version of the text (see appendix A). The current version of both the source code and the text can also be downloaded at GitHub:

Source Code `https://github.com/danielgall/chr-actr/`
Thesis `https://github.com/danielgall/master-thesis/`

For the matter of archiving, the versions of the submission date of the thesis are both tagged with `thesis` in the repository.

2 Description of ACT-R

Adaptive Control of Thought-Rational (ACT-R) is a cognitive architecture, that allows to implement cognitive models that are executable by a computer to produce experimental results that can be compared to experimental data from experiments that have been conducted with humans.

Because of the underlying theory which is the basis for the ACT-R cognitive architecture, modeling is facilitated, since the underlying concepts have not to be modelled again and again. On the other hand, it constrains the modeling process to, ideally, only plausible models. When talking about ACT-R, one can refer to the theory or the implementation. The theory gives a view which abstracts from implementational details that may be concerned when talking about implementation which is a specific instantiation of the theory [Actb, unit 1, p. 1]. In this work, implementation always refers to the vanilla Lisp implementation that can be downloaded from [Acta].

In this chapter, a short overview over the theory of ACT-R is given. First, the description is informal to provide a general image of how ACT-R works. Then, some important parts of the system are defined more formally in chapter 4, as soon as it is needed in the implementation. All of the information in this chapter refers to the theory. Implementation is discussed in chapter 4. A lot of the information in this chapter is based on [And07; And+04; TLA06], where a much more comprehensive discussion of the ACT-R theory including complex examples, referrings to the neuro-biology and the reasons why this particular modeling of human cognition has been chosen. In this work, only the basic concepts of ACT-R are presented.

2.1 Procedural and Declarative Knowledge

A central idea of ACT-R is the distinction between *declarative* and *procedural knowledge.* The declarative knowledge consists of simple facts, whereas the procedural knowledge contains information on what to do with those facts.

2.1.1 Modular organization

This approach leads to a modular organization of ACT-R with modules for each purpose needed to simulate human cognition. Figure 2.1 provides an overview of some of the default modules of ACT-R. For example, the declarative module stores the factual information (the declarative knowledge), the visual module perceives and processes the visual field, the procedural module holds the procedural information and controls the computational process.

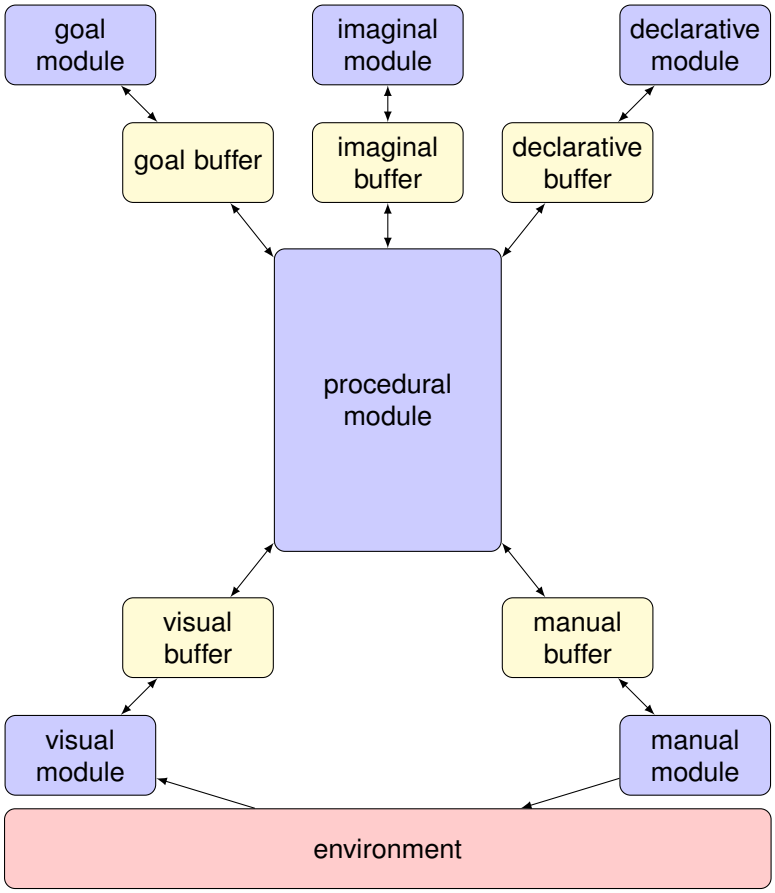


Figure 2.1: The modular organization of ACT-R. Each module has an associated buffer which can communicate with the procedural module. The perceptual/motor modules can interact with the external world. [And+04, fig. 1]

Each module is independent from the other modules and computations in the modules can be performed parallel to other modules, for instance: The declarative module can

search a specific fact while the visual module processes the visual field. Additionally, within one module computations are executed massively parallel, e.g., the visual module can process the entire visual field at once to determine the location of a certain object, which implies the processing of a huge amount of data at a time.

However, each module can perform its computation only locally and has no access to computations of other modules. To communicate, modules have associated *buffers*, where they can put a limited amount of information – one primitive knowledge element – and the procedural module can access each of these buffers. The information in a buffer could be one single fact retrieved from declarative memory or one visual object from the visual field perceived by the visual module. Information between modules is exchanged by the procedural module taking information from one buffer and putting it into another (with an optional computation on the way). This leads to a serial bottleneck in the computation, since every communication between modules has to go its way through the procedural module.

In figure 2.2 the general functionality of ACT-R is illustrated: The computational process is controled by the *recognize-act-cycle*: The procedural information is stored as rules that have a *condition* and an *action*. The condition refers to the so-called *working memory*, which basically is the content of all the buffers. In the recognize-phase of the cycle, a suitable rule that matches the current state of the working memory is searched. If the condition of a rule holds, it *fires* and performs its actions – this is the act-phase of the cycle. Those actions can cause changes on the buffers, so the next rule may match the current state in the next recognize-part of the cycle. In the following sections, some of the modules and their precise interaction will be described in more detail.

2.1.2 Declarative Knowledge

The declarative module organizes the factual knowledge as an associative memory. I.e., it consists of a set of concepts that are connected to each other in a certain way. Such elementary concepts are represented in form of chunks that can be seen as basic knowledge elements. They can have names, but those names are not critical for the description of the facts and just for readability in the theory. So, the definition of a chunk is based only on its connections. However, in implementations, chunk names are used as unique identifiers for chunks.

Chunks can have slots that are connected to other chunks or primitive elements. Such an element can be regarded as a chunk without any slots. For instance, the fact 5 + 2 = 7 can be modeled as a chunk that is connected to the numbers 5, 2 and 7 (see figure 2.3).

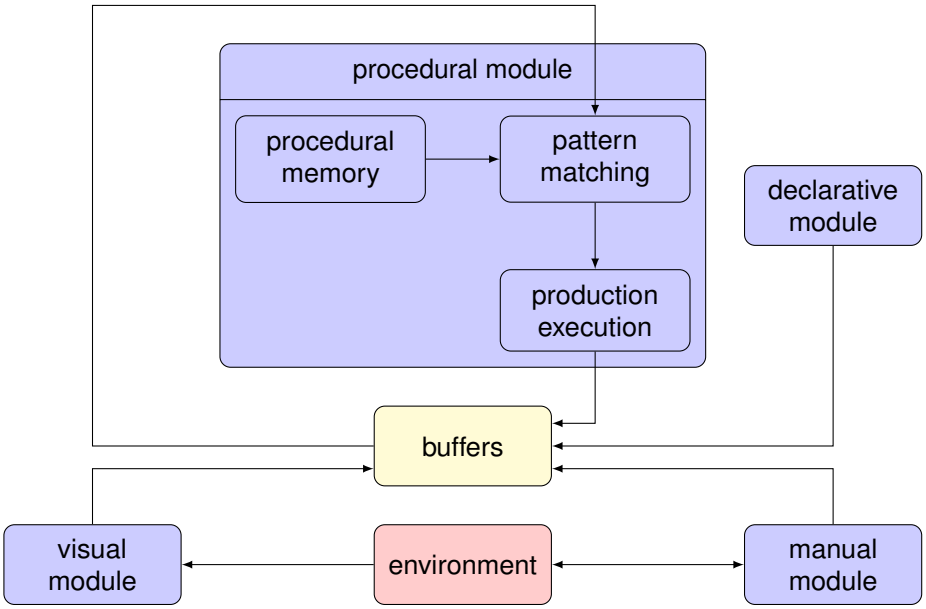


Figure 2.2: Modules and communication in ACT-R. The procedural module is the heart of the computational system and consists of a procedural memory containing all the production rules, a pattern matching unit and an execution unit. The pattern matching unit checks if the current content of the buffers match the condition of a rule in the procedural memory. If a rule matches, it is executed by the execution unit. The execution may lead to buffer changes or requests. Additionally, the other modules may change the content of their buffers by a request which may lead to new rules matching the current content of the buffer. The procedural module can only apply one rule at a time. [Abo]

Notice that in the figure each slot has an individual name. This is necessary to distinguish the connections of the chunks, otherwise the summands would be indistinguishable from the sum in the example.

Thus, chunks are defined by their name and the values of their slots. When talking about chunk descriptions, often the term *slot-value pairs* is used especially for partial chunk descriptions, i.e. descriptions which do not have a value for all possible slots. This simply refers to an arbitrary chunk that has the specified values in its slots (and the others are ignored).

Each chunk is associated with a chunk-type that determines the slots a chunk can have. A chunk-type consists of a name which serves as unique identifier chunk-types and a list of slots the chunks of this type offer. For example, the fact in figure 2.3 has the type *addition-fact*. All chunks of this type must provide the slots arg_1, arg_2 and *sum*. For the

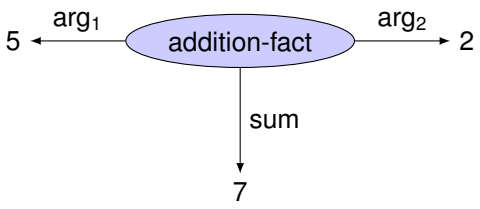


Figure 2.3: A chunk of type *addition-fact* and slots arg_1, arg_2 and *sum* which models the fact 5 + 2 = 7. The slots are connected to the primitive elements 5, 2 and 7. Chunks are illustrated as ellipses, whereas primitive elements are simple text. A simple arrow (⟶) signifies that the chunk at the start of the arrow has the value at the end of the arrow in the slot with the name of the label of the arrow.

chunk-types there is no upper limit of slots they can define. However, it is suggested to limit the number of slots to Miller's Number of 7 ± 2, for the reason of plausibility [SW07, p. 230].

Buffers

As mentioned before, modules communicate through buffers by putting a limited amount of information into their associated buffers. More precisely, each buffer can hold only *one chunk at a time*.

For example, the declarative module has the retrieval buffer associated with it, which can hold one specific declarative chunk. The declarative module can put chunks into the buffer that can be processed by the procedural module, which is described in the next section. Note that the chunks in the buffers are copies of the original chunks in the module. Hence, changes applied to a chunk in the buffer do not affect the original chunks in the requested module [Actb, unit 1, pp. 18 sq.].

2.1.3 Procedural Knowledge

Procedural Knowledge in ACT-R is formulated as a set of condition-action rules. Each rule defines in its condition-part the circumstances under which it can be applied. Those conditions refer to the current chunks in the respective buffer. The condition-part of a rule defines which kind of chunk with which slot values must be present in which buffer for the rule to fire. For example, one rule in the process of adding the numbers 5 and 2 could

have the conditions that there is a chunk of type *addition-fact* in the retrieval buffer with 5 and 2 in its argument-slots and specify certain actions if this is the case.

If the chunks in the buffers match all the conditions stated in a rule, it can be applied ("fired"), which leads its action-part to be performed. Possible actions are changes of some of the values in the chunk of a buffer, the clearing of a buffer or a buffer request, which leads the corresponding module to put a certain chunk into the requested buffer. Buffer requests are also stated in form of a (partial) chunk description[1] where chunk-type and slots encode the query of the request. So all the arguments and even the task which should be performed by the module are specified through a chunk representation. The actual semantics of a request depends on the module. For example, the declarative module will search a chunk that matches the chunk in the description of the request. One production rule, for instance, in the process of adding the numbers 5 and 2 could be, if the wrong *addition-fact* chunk is stored in the retrieval buffer, a retrieval request will be performed, which states that the declarative module should put a chunk into the retrieval buffer, that has 5 and 2 in its argument slots and is of type *addition-fact*. After the successful performance of the request, a chunk with 5 and 2 in its argument slots will be stored in the retrieval buffer, that also has a value for the sum. The actions are described in more detail in the following section.

Although the term *module* is used for the procedural system, it differs a lot from the other modules: In contrast to other modules, the procedural module has no own buffers, but can access the buffers of all the other modules. "It really is just a system of mapping cortical buffers to other cortical buffers" [And07, p. 54].

The procedural system can only fire one rule at once and it takes 50 ms for a rule to fire [And07, p. 54]. After firing the selected rule, the next recognize cycle starts and a suitable rule will be detected and caused to fire. During this time, other modules may perform requests triggered by the action of the last rule. Sometimes, rules have to wait for results of certain modules and they cannot fire before those results are available. Those two facts illustrate how the procedural module can become a serial bottleneck in the computation process.

Description of Procedural Actions

In this section, the actions that can be performed by a production rule are described in more detail than before. The information in this section has been taken from [Botb,

[1] A partial chunk description is just a chunk description that does not specify all slots that are available as defined in the chunk-type.

pp. 168 sqq.] and is – in this degree of detail – not part of the theory, but focuses more on the implementation to give a more detailed understanding of the concepts needed in chapter 4.

Buffer Modification: An in-place operation, that overwrites the slot values of a chunk in a buffer with the specified values in the action of the rule.

Buffer Request: A buffer request will cause the corresponding module to calculate some kind of result that will be placed into the requested buffer. The input values of this computation are given as chunks with a type and slot-value pairs specified in the request. For instance, the declarative module could search for a chunk that has the specified values in its slots.
The execution of the request is independent from the execution of production rules and after the request has been stated by the procedural module, it can begin with the next recognize-cycle while the requested module calculates its result.
Before the request is performed, the corresponding buffer will be cleared.

Buffer Clearing: If a buffer is cleared, its containing chunk will be placed into the declarative memory from where it can be retrieved later on. The clearing of a buffer with the implicit storing of the chunk in the declarative memory is an implementational detail which is very important for further considerations.

Chunks as Central Data Structure

As may have become obvious in the previous sections, chunks are the central data structures in ACT-R. They are used to model factual knowledge in the declarative memory, but are also used for communication: Requests are stated as chunks that encode the input of the request, for instance a chunk pattern for a result chunk the declarative memory should retrieve. The result of a request is a chunk placed into a buffer and even the procedural system, which technically is separated from the declarative knowledge, tries to match the chunks in the buffers in the condition part. Additionally, the action of a rule is specified by slot-value pairs that are basically just partial chunk descriptions.

Process of Rule Selection and Execution

As stated above, the procedural module can execute only one rule at a time. If no rule has been selected to fire – so no rule is in progress – the procedural module is *free* and therefore can select a matching rule according to the recognize-act-cycle as soon as it is available. If a rule has been selected, the module is *busy* and cannot choose another

rule to fire. As mentioned before, the module has to wait 50 ms between selection and firing of a rule. Then all in-place actions of the rule like modifying or clearing a buffer are performed. Afterwards, the requests are stated and the module is free. However, the requested modules most likely will take a certain time to perform the request. During this time the procedural module can select and fire the next matching rule nevertheless.

If at a certain time the procedural module is free, but there are no matching rules, the module waits until the system reaches a state where a rule matches. This is possible, since requests can take a certain time in which the procedural module is free and cannot find a matching rule. If the request has been performed, it usually causes a change of buffers. When the content of a buffer has changed, this could provoke the next rule to match and fire.

2.1.4 Goal Module

An essential part of human cognition is the ability to keep track of the current goal to achieve and to subordinate all actions to the goal [And+04, p. 1041]. For complex cognitive tasks, several rules have to be applied in series and intermediate results must be stored (without changing of the environment). Another important aspect is that complex tasks may consist of several subgoals which have to be achieved to accomplish the main goal. For instance, if one wants to add two multi-digit numbers, he would add the columns and remember the results as intermediate results in each step. In ACT-R, the goal module with its goal buffer is used for this purpose: It is able to keep track of the current goal, introduce subgoals and remember intermediate results in its buffer.

Working memory

The goal module and buffer are often referred to as *working memory* [And+04, p. 1041], but actually, as stated in [ARL96], it also can have another meaning: The usual definition in production systems is that everything which is present to the production rules and can match against them is part of the working memory. With this definition, all chunks in the buffers form the working memory.

In this work, the term *working memory* will be used in this second meaning, since it discusses the topic from a computer science view and the second definition is related to production rule systems. When talking about the content of the goal buffer, this will be remarked explicitly.

2.1.5 Other Modules

In figure 2.1 some more modules are shown. In the following, a short description of some of those modules is given.

The Outside World

Since human cognition is embodied, there must be a way to interact with the outside world to simulate human cognition in realistic experiments. Therefore, ACT-R offers *perceptual/motor modules* like the manual module for control of the hands, the visual module for perceiving and processing the visual field or the aural module to perceive sounds in the environment. Like with every other module, communication is achieved through the buffers of those modules. In the following, the visual module is described to exemplify the functionality of perceptual modules.

The Visual Module The visual system of ACT-R separates vision into two parts: visual location and visual objects [And+04, p. 1039]. There are two buffers for those purposes: the *visual-location* buffer and the *visual* buffer, which represents the visual objects [Actb, unit 2]. In the visual module it is not encoded how the light falls on the retina, but a more attentional approach has been chosen [And+04, p. 1039].

Requests to the *visual-location buffer* specify a series of constraints in form of slot-value pairs and the visual module puts a chunk representing the location of an object meeting those constraints into the visual-location buffer. Possible constraints are properties of objects like the color or the spatial location. The visual system can process such requests in parallel, i.e. that the whole visual field is processed massively parallel and, for example, the time of finding one green object surrounded by blue objects is constant, regardless of the number of blue objects. If more than one object meets the constraints, one of them will be chosen at random [And+04, p. 1039; And07, p. 68].

Requests to the *visual-object system* specify a visual location and the visual module will move its attention to that location, create a new chunk representing the object at that location and put that chunk into the visual buffer [Actb, unit 2, chapter 2.5.3].

These two kinds of requests to the visual module are summarized in table 2.1. The visual system and its capabilities are described in detail in [Actb, unit 2] where also the implementational details of the system are regarded.

Table 2.1: Requests to the visual module

	Visual location buffer	Visual (object) buffer
Input	Object Constraints	Visual location
Output	Visual location	Visual object

The Imaginal Module

The imaginal module, described in [Actb, unit 2], is capable of creating new chunks. This is useful, if for instance the visual module produces a lot of new information in sequence (like reading a sequence of letters), but the visual-object buffer can hold only one chunk at once. To solve this problem, all the information could be stored in the slots of the goal chunk. However, since a goal chunk with a large amount of slots seems to be unplausible[2] and the number of read instances would have to be known in advance due to the static chunk-type definition, a better way to deal with this problem is to create new knowledge elements.

This task can be achieved by using the imaginal module: On a request, it creates a new chunk of the type and with the slots stated in the request and puts it into its *imaginal buffer*. Since the chunk in a buffer is stored in the declarative memory when the buffer is cleared,[3] an unlimited amount of data can be produced and remembered by stating retrieval requests later on.

It is important to mention that it takes the imaginal module .2 ms to create a chunk. This amount of time is constant, but can be set by the modeler. Additionally, the imaginal module can only produce one chunk at a time.[4]

2.1.6 Example: Counting

The first ACT-R example model deals with the process of counting. This model relies on count facts a person has learned, e.g. “the number after 2 is 3”. To model this in ACT-R, a chunk-type for those facts has to be defined: A chunk of type *count-fact* has the slots *first* and *second*. The chunks in figure 2.4 of this type model the facts that 3 is the successor of 2 and 4 is the successor of 3.

[2] As described in section 2.1.2, one should stick to 7 ± 2 slots for each chunk.
[3] see section 2.1.3
[4] like every module can only handle one request at a time

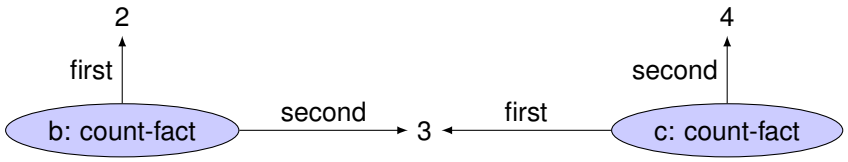


Figure 2.4: Two count facts with names *b* and *c* which model the counting chain 2, 3, 4.

The next step is to define the goal chunk stored in the goal buffer. In this chunk it somehow has to be encoded that the current goal is to count. This can be modeled in ACT-R by the chunk-type. To track the current number in the counting process as an intermediate result, the goal chunk could have a slot which always holds the current number that has been counted to. This leads to a goal chunk as illustrated in figure 2.5, where the current number is 2.

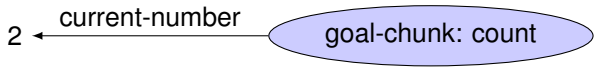


Figure 2.5: The goal-chunk of type *count* with the current number 2.

In this example we assume that the model starts with this goal chunk in the goal buffer and the first count fact has been retrieved:

goal buffer: *goal-chunk* of type *count*
current-number 2

retrieval buffer: *b* of type *count-fact*
first 2
second 3

This notation indicates that the goal buffer holds a chunk with the name *goal-chunk* of the type *count*, which has the slot *current-number* with the value 2 (the same is valid analogously for the retrieval buffer). Now the rule to implement counting can be defined as:

count-rule
IF the goal is to count, the current number is n
AND the retrieval buffer holds a chunk of type *count-fact* with the *first* value n and the *second* value m
THEN set the current number in the goal to m
AND send a retrieval request for a chunk that has m in its *first* slot

The rule matches the initial state: In the goal there is a chunk of type *count*, that indicates that the goal is to count, the current number n is 2. In the retrieval buffer, there is a *count-fact* with the *first* number $n = 2$ and the *second* number $m = 3$.

After applying this rule, the current number will be 3 and the next fact in the retrieval buffer will be a *count-fact* with the *first* value 3 and a value in the *second* slot, which will be the next number in the counting process. This illustrates the functionality of module requests: In the request a (potentially partial) chunk definition is stated and the corresponding module puts the result of the request in a fully defined chunk of some appropriate type into its buffer. For the declarative module, the request specifies the chunk-type and some slot values which describe the chunk that the module should be looking for. The result is a fully described chunk of that type with values for all slots, that describe an actual chunk from the declarative memory. As mentioned before, the chunks in the buffers are copies of the chunks of the requested modules. The count-rule will be applicable as long as there are *count-facts* in the declarative memory.

In this example, the rules have been defined in a very informal way. In the following chapters which deal with implementation, a formalization of such rules will be discussed, that defines clearly what kinds of rules are allowed. Furthermore, it introduces a formalism to describe such rules uniquely and less verbosely. The following chapters will refer to this example and refine it gradually.

The example also uses the concept of *variables*, which will be introduced more formally in chapter 4 when talking about implementation. Variables allow rule conditions to act like patterns that can match various system states instead of defining a rule for each state, since computation is the same regardless of the actual values in the buffers.

2.1.7 Serial and Parallel Aspects of ACT-R

In the previous sections there were some remarks on the serial and parallel aspects of ACT-R. According to [And07, p. 68], four types of parallelism and seriality can be distinguished:

Within-Module Parallelism: As mentioned above, one module is able to explore a big amount of data in parallel. For example, the visual module can inspect the whole visual field or the declarative module performs a massively parallel search over all chunks.

Within-Module Seriality: Since modules have to communicate, they have a limited amount of buffers and each of those buffers can only hold one chunk. For example, the visual module only can concentrate on one single visual object at one visual

location, the declarative module only can have one single concept present, the production system can fire only one rule at a time, ...

Between-Module Parallelism: Modules are independent of each other and their computations can be performed in parallel.

Between-Module Seriality: However, if it comes to communication, everything must be exchanged via the procedural module that has access to all the buffers. Sometimes, the production system has to wait for a module to finish, since the next computation relies on this information. So, modules may have to wait for another module to finish its computation before they can start with theirs triggered by a production rule that states a request to those modules.

The procedural module is the central serial bottleneck in the system, since the whole communication between modules is going through the production system and the whole computation process is controlled there. The fact that only one rule can fire at a time leads to a serial overall computation. Another serial aspect is that some computations need to wait for the results of a module request. If no other rule matches in the time while the request is performed, the whole system has to wait for this calculation to finish. After the request, the module puts the result in its buffer and the rule waiting for this result can fire and computation is continued.

2.2 Subsymbolic layer

The previously discussed aspects of the ACT-R theory are part of the so-called symbolic layer. This layer only describes discrete knowledge structures without dealing with more complex questions like:

- How long does it take to retrieve a certain chunk?
- Forgetting of chunks
- If more than one rule matches, which one will be taken?

Therefore, ACT-R provides a subsymbolic layer that introduces “neural-like activation processes that determine the availability of [...] symbolic structures” [AS00].

2.2.1 Activation of Chunks

The activation A_i of a chunk i is a numerical value that determines if and how fast a chunk can be retrieved by the declarative module. Suppose there are, for example, two

chunks that encode addition facts for the same two arguments (let them be 5 and 2), but with different sums (6 and 7). This could be the case, if, e.g., a child learned the wrong fact about the sum of 5 and 2. When stating a module request for an addition fact that encodes the sum of 5 and 2, somehow one of the two chunks has to be chosen by a certain method, since they are both matching the request. This is determined by the activation of the chunks: The chunk with the higher activation will be chosen.

Additionally, a very low chunk activation can prevent a chunk from being retrieved: If the activation A_i is less than a certain *threshold* τ, the chunk i cannot be found. At last, activation determines also how fast a chunk is being retrieved: The higher the activation, the shorter the retrieval time.

Base-Level Activation

The activation A_i of a chunk i is defined as:

$$A_i = B_i + \Gamma \tag{2.1}$$

where B_i is the *base-level activation* of the chunk i. Γ is a context component that will be described later on. Equation (2.1) is a simplified variant of the *Activation Equation* which is completed in equation (2.3).

The base-level activation is a value associated with each chunk. It depends on how often a chunk has been practiced and when this practice has been performed. A chunk is *practiced* when it is retrieved. Hence, B_i of chunk i is defined as:

$$B_i = \ln\left(\sum_{j=1}^{n} t_j^{-d}\right) \tag{2.2}$$

where t_j is the time since the jth practice, n the number of overall practices of the chunk and d is the decay rate that describes how fast the base-level activation decreases if a chunk has not been practiced (how fast a chunk will be forgotten). Usually, d is set to 0.5 [And+04, p. 1042]. Equation (2.2) is called *Base-Level Learning Equation* as it defines the adaptive learning process of the base-level value.

This equation is the result of a rational analysis by Anderson and Schooler. It reflects the log odds that a chunk will reappear depending on when it has appeared in the past [TLA06, p. 33]. This analysis led to the *power law of practice* [And+04, p. 1042]. In [AS00, pp. 8–11] equation (2.2) is motivated in more detail by describing the power law

of learning/practice, the power law of forgetting and the multiplicative effect of practice and retention with some data. Shortly, it states that if a particular fact is practiced, there is an improvement of performance which corresponds to a power law. At the same time, performance degrades with time corresponding to a power law. Additionally, they state that if a fact has been practiced a lot, it will not be forgotten for a longer time.

Activation Spreading

In ACT-R, the basic idea of activation is that it consists of two parts: The base-level component described above, and a context component. Every chunk in the current context has a certain amount of activation that can spread over the declarative memory and enhance activation of other chunks that are somehow connected to those chunks in the context. The activation equation (2.1) is extended as follows:

$$A_i = B_i + \sum_{j \in C} W_j S_{ji} + \varepsilon \tag{2.3}$$

where W_j the *attentional weighting* of chunk j, S_{ji} the *associative strength* from chunk j to chunk i and C is the *current context*, usually defined as the set of all chunks that are in a buffer [And+04, p. 1042; TLA06, p. 33; Actb, unit 5]. The chunks in the current context are often referred to as *sources of activation*. ε is a noise value "generated according to a logistic distribution" [Actb, unit 4, p. 4]. Figure 2.6 illustrates the addition-fact 5 + 2 = 7 with the corresponding quantities introduced in the last equation.

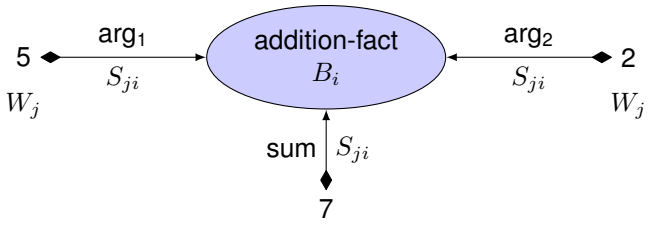


Figure 2.6: The *addition-fact* chunk from figure 2.3 with subsymbolic quantities. The special arrows (◆→) in this figure indicate the direction of the activation spreading: The chunks j are at the start (◆—) of the arrow, the chunk which receives the activation is at the end (—→). The elements 5 and 2 are supposed to be in the working memory and therefore have a value for the attentional weighting W_j. The chunk itself has a base-level activation B_i. The S_{ji} values are the associative strengths from the elements to the chunk. [And+04, fig. 5]

The values for W_j determine how much activation can spread from a single source of activation in the current context. A source of activation is a chunk in the goal buffer or

in all buffers, depending on the version of the ACT-R theory [And+04, p. 1042; TLA06, p. 33; Actb, unit 5, p. 1]. To limit the total amount of source of activation, W_j is set to $\frac{1}{n}$, where n is the number of sources of activation. With this equation, the total amount of activation that can spread over declarative memory is limited, since the more chunks are in the current context, the less important a particular connection between a chunk from the context with a chunk from declarative memory becomes.

Strength of Association and Fan effect In equation (2.3) the strength of association S_{ji} from a chunk j to a chunk i is used to determine the activation of a chunk i. In the ACT-R theory, the value of S_{ji} is determined by the following rule: If chunk j is not a value in the slots of chunk i and $j \neq i$, then S_{ji} is set to 0. Otherwise S_{ji} is set to:

$$S_{ji} = S - \ln(\mathrm{fan}_j) \tag{2.4}$$

where fan_j is the number of facts associated to term j [AS00, p. 1042]. In more detail: "fan_j is the number of chunks in declarative memory in which j is the value of a slot plus one for chunk j being associated with itself" [Actb, unit 5, p. 2]. Hence, equation (2.4) states that the associative strength from chunk j to i decreases the more facts are associated to j.

This is due to the *fan effect:* The more facts a person studies about a certain concept, the more time he or she needs to retrieve a particular fact of that concept [AR99, p. 186]. This has been demonstrated in an experiment presented in [AR99], where every participant studied facts about persons and locations like for example:

- A hippie is in the park.
- A hippie is in the church.
- A captain is in the bank.

For every person the participants studied either one, two or three facts. Afterwards, they were asked to identify targets, that are sentences they studied, and foils, i.e. sentences constructed from the same persons and locations, but that were not in the original set of sentences. "The term *fan* refers to the number of facts associated with a particular concept" [AR99, p. 186]. Figure 2.7 represents an example chunk network of the studied sentences with their fan, S_{ji} and B_i values.

The result of the experiment was, that the more facts are associated with a certain concept, the higher the retrieval time for a particular fact about that concept was. In the ACT-R theory, this result has been integrated in the calculation of the strengths of association: In equation (2.4) the associative strength decreases with the number of

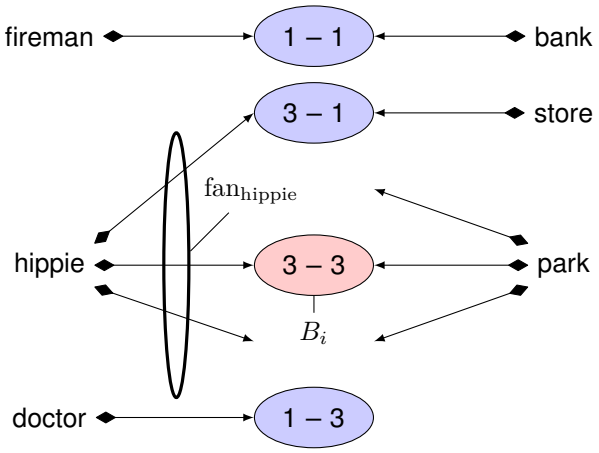


Figure 2.7: The ellipses represent chunks which encode one fact like “a hippie is in the park” (red highlighted chunk i). The words are primitive elements. The special arrows show the direction of activation spreading from an element to a chunk. The numbers in the chunks signify the fan of the associated element, e.g. the fan of the element *hippie* is 3. The associative strength of the element hippie to the highlighted chunk i is $S_{\text{hippie},i} = S - \ln(\text{fan}_{\text{hippie}}) = S - \ln(3)$. [And+04, fig. 6]

associated elements. The value S is a model-dependent constant, but in many models estimated about 2 [And+04, p. 1042]. Modelers should take notice of setting S high enough that all associative strengths in the model are positive [Actb, unit 5, p. 3]. Figure 2.8 illustrates the activation spreading process with respect to the associative strengths.

Latency of Retrieval

As mentioned before, the activation of a chunk affects if the chunk can be retrieved (depending on a threshold and the activation values of the other matching chunks). In addition, activation also has an effect on the time it takes to retrieve a particular chunk. This time is called *latency* and is calculated as follows:

$$T_i = F \cdot \mathrm{e}^{-A_i} \tag{2.5}$$

where T_i is the *latency* of retrieving chunk i, A_i the activation of this chunk, as defined in equation (2.3), and F the *latency factor*, which is usually estimated to be

$$F \approx 0.35\mathrm{e}^{\tau} \tag{2.6}$$

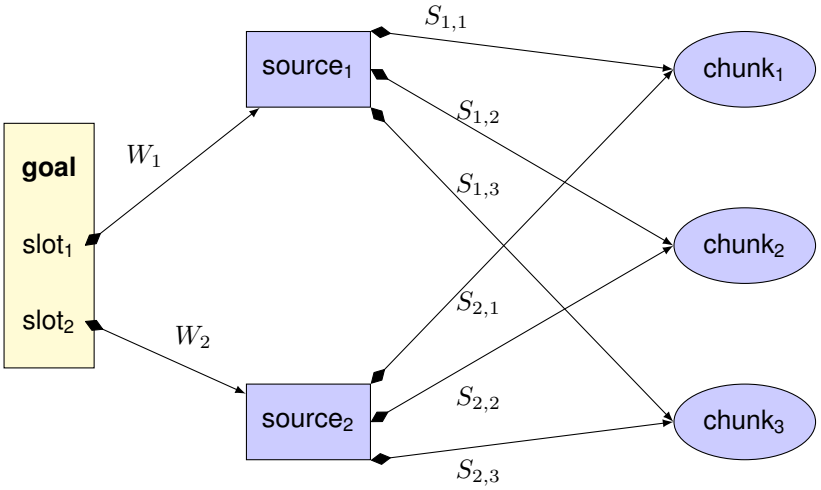


Figure 2.8: The goal buffer holds a chunk with two slots wich therefore are sources of activation. The special arrows signify the spreading of activation from those sources to the chunks in the declarative memory. The attentional weighting values W_1 and W_2 determine how much of the overall amount of activation spreads from each single source. If there are only those two sources, the W_i are set to $\frac{1}{2}$. From each source the activation may spread to the chunks in the declarative memory. The amount of spreading activation depends on the connection between the source chunk and the chunk in the memory. If the source chunk does not appear in the slots of a chunk, the value of the associative strength from the source j to the chunk i is $S_{ji} = 0$. Otherwise, if the source appears in the slots of a chunk, the associative strength depends on the fan value of the source chunk – the higher the fan of the source, the lower the associative strength. I.e. if the source chunk appears in the slots of many chunks, the value of the associative strength will be lower than if the connection is exclusive. [Actb, unit 5]

where τ is the retrieval threshold as mentioned in section 2.2.1, but F can also be set individually by the modeller. Nevertheless, in [And+04, p. 1042] it is stated that the relationship of the retrieval threshold and the latency factor in equation (2.5) seems to be suitable for a lot of models.

2.2.2 Production Utility

For the production system, there is the subsymbolic concept of *production utilities* to deal with competing strategies. For instance, if a child learns to add numbers, it may have learned different strategies to compute the result: One could be counting with the fingers

and the other could be just retrieving a fact for the addition from declarative memory. If the child has the goal to add two numbers, it somehow has to decide which strategy to choose, since both of them match the context.

In ACT-R, the production utility is a number attached to each production rule in the system. Just like with activation of chunks, the production rule with the highest utility will be chosen, if there is more than one matching rule. The utilities can be set statically by the modeler, but they can also be learned automatically by practice.

In the current version of the ACT-R theory, a reinforcement learning rule based on the Rescorla-Wagner learning rule [RW72] has been introduced. The utility U_i of a production rule i is defined as:

$$U_i(n) = U_i(n-1) + \alpha\left(R_i(n) - U_i(n-1)\right) \tag{2.7}$$

where α is the *learning rate* which is usually set around .2 and $R_i(n)$ is the reward the production rule i receives at its n^{th} application [And07, pp. 160–161]. This leads the utility of a production rule being gradually adjusted to the average reward the rule receives [Actb, pp. 6–7].

Usually, rewards can occur at any time and it is not clear which production rule will be strengthened by the reward. In *How can the human mind occur in the physical universe?* Anderson describes an example, where a monkey receives a squirt of juice a second after he presses a button [And07, p. 161]. Now, the question is, which production rule is rewarded, since between the reward and the firing of a rule there always is a break. In ACT-R, every production that has been fired since the last reward event will be rewarded, but the more time lies between the reward and the firing of the rule, the less rewarded the particular rule gets. The reward for a rule is defined as the amount of external reward minus the the time from the rule to the reward. This implies, that the reward has to be measured in units of time, e.g., how much time is a monkey willing to spend to get a squirt of juice? [And07, p. 161]

In implementations of ACT-R, rewards can be triggered by the user at any time or can be associated with special production rules that model the successful achievement of a goal (they check if the current state is a wanted state and then trigger a reward, so every rule that has led to the successful state will be rewarded). It is important to mention that by definition, rules can also get a negative reward if their selection was too long ago. If one wants to penalize all rules since the last reward, a rule that distributes a reward of 0 can be triggered, which leads all rules applied before being rewarded with a negative amount of reward, so their utility value decreases and they become less likely to fire [Actb, unit 6, p. 8].

2.3 Learning

Learning in ACT-R can be divided into four categories depending on the involvement of the symbolic or subsymbolic layer and the declarative or the procedural module. Table 2.2 names the four types that are described in this section.

Table 2.2: ACT-R's Taxonomy of Learning [And07, pp. 92–95]

	Declarative	Procedural
Symbolic	Fact learning	Skill acquisition
Subsymbolic	Strengthening	Conditioning

2.3.1 Symbolic Layer

Symbolic learning somehow influences the objects of the symbolic layer – i.e. chunks and production rules – in a way that new objects are created or objects are merged. Those learning possibilities are described very briefly in the following, but are not yet part of the implementation discussed in this work. However, it may become part of the implementation in future work.

Fact Learning

Fact learning is the creation of new chunks. In ACT-R, the imaginal module is capable of this as described in section 2.1.5 on page 16. The chunks usually are created because of external input from the environment, but it is also possible to create a chunk that holds the result of a production rule application. Furthermore, due to buffer modification chunks can be altered and may be stored in the declarative module by buffer clearing which basically creates a new chunk. In this case, the input may come from the external world or is the result of of a production rule application [Whi, p. 7].

Skill acquisition

In [TL03], Taatgen and Lee present the method of *production rule compilation* as a method of skill acquisition, i.e. the creation of new production rules. "Production compilation

combines two mechanisms [...] – proceduralization and composition – into a single mechanism" [TL03, p. 62]. *Proceduralization* is based on the idea, that some rules containing variables are applied very often on the same values for the variables. If this is the case, a new rule can be introduced, which replaces the variables with ground values [Whi, p. 7]. *Composition* takes two production rules which are performed in sequence very often and creates a new rule which performs all the steps of those rules at once. Since each application of a rule takes a certain time, this can speed up the computation process [Whi, p. 7].

2.3.2 Subsymbolic Layer

The concepts introduced in section 2.2 are a kind of learning: The practice of particular facts strengthens the chunks encoding this fact and chunks that are not practiced are forgotten over time.[5] Additionally, associative weights are learned from the current context. These processes adapt to the problems a particular human mind is confronted with and thereby work autonomously.

The same is valid for production rules: Over time, the experience tells us, which strategies might be successful in certain situations and which are not. This process is also called *conditioning* and described in equation (2.7).

2.4 Experiment Environment

In experiments conducted with humans, the test subjects are very often confronted with visual or aural stimuli, for instance in the fan effect experiment described in section 2.2.1 on page 22 a permutation of sentences over some objects has been shown and people had to remember the combinations. ACT-R provides an environment, where visual stimuli, mouse gestures or clicks, window openings etc. can be produced or measured respectively, so experiments can be run with both humans and the implemented model using the same environment. The model is then able to perform clicks or write text on the same graphical user interface as the humans are confronted with. However, this part of ACT-R will not be subject of this particular work, which examines the fundamental concept of the ACT-R theory first.

[5] This is the concept of base-level learning as described in section 2.2.1

3 Constraint Handling Rules

This chapter only gives a very brief introduction to Constraint Handling Rules. A much more comprehensive introduction can be found in the book *Constraint Handling Rules* by Frühwirth [Frü09], which gives a very detailed definition of the exact semantics of the language and of the analysis methods as well as practical examples. Additionally, there are a lot of tutorials, slides and exercises that can be found on *The CHR Homepage* [Chr]. This short summary is based on [Frü09].

Constraint Handling Rules (CHR) is a high-level declarative programming language originally designed to write constraint-solvers [Sne+10, p. 2]. CHR programs are defined by a sequence of rules that operate on a *constraint store*, which is a multiset of constraints (built from predicate symbols and arguments). I.e., a constraint `c/n` with arity `n` is written as `c`$(t_1, \ldots, t_n)$, where `c` is a predicate symbol and the t_i – the arguments of the constraint – are logical terms. Constraints in the store can be *ground*, i.e. they do not contain any variables, but they also may contain unbound variables. CHR is usually embedded in a *host language*, which provides so-called *built-in* constraints. Such built-in constraints are constraints that are implemented in the host language. In this work, Prolog is assumed to be the host language. There are three types of rules:

Listing 3.1: CHR rule types

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

Rules consist of a *head* $\mathtt{H}_*$, which is a conjunction of at least one constraint, an optional *guard* `G`, which is a conjunction of built-in constraints, and a *body* `B`, which is a conjunction of both built-in and CHR constraints. Optionally, the rules can be given names followed by the symbol `@`.

At the beginning of a program, the user provides an initial constraint store, which is also called *query* or *goal* and the CHR program then begins to operate on this initial store by applying rules.

A rule can be applied on the constraint store, if there are constraints in the store that match the head of the rule, i.e. the constraints in the store are "an instance of the head, [...so] the head serves as pattern" [Frü09, p. 11]. Note that in the matching process no variables of the query are bound since CHR is a committed-choice language, so rule applications cannot be made undone by backtracking. If matching constraints have been found, the guard is checked and if it holds, the rule is applied. The result of the rule application is dependent on the type of the rule:

Simplification rules (as in line 1) remove the constraints which match the head $\mathtt{H_r}$ (the *"heads removed"*) from the store and add the constraints in the body `B` to the store.

Propagation rules (as in line 2) add the constraints in the body `B` to the store and keep the matching constraints from the head $\mathtt{H_k}$ (the *"heads kept"*).

Simpagation rules (as in line 3) are a mix of the first two rule types: They add the constraints in the body `B` to the store, keep the matching constraints from the head $\mathtt{H_k}$ and remove the constraints from the head $\mathtt{H_r}$. In general, simplification and propagation rules can be expressed by simpagation rules where one side of the head is empty, i.e. only contains the constraint `true`.

Example 3.1 (Minimum)**.** *This simple example has been taken from [Frü09, pp. 19 sqq.] and describes a CHR program that computes the minimum among the numbers n_i given as multiset of numbers in the query* `min(`n_1`), ... , min(`n_k`)`*. The* `min/1` *constraint is interpeted such that its containing number is a minimum candidate. The minimum computation is achieved by specifying one simpagation rule:*

Listing 3.2: Minimum program

```
1 min(N) \ min(M) <=> N=<M | true.
```

For every pair of numbers in the store, the rule removes the greater one and keeps the smaller one. If this rule is applied to exhaustion, only one constraint is left – the minimum. Then the rule is not applicable any more, since it lacks a partner constraint for the single `min` *constraint to match the head.*

Note that in pure CHR no assumptions on the order of the application of the rules and the involved constraints are made. One possible derivation of the result could be (the constraints the rule is applied to are underlined in each step):

Listing 3.3: Sample derivation of the minimum example

```
1 min(3), min(1), min(4), min(0), min(2)
2 min(1), min(4), min(0), min(2)
3 min(1), min(0), min(2)
```

```
4 min(0), min(2)
5 min(0)
```

Example 3.2 (Matching)**.** *This example tries to clarify the matching process and is taken from [Frü09, p. 12]. The following rules are added into individual, separated programs and some queries are tested in those programs.*

```
1 p(a) <=> true.
2 p(X) <=> true.
3 p(X) <=> X=a.
4 p(X) <=> X=a | true.
5 p(X) <=> X==a | true.
```

The query `?- p(a)` *matches all the rules, so the result will always be an empty constraint store, indicated by* `true`*.*

For the query `?- p(b)` *the first rule does not match and* `p(b)` *is added to the constraint store. The second rule is applicable, since* `p(X)` *is a pattern for* `p(a)`*. For the third rule, the result of the computation is* `false`*, because the rule is applicable, but the unification* `a=b` *in the body will fail. Since CHR is a committed-choice language, the rule selection will not be withdrawn by backtracking. The last two rules are not applicable, since the guard fails.*

The query `?- p(Y)` *does not match the first rule, since* `p(a)` *is not a pattern* `p(Y)` *– the goal is more general than the head of the rule and no bindings of the constraints in the goal will be performed for the matching. The second rule does match, because the* `X` *from the head of the rule can be bound to* `Y` *from the goal. The result of the application of the third rule is* `Y=a`*, because the rule does match and leads to a binding of* `Y` *to* `a` *in the body. The last two rules do not fire, because their guards fail. If* `Y` *has been bound to* `a` *explicitly before (e.g. by another rule), the rule could fire.*

There are various definitions of the operational semantics (i.e. the behaviour of CHR programs) for different purposes and there may come more in the future [Frü09, p. 11]. In this work, the implementation from the K.U. Leuven as shipped with SWI-Prolog is used [Swi]. This version implements the so-called *refined operational semantics*. Constraints in the goal are processed from left to right. When entering the constraint store, a CHR constraint becomes *active*, i.e. each rule which contains the active constraint in the head is checked for applicability. The rule applicability is checked in textual order of the rules (top-down). The first rule that matches is fired. If the active constraint is removed by

the rule application, the next constraint from the body will be added and become active. Otherwise, if the active constraint is kept, the next rule below the first matching rule is checked for applicability, etc. If no rule matches, the constraint becomes passive and is actually put to the constraint store, where it waits to become the partner constraint in a further matching rule and the next constraint from the query is added. A passive constraint becomes active again, if its containing variables are bound.

If a rule fires, its body is processed from left to right and behaves like a procedure call: It will add its constraints one after another (and they will become active sequentially). When all constraints from the body have been added and no constraint is active anymore, the next constraint from the query is added. If no constraints are left in the query, the program terminates and the final constraint store is printed.

In the implementation of ACT-R, it was necessary at some points to rely on the order of rule applications from top to bottom. This is very common, for instance, if in an advanced version of the minimum computation in example 3.1 the process should be triggered by a `get_min/1` constraint that also gets the result bound to its argument. I.e., after the complete computation of the query in listing 3.3 triggered by `get_min(Min)`, the variable `Min` should be bound to `Min=0`, the minimal number in the query. This can be achieved as follows:

Listing 3.4: Minimum program with trigger

```
1 get_min(_), min(N) \ min(M) <=> N=<M | true.
2 get_min(Min), min(N) <=> Min=N.
```

Obviously, the result of this program depends on the rule application order: The second rule is applicable as soon as the minimum computation has been triggered by `get_min` and one single minimum candidate `min(N)`. After the application of the second rule, the first one will not be applicable anymore, because the `get_min/1` constraint is removed from the store. Hence, it must be ensured that this rule is not applied before the first rule has been applied to exhaustion. This is achieved by the refined operational semantics. The program computes the minimum of all numbers that are in the store at the time the `get_min/1` trigger constraint is introduced. If there are added new minimum candidates afterwards, they will not be respected in this particular minimum computation. However, if later on a new minimum computation is triggered, all those minimum constraints will be part of the computation.

4 Implementation of ACT-R in CHR

After the overview of the ACT-R theory in chapter 2, this chapter gives a more detailed survey and presents a possible implementation of the described concepts of the ACT-R theory in CHR, inspecting some implementational details of the theory and formalizing some basic concepts. Note that this work only regards the fundamental concepts of ACT-R and abstracts from some details. For example, the experiment environment as described in section 2.4 is ignored.

For the implementation, some special cases and details that are not exactly defined in theory have to be considered. Hence, some concepts of the theory that are implemented in this work are formalized first. The implementation in form of CHR rules sticks to those formalisms and is often very similar to them.

Additionally, the implementation is described incrementally, i.e. first, a very minimal subset of ACT-R is presented that will be refined gradually with the progress of this chapter. In the end, an overview of the actual implementation as a result of this work is given. Some of the definitions in this chapter result directly from the theory, some of them needed a further analysis of the official ACT-R 6.0 Reference Manual [Botb] or the tutorials [Actb].

4.1 Declarative and Procedural Knowledge

The basic idea of the implementation is to represent declarative knowledge, working memory etc. as constraints and to translate the ACT-R production rules to CHR rules. This approach leads to a very compact and direct translation of ACT-R models to Constraint Handling Rules.

In addition to the production rules there will be rules that implement parts of the framework of ACT-R, for example rules that implement basic chunk operations like modifying or deleting chunks from declarative memory or a buffer. Those parts of the system are described as well as the central data structures and the translation.

First, a formalization of declarative knowledge in form of chunk networks and their implementation in CHR is given. Then, the working memory – also referred to as the buffer system – is explored and the implementation is discussed. After those definitions of the basic data structures of ACT-R, the procedural system is described including the translation of ACT-R production rules to CHR rules using the previously defined data structures. Furthermore, the reproduction of ACT-R's modular architecture is shown and the implementation of the declarative module is presented. After this overview of the basic concepts of ACT-R, the description goes into more detail about timing issues and the subsymbolic layer.

4.2 Chunk Stores

Since chunks are the central data-structure of ACT-R used for representation of declarative knowledge, to exchange information between modules and to state requests, this section first deals with this essential part of ACT-R.

4.2.1 Formal Representation of Chunks

In multiple parts of ACT-R it is necessary to store chunks and then operate on them. Hence, the abstract data structure of such a chunk store is defined. Since chunk stores have been referred to as networks in the previous chapters, the general idea of this definition of a chunk store is based upon a relation that represents such a network.

Definition 4.1 (chunk-store)**.** *A* chunk-store Σ *is a tuple* $(C, E, \mathcal{T}, \mathrm{HasSlot}, \mathrm{Isa})$*, where* C *is a set of chunks and* E *a set of primitive elements both identified by unique names, with* $C \cap E = \emptyset$*. The* values *of* Σ *are defined by the set* $V = C \cup E \cup \{\texttt{nil}\}$*.* $\mathcal{T}$ *is a set of chunk-types. A chunk-type* $T = (t, S) \in \mathcal{T}$ *is a tuple with a* unique[1] *type name* t *and a set of slots* S*. The set of all chunk type names is* $\mathcal{T}_{name}$ *and the set of all slot names is* $\mathcal{S}$*.*

$\mathrm{HasSlot} \subseteq C \times \mathcal{S} \times V$ *and* $\mathrm{Isa} \subseteq C \times \mathcal{T}$ *are relations and are defined as follows:*

- $c \ \mathrm{Isa} \ T \Leftrightarrow$ *chunk* c *is of type* T*.*
- $(c, s, v) \in \mathrm{HasSlot} \Leftrightarrow v$ *is the value of slot* s *of* c*. This can also be written as* $c \xrightarrow{s} v$ *and is spoken "*c *is connected to* v*" or "*v *is in the slot* s *of* c*".*

[1] $\forall (t, S), (t', S') \in \mathcal{T} : t = t' \Rightarrow S = S'$

The Isa *relation has to be right-unique and left-total, so each chunk has to have exactly one type. The following functions are defined:*

$$slots : C \to \mathcal{S} \times V$$
$$slots(c) = \{\ (s, v)\ \mid\ (c, s, v) \in HasSlot\ \}\ \textit{and}$$
$$slots : \mathcal{T}_{name} \to \mathcal{S}$$
$$slots(t) = \begin{cases} S & for\ (t, S) \in \mathcal{T} \\ \emptyset & otherwise \end{cases}$$

A chunk-store is type-consistent*, iff* $\forall(c, (t, S)) \in \mathrm{Isa} : \forall s \in S\ \exists!(c, s, v) \in \mathrm{HasSlot}$*. So every chunk must have exactly one value for each slot of its type and only describe slots of its type. Empty slots are represented by the value* `nil`*. Since every chunk has exactly one type, this is valid for all chunks in the store.*

Definition 4.2 (abstract methods of a chunk store)**.** *The following methods can be defined over a chunk store* $\Sigma = (C, E, \mathcal{T}, \mathrm{HasSlot}, \mathrm{Isa})$*:*

chunk-type(name slot₁ ...slotₙ) *is a method for defining a new type* $T = (\mathtt{name}, \{\mathtt{slot_1}, \ldots, \mathtt{slot_n}\})$ *which is added to the store, i.e.* $\mathcal{T}' = \mathcal{T} \cup \{T\}$*.*

add-chunk(name isa type slot₁ val₁ ...slotₙ valₙ) *adds a chunk specified by a name and a list of slot-value pairs to the store, i.e.* $C' = C \cup \{\mathtt{name}\}$*,* $\mathrm{Isa}' = \mathrm{Isa}\ \cup\ (\mathtt{name}, (\mathtt{type}, slots(\mathtt{type}))$ *and* $\mathrm{HasSlot}' = \bigcup_{i=1}^{n} (\mathtt{name}, \mathtt{slot_i}, \mathtt{val_i})\ \cup\ \mathrm{HasSlot}$*. Note that due to the expansion of* C*, the condition that* C *and* E *have to be disjoint may be violated. To fix this violation, the element can be removed from* E*:* $E' = (E \cup C) - (E \cap C)$*.*
Additionally, a valid mechanism to restore type-consistency may be introduced: It might happen that not all slots are specified in the call of the `add-chunk` *method. Since it is claimed by the definition of* $\mathrm{HasSlot}$ *that for all slots* s *of a chunk* c *there must be a* $(c, s, v) \in \mathrm{HasSlot}$*, in implementations the unspecified slots are initialized as empty slots, represented by the empty value* `nil`*. Furthermore, slots specified in the call of the method that are not a member of the chunk's type should cause an error to preserve type-consistency.*

alter-slot(name slot₁ val₁ ...slotₙ valₙ) *changes the values of the specified slots of a chunk identified by its name to new values. Only existing slots can be altered, but the list of slot-value pairs may be a partial chunk-description. The values of the slots not in the list remain at their old values.*

remove-chunk(name) *removes the chunk with the given name from* C *and all of its occurrences in* Isa *and* $\mathrm{HasSlot}$*.*

return-chunk(name) *gets a chunk name as input and returns a chunk specification, i.e. the name, type and all slot-value pairs of this chunk in the store.*

Example 4.1. *The addition-fact chunk in figure 2.3 and its chunk-type are defined as follows:*

```
chunk-type(addition-fact arg1 arg2 sum)
add-chunk(a isa addition-fact arg1 5 arg2 2 sum 7)
```

This leads to the following chunk-store:

$$\begin{aligned}(\{a\}, \{2, 5, 7\},\\ \{(addition\text{-}fact, \{arg_1, arg_2, sum\})\},\\ \{(a, arg_1, 5), (a, arg_2, 2), (a, sum, 7)\},\\ \{(a, addition\text{-}fact)\}).\end{aligned}$$

Type consistency is checked as follows:

$$\begin{aligned}slots(a) &= \{(arg_1, 5), (arg_2, 2), (sum, 7)\}\\ slots(addition\text{-}fact) &= \{arg_1, arg_2, sum\}\end{aligned}$$

Since the chunk a defines values for all of the possible slots of the type $addition\text{-}fact$ and only those slots, the store is type-consistent.

4.2.2 Representation of Chunks in CHR

Declarative knowledge is represented as a network of chunks, defined by the two relations Isa, specifying the belonging of a chunk to a type, and $\mathrm{HasSlot}$, specifying the slot-value pairs of a chunk. Those relations can be translated directly into CHR by defining the following constraints representing the relations and sets:

```
:- chr_constraint chunk_type(+).
% chunk_type(ChunkTypeName)

:- chr_constraint chunk_type_has_slot(+,+).
% chunk_type_has_slot(ChunkTypeName, SlotName).
```

The `chunk_type/1` constraint represents the set $\mathcal{T}$ of chunk-types in the store, but refers only to the chunk-type names. The set of slots of a chunk-type is specified by the `chunk_type_has_slot/2` constraint, i.e. for a chunk-type $T \in \mathcal{T}$, with $T =$

(t, S), there exists a `chunk_type(t)` and for every slot $s \in S$ there is a constraint `chunk_type_has_slot(t,s)` in the constraint store. For the chunks, the following constraints are defined:

```
:- chr_constraint chunk(+,+).
% chunk(ChunkName, ChunkType)

:- chr_constraint chunk_has_slot(+,+,+).
% chunk_has_slot(ChunkName, SlotName, Value)
```

The `chunk/2` constraint represents both the set C of chunks and the Isa relation, since the presence of a constraint `chunk(c,t)` signifies that chunk `c` is of a type $T = (\mathtt{t}, S)$. The $\mathrm{HasSlot}$ relation is represented by the `chunk_has_slot(c,s,v)` constraint, which really is just a direct translation of an element $(c, s, v) \in \mathrm{HasSlot}$.

Note that all values in the just presented constraints have to be ground. This is a demand claimed by the original ACT-R implementation and makes sense, since each value in a slot of a chunk is a real, ground value and the concept of variables does not have an advantage in this context, because every element that can be stored in the brain is assumed to be known by the brain.

Additionally, from the definition of a chunk store it is known that the Isa relation has to be left-total and right-unique. Therefore, for every chunk `c` in the store, exactly one `chunk(c,t)` constraint has to be in the store. Due to type-consistency, for each constraint `chunk_type_has_slot(t,s)` a `chunk_has_slot(c,s,v)` constraint has to be defined. If it should be expressed that a chunk has an empty slot, the special chunk name `nil` can be used as slot value to indicate that. Note that `nil` must not be the name of a regular chunk or chunk-type.

Example 4.2. *The chunk and chunk-type in example 4.1 are represented as:*

```
chunk_type(addition-fact)
chunk_type_has_slot(addition-fact,arg1)
chunk_type_has_slot(addition-fact,arg2)
chunk_type_has_slot(addition-fact,sum)

chunk(a,addition-fact)
chunk_has_slot(a,arg1,5)
chunk_has_slot(a,arg2,2)
chunk_has_slot(a,sum,7)
```

Distinction of Elements and Chunks

A chunk store distinguishes between a set of chunks C and a set of elements E. For implementational reasons it can be helpful if there are only chunks in the system, because elements just behave like chunks with no slots. Hence, a chunk-type `chunk` with no slots will be added automatically to the store. Each element $e \in E$ is added as a chunk of type `chunk` to the set of chunks C. After this operation $E = \emptyset$, and for every former element e of E: $e \in C$, $(e, (chunk, \emptyset)) \in \mathrm{Isa}$. So E is now represented by $\{c \in C \mid c \ \mathrm{Isa} \ (chunk, \emptyset)\}$ in the implementation.

Example 4.3. *The chunk representation from example 4.2 is changed to:*

```
1  chunk_type(addition-fact)
2  chunk_type_has_slot(addition-fact,arg1)
3  chunk_type_has_slot(addition-fact,arg2)
4  chunk_type_has_slot(addition-fact,sum)
5
6  chunk_type(chunk)
7
8  chunk(a,addition-fact)
9  chunk_has_slot(a,arg1,5)
10 chunk_has_slot(a,arg2,2)
11 chunk_has_slot(a,sum,7)
12
13 chunk(5,chunk)
14 chunk(2,chunk)
15 chunk(7,chunk)
```

Simple Implementation of the Default Methods

To implement the methods in definition 4.2, first a data type for chunk specifications has to be introduced. From this specification the correct constraints modeling the chunk-store are added or modified. This data type is necessary to allow communication between implementationally independent modules which do not share a joint memory of constraints (or data in general) as the ACT-R theory suggests (see section 2.1.1). The modular organization and the communication between modules is described in section 4.4.

The straight-forward definition of a data type for chunk specifications is just to use the specification like in definition 4.2: In both ACT-R and the aforementioned definition, chunks are defined by a term `(name isa type slot1 val1 ... slotn valn)` which basically is just a list in Lisp and specifies a chunk uniquely. Hence, a similar Prolog term can be used:

```
1 :- chr_type chunk_def ---> nil; chunk(any, any, slot_list).
2 :- chr_type list(T) ---> []; [T | list(T)].
3 :- chr_type slot_list == list(pair(any,any)).
4 :- chr_type pair(T1,T2) ---> (T1,T2).
```

By the `:- chr_type` directive, new types can be defined. For example, the definition of `list` states that a list of type `T` is either empty (`[]`) or a term `[X|Xs]` where `X` is a value of type `T` and `Xs` is a list of type `T` itself. The first definition states that the type `chunk_def` can be either the atom `nil` or a term of the form `chunk(Name,Type,SVP)` where `Name` and `Type` can be of any type and `SVP` is a `slot_list` which is an alias for a list of pairs and describes a list of slot-value pairs, i.e. a Prolog list of terms `(S,V)`, where `S` is the name of a slot and `V` is an identifier for the value for this slot. This is the direct translation of the chunk-specification used in the definition, amended by the `nil` construct, that may be needed for later purposes.

The defined types can be used in the definitions of CHR constraints. The arguments of these constraints are then checked at runtime for the correct types. For example, the directive `:- chr_constraint a(pair,any)` states that the constraint `a` is expected to have a pair as its first argument and any type as second argument. The default methods can be implemented as follows:

add_chunk This method creates the chunks and elements of the chunk store. The set E of elements is minimal, i.e. only elements that appear in the slots of a chunk but are not chunks themselves are members of E. However, the set E is never constructed explicitly, but represented by chunks of the special type `chunk` that provides no slots. So each value in the slot of a chunk that is added to the store and that is not an element of the chunk store yet, gets its own chunk of type `chunk`. As soon as a chunk with the name of such a primitive element is added to the store, the chunk of type `chunk` is removed from the store.

Listing 4.1: Rules for `add_chunk`

```
1 % empty chunk will not be added
2 add_chunk(nil) <=> true.
```

```
% initialize all slots with nil
add_chunk(chunk(Name,Type, _)), chunk_type_has_slot(Type,S) ==>
  chunk_has_slot(Name,S,nil).

% chunk has been initialized with empty slots -> actually add
    chunk
add_chunk(chunk(Name,Type, Slots)) <=>
  do_add_chunk(chunk(Name,Type,Slots)).
```

First, all `chunk_type_has_slot` constraints are added to the store and initialized with `nil` as slot value. This leads to complete chunk specifications that are consistent to the type as demanded by a type-consistent chunk-store.

If all slots have been initialized, `do_add_chunk` performs the actual setting of the real slot values:

Listing 4.2: Additional rules for adding chunks

```
% base case
do_add_chunk(chunk(Name, Type, [])) <=>
  chunk(Name, Type).

% overwrite slots with empty values
chunk(V,_) \ do_add_chunk(chunk(Name, Type, [(S,V)|Rest])),
    chunk_has_slot(Name,S,nil)  <=>
  chunk_has_slot(Name,S,V),
  do_add_chunk(chunk(Name,Type,Rest)).

% overwrite slots with empty values
do_add_chunk(chunk(Name, Type, [(S,V)|Rest])),
    chunk_has_slot(Name,S,nil)  <=>
  V == nil | % do not add chunk(nil,chunk)
  chunk_has_slot(Name,S,V),
  do_add_chunk(chunk(Name,Type,Rest)).

% overwrite slots with empty values
do_add_chunk(chunk(Name, Type, [(S,V)|Rest])),
    chunk_has_slot(Name,S,nil)  <=>
  V \== nil |
  chunk_has_slot(Name,S,V),
```

```
20   chunk(V,chunk), % no chunk for slot value found => add chunk
         of type chunk
21
22 do_add_chunk(_) <=> false.
```

The first rule is the base case, where no slots have to be added any more. Then, as a last step the actual `chunk` constraint of the chunk that is added to the store is created. The second rule deals with the case, that a slot-value pair has to be added with a value that is already described by a chunk. Then the `nil`-initialized slot of this chunk is removed and replaced by another slot containing the actual value.

The next rule ensures that the helper chunk specification `nil` will not get a chunk in the store, even if it is in the slots of a chunk. Otherwise, if the value of the slot to be added is not `nil`, the next rule can fire and the value of the previously `nil`-initialized slot will be replaced with the actual value. Additionally, since the first rule obviously did not fire for this constellation, `V` is a value different from `nil` that does not have a chunk in the store. Hence, it must be a primitive element. Thus a new chunk of type `chunk` is added to the store for this value as described in section 4.2.2 (see page 38). If no rule matches, the user tried to create a chunk with slots that are not specified in the chunk-type. This leads to an error.

On top of the rules in listing 4.1, there must be added a rule that deletes a primitive element (i.e. a chunk of type `chunk`), if the user introduces a real chunk with the name of this element:

Listing 4.3: Clean up primitive elements

```
1 % delete chunk of Type chunk, if real chunk is added
2 add_chunk(chunk(Name,_,_)) \ chunk(Name,Type) <=>
3   Type == chunk |
4   true.
```

add_chunk_type The following rules create a new chunk type:

```
1 add_chunk_type(CT, []) <=>
2   chunk_type(CT).
3 add_chunk_type(CT, [S|Ss]) <=>
4   chunk_type_has_slot(CT, S),
5   add_chunk_type(CT, Ss).
```

alter_slot This method replaces the value of an existing slot for a given chunk, but only if it is a valid slot for the chunk-type of the altered chunk.

```
1 alter_slot(Chunk,Slot,Value), chunk_has_slot(Chunk,Slot,_) <=>
2   chunk_has_slot(Chunk,Slot,Value).
3 alter_slot(Chunk,Slot,Value) <=>
4   false.
```

The first rule replaces the existing `chunk_has_slot` constraint by a new one. This is called *destructive assignment* as described in [Frü09, p. 32]. The second rule only matches, if the first did not match (due to the refined operational semantics of CHR). This is only the case, if it is tried to alter a slot with a non-existing `chunk_has_slot` constraint. However, since the chunk descriptions are complete, the slot cannot be valid for the type of the chunk and the altering has to fail.

remove_chunk This method removes all occurrences of a chunk from the store. I.e., all `chunk` and `chunk_has_slot` constraints the chunk is involved in are removed.

```
1 remove_chunk(Name) \ chunk(Name, _) <=> true.
2 remove_chunk(Name) \ chunk_has_slot(Name, _, _) <=> true.
3 remove_chunk(_) <=> true.
```

return_chunk This method creates a chunk specification as defined in section 4.2.2 from the chunk name of a chunk in the store.

```
1 chunk(ChunkName, ChunkType) \ return_chunk(ChunkName,Res) <=>
2   var(Res) |
3   build_chunk_list(chunk(ChunkName, ChunkType, []),Res).
4 chunk_has_slot(ChunkName, S, V) \
    build_chunk_list(chunk(ChunkName, ChunkType, L), Res) <=>
5   \+member((S,V),L) |
6   build_chunk_list(chunk(ChunkName, ChunkType, [(S,V)|L]),Res).
7 build_chunk_list(X,Res) <=> Res=X.
```

The first rule creates the initial chunk specification with name and type set, but without any slot specification. This initial representation is handed to the `build_chunk_list`

constraint. The second rule adds a slot-value pair from the store to the list of slot-value pairs in the specification and builds the next chunk specification from this new representation. In the last rule, the process terminates if no other rule can fire any more. Then the result is bound to the handed specification.

Checking Consistency and Type-Consistency

At the moment, there are no rules that check the consistency of the chunk store. However, if the default methods for adding chunks are used, a type-consistent store is built automatically, since every chunk has exactly one chunk-type.[2] Furthermore, all slots from its chunk-type are described and only those slots are described (satisfies type-consistency). Additionally, there are no two different slot descriptions for the same chunk and every chunk in the store is described[3] (satisfies the definition of a chunk-store). Rules for checking those constraints could be added easily to the implementation.

4.3 Procedural Module

The part of the system, where the computations are performed, is the procedural module. It is the central component that holds all the production rules, the working memory (in the buffer system) and organizes communication between modules (through buffers and requests). In the following, all of those subcomponents of the procedural module are described.

4.3.1 Buffer System

The buffer system can be regarded as a chunk-store that is enhanced by buffers. A buffer can hold only one chunk at a time. The procedural module has a set B of buffers, a chunk-store Σ and a relation between the buffers and the chunks in Σ.

Definition 4.3 (buffer system)**.** *A* buffer system *is a tuple* $(B, \Sigma, \mathrm{Holds})$*, where* B *is a set of buffers,* $\Sigma = (C, E, \mathcal{T}, \mathrm{HasSlot}, \mathrm{Isa})$ *a type-consistent chunk-store and* $\mathrm{Holds} \subseteq B \times (C \cup \{\texttt{nil}\})$ *a right-unique and left-total relation that assigns every buffer at most*

[2] left-totality and right-uniqueness of Isa

[3] This is demanded by the type-consistency: Since Isa is left-total, every chunk is in the Isa relation. Type-consistency demands, that every chunk in the Isa relation has a value for all slots of its type.

one chunk that it holds. If a buffer b *is empty, i.e. it does not hold a chunk, then* $(b, \texttt{nil}) \in \mathrm{Holds}$.

A buffer system is consistent*, if every chunk that appears in* Holds *is a member of* C *and* Σ *is a type-consistent chunk-store.*

A buffer system is clean*, if its chunk-store only holds chunks which appear in* Holds.

For the implementation of a buffer system, the code of a chunk-store can be extended by a `buffer/2` constraint that encodes the set B and the relation Holds at once, since the relation is complete by definition.[4]

Destructive Assignment and Consistency

The demand of Holds being right-unique[5] is a form of destructive assignment as described in [Frü09, p. 32], i.e. if a new chunk is assigned to a buffer, the old `buffer` constraint is removed and a new `buffer` constraint is introduced, holding the new chunk:

```
1 set_buffer(B, C) \ buffer(B, _) <=> buffer(B, C).
```

This rule ensures that only one `buffer` constraint exists for each buffer in B. At the beginning of the program, a `buffer` constraint has to be added for all the available buffers of the modules. This problem is discussed in section 4.6.

In addition, if a new chunk is put into a buffer, it also has to be present in the chunk-store, since the production system relies on the knowledge about the chunks in its buffers and chunks are essentially defined by their slots (*consistency property* in definition 4.3). Hence, every time a chunk is stored in a buffer, the `add_chunk` method described in definition 4.2 has to be called. However, the chunks in the slots are not loaded, since they are not present for working memory. They only appear as primitive elements in the chunk-store. Figure 4.1 shows a buffer system which has two buffers. One of the buffers holds a chunk. Note, that the values v_1 and v_2 might be chunks themselves. However, their slots are not loaded to the chunk store and hence they only appear as primitive elements.

[4] $\forall b \in B \;\; \exists c \in (C \cup \{\texttt{nil}\}) : (b, c) \in \mathrm{Holds}$
[5] $\forall b \in B \;\; \forall c, d \in (C \cup \{\texttt{nil}\}) : (b, c), (b, d) \in \mathrm{Holds} \Rightarrow b = c$

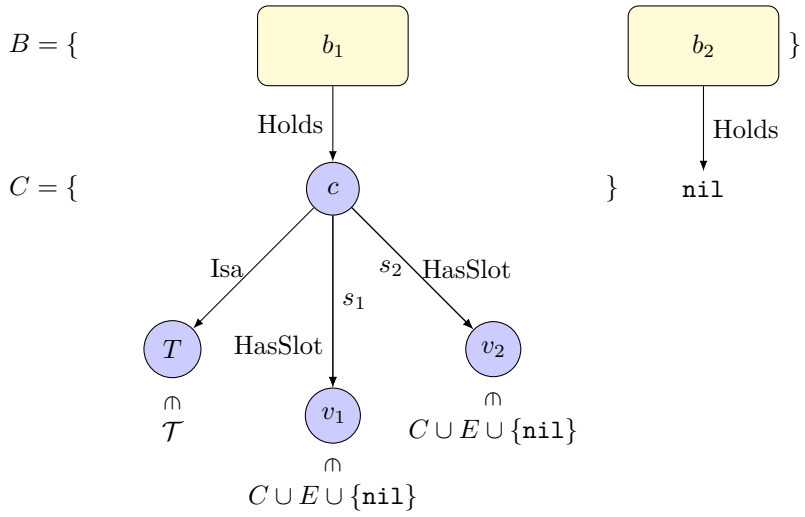


Figure 4.1: A buffer system $(B, \Sigma, \mathrm{Holds})$ with two buffers $b_1, b_2 \in B$ and a chunk store $\Sigma = (C, E, \mathcal{T}, \mathrm{HasSlot}, \mathrm{Isa})$. The relations Holds, Isa and $\mathrm{HasSlot}$ are illustrated by arrows. The buffer b_1 holds a chunk c of type $T = (t, \{s_1, s_2\})$ and with values v_1 and v_2. Buffer b_2 is empty. The buffer system is clean, since only the chunks which are held by a buffer are in the set of chunks C of the chunk store.

Buffer States

Another formal detail of the buffer system is that buffers can have various states: *busy*, *free* and *error*. Those states represent the status of the underlying system, for example the state of the retrieval buffer is based on the current action or result of the declarative module's retrieval system.

A module is busy, if it is completing a request and free otherwise. Since a module can only handle one request at a time and requests may need a certain time (like the retrieval request for example), the procedural module could state another request to a busy module. This is called *jamming* which leads to error messages and should be avoided. One technique to avoid module jamming is to *query* the buffer state in the conditional part of a production rule [Actb, unit 2, p. 9]. The possibility to query buffer states is discussed in the next section.

A buffer's state is set to *error*, if a request was unsuccessful because of an invalid request specification or, in case of the declarative module for instance, a chunk that could not be found. In CHR, a buffer state can be represented by a `buffer_state(b,s)` constraint,

which signifies that buffer `b` has the state `s`. Since every buffer has exactly one state all the time, it is required, that for every buffer there is such a constraint and it is ensured, that only one `buffer_state` constraint is present for each buffer. This can be achieved by the destructive assignment method described in section 4.3.1.

At the beginning of the program, when a buffer is created (a `buffer` constraint is placed into the store), a corresponding `buffer_state` constraint has to be added. The initial state can be set to `free`, since no request is being computed at the time of creation.

4.3.2 Production Rules

Production rules consist of a *condition* part and an *action* part. Syntactically, in ACT-R the condition is separated from the action by **==>**. Additionally, each production rule has a name. Thus, a rule is defined by:

```
1 (p name condition* ==> action*)
```

The condition part is also called the *left hand side* of a rule (LHS) and the action part is called *right hand side* (RHS).

The Left Hand Side of a Rule

Generally, a condition is either a *buffer test*, i.e. a specification of slot-value pairs that are checked against the chunk in the specified buffer or a *buffer query*, i.e. a check of the state of a buffer's module (either busy, free or error). A buffer test on the LHS of a rule is indicated by a = followed by the buffer name of the tested buffer; a query is indicated by a `?` in front of the buffer name. The LHS of a rule may contain bound or unbound variables: `=varname` is a variable with name `varname`.

If the chunks in the buffers pass all buffer tests specified by the rule, the rule can fire, i.e. its right hand side will be applied. The LHS is a conjunction of buffer tests, i.e. there is no specific order for the tests [Botb, p. 165].

Example 4.4 (counting example – left hand side)**.** *This example introduces the ACT-R syntax to define the left hand side of a production rule. The left hand side of the counting rule specified in the example in section 2.1.6 can be defined as follows:*

```
(p count-rule
  =goal>
    isa    count
    number =n
  =retrieval>
    isa    count-fact
    first  =n
    second =m
==>
  ... )
```

The condition part consists of two buffer tests:

1. *The goal buffer is tested for a chunk of type* `count` *and a slot with name* `number`*. The value of the slot is bound to the variable* `=n`*.*
2. *The retrieval buffer is tested for a chunk of type* `count-fact` *that has the variable* `=n` *in its* `first` *slot (with the same value as the* `number` *slot of the chunk in the goal buffer, since* `=n` *has been bound to that value), and another value in its second slot which is bound to the variable* `=m`*.*

The Right Hand Side of a Rule

For the right hand side of a rule the following actions are allowed:

Buffer Modifications of the form

```
=buffer>
  s  v
  ...
```

They are indicated by the buffer modifier `=` right before the buffer name and are specified through a list of slot-value pairs. Note that this list can be incomplete and must not contain a `isa` specification. It leads to all slots mentioned in the modification action of a buffer being replaced by the specified values.

Buffer Requests of the form

```
+buffer>
  s  v
  ...
```

Requests are indicated by the buffer modifier + and specified by a list of slot-value pairs. A request will be sent to the module of the specified buffer which reacts to the transmitted chunk of the request specification (in form of the slot-value pairs). The semantics of a request depends on the module, but the buffer is always first cleared before stating the request.

Buffer Clearings of the form

```
-buffer>
```

Clearings are indicated by the buffer modifier −. When a buffer is cleared, the chunk it contains will be removed from the chunk-store of the buffer system and then be added to the declarative memory.

Example 4.5 (counting example)**.** *The counting rule specified in the example in section 2.1.6 could be defined as follows:*

```
(p count-rule
  =goal>
    isa    count
    number =n
  =retrieval>
    isa    count-fact
    first  =n
    second =m
==>
  =goal>
    number =m
  +retrieval>
    isa    count-fact
    first  =m
)
```

Direct Translation of Buffer Tests

An ACT-R production rule of the form

```
(p name
  =buffer1>
    isa   type1
```

```
    slot1,1 val1,1
    ...
    slot1,n val1,n
  ...
  =bufferk>
    isa   typek
    slotk,1 valk,1
    ...
    slotk,m valk,m
==>
... )
```

states formally, that: If $buffer_1$ Holds $c_1 \ \wedge \ c$ Isa $type_1 \ \wedge \ c_1 \xrightarrow{slot_{1,1}} val_{1,1} \ \wedge \ \ldots \ \wedge$ $buffer_k$ Holds $c_k \ \wedge \ c_k$ Isa $type_k \ \wedge \ \ldots$ is true, then the rule matches and the RHS should be performed. This can be directly translated into a CHR rule:

```
name @
  buffer(buffer1,C1),
  chunk(C1,type1),
  chunk_has_slot(C1,slot1,1,val1,1),
  ...
  chunk_has_slot(C1,slot1,n,val1,n),
  ...
  buffer(bufferk,Ck),
  chunk(Ck,typek),
  chunk_has_slot(Ck,slotk,1,valk,1),
  ...
  chunk_has_slot(Ck,slotk,m,valk,m)
==>
  ...
```

This rule checks the buffer system for the existence of a buffer holding a particular chunk and then checks the chunk store of the buffer system for that chunk with the type and slots specified in the ACT-R rule. The rule is a propagation rule, because the information of the chunk-store should not be removed. If the values in the slot tests are variables, they can be directly translated to Prolog variables.

The CHR rule only fires, if all the checked buffers hold chunks that meet the requirements specified in the slot tests of the ACT-R rule. Since those slot-tests are just a conjunction of

relation-membership tests and the CHR rule is a translation of these tests into constraints, both are equivalent. In detail:

- If a checked buffer `b` holds no chunk, the constraint `buffer(b,nil)` will be present, but the chunk store will not hold any of the required constraints `chunk` or `chunk_has_slot` and the rule will not fire.
- If a checked buffer `b` holds a chunk, but the chunk does not meet one of the requirements in its slots, the rule does not fire.
- The rule only fires, if for all checked buffers there are valid `buffer`, `chunk` and `chunk_has_slot` constraints present that meet all the requirements specified by the ACT-R rule.
- Variables on the LHS of a rule are bound to the values of the actual constraints that are tried for the matching. After the matching, each variable from the rule has a ground value bound to it, because there are no variables in the implementation of the buffer store. This corresponds to the semantics of an ACT-R production rule with variables on the LHS.

Translation of Actions

For each action type a constraint

```
1 buffer_action(buffer,chunk-specification)
```

and the corresponding rules that handle the action have to be added. Note that the buffer action is defined by the action name, the buffer name and a chunk specification. That is because actions in ACT-R are defined through a buffer modifier that specifies the action, the name of the buffer and a list of slot-value pairs. Hence, this is a direct translation to CHR.

Buffer Modifications The modification of a buffer takes an incomplete chunk specification and modifies the given slots of the chunk in the specified buffer. This can be implemented as follows:

```
1 buffer(BufName, OldChunk) \ buffer_change(BufName,
    chunk(_,_,SVs)) <=>
2   alter_slots(OldChunk,SVs).
```

This implementation uses a generalization of the `alter_slot` method as described in definition 4.2 and section 4.2.2:

```
1 alter_slots(_,[]) <=> true.
2 alter_slots(Chunk,[(S,V)|SVs]) <=>
3   alter_slot(Chunk,S,V),
4   alter_slots(Chunk,SVs).
```

Note that types cannot be changed. This corresponds to the grammar definition of ACT-R as presented in section 4.3.3.

Buffer Clearings When clearing a buffer, the chunk that was stored in the buffer will be removed from the chunk store and `nil` will be written to the store. Additionally, the chunk is written to declarative memory.

```
1 buffer_clear(BufName), buffer(BufName, ModName, Chunk) <=>
2   write_to_dm(Chunk),
3   delete_chunk(Chunk),
4   buffer(BufName, nil).
```

`write_to_dm` handles the writing of the chunk to the declarative memory:

```
1 write_to_dm(ChunkName) <=> return_chunk(ChunkName, ResChunk),
    add_dm(ResChunk).
```

where `add_dm` is basically just a global wrapper for the `add_chunk` method of the declarative memory and will be explained in section 4.5.1.

Buffer Requests The buffer requests have to be handled a little bit differently from the other actions. Therefore, they will be explained in section 4.4.2. The changes to the buffer system are presented in section 4.4.3 and an example implementation of the module request interface for retrieval requests is given in section 4.5.2.

Example 4.6 (counting example in CHR – simple)**.** *In this example, the translation of a simple ACT-R production rule to a CHR rule is shown. Hence, the translation of the production rule in example 4.5 is inspected. It leads to the following translation:*

```
count-rule @
  buffer(goal,C1),
    chunk(C1,count),
    chunk_has_slot(number,N),
  buffer(retrieval,C2),
    chunk(C2,count-fact),
    chunk_has_slot(first,N),
    chunk_has_slot(second,M)
==>
  buffer_change(goal,chunk(_,_,[(number,M)])),
  buffer_request(retrieval,chunk(_,count-fact,[first,M])).
```

Translation of Buffer Queries

A buffer query

```
?buffer>
  state  bstate
```

on the LHS of a production rule can be translated to the following CHR rule head:

```
... buffer_state(buffer,bstate) ... ==> ...
```

4.3.3 The Production Rule Grammar

The discussed concepts lead to the following grammar for production rules, which is a simplified version of the actual grammar used in the original ACT-R implementation [Botb, p. 162]. Some of the details in this grammar that have not been discussed yet are presented in the following sections.

Listing 4.4: The ACT-R production rule grammar

```
production-definition ::= (p name condition* ==> action*)
name ::= a symbol that serves as the name of the production for reference
condition ::= [ buffer-test | query ]
```

action **::=** [buffer-modification | request | buffer-clearing | output]
buffer-test **::=** =buffer-name> isa chunk-type slot-test*
buffer-name **::=** a symbol which is the name of a buffer
chunk-type **::=** a symbol which is the name of a chunk-type in the model
slot-test **::=** *{*slot-modifier*}* slot-name slot-value
slot-modifier **::=** [= | − | < | > | <= | >=]
slot-name **::=** a symbol which names a possible slot in the specified chunk-type
slot-value **::=** a variable or any Lisp value
query **::=** ?buffer-name> query-test*
query-test **::=** *{*−*}* queried-item query-value
queried-item **::=** a symbol which names a valid query for the specified buffer
query-value **::=** a bound-variable or any Lisp value
buffer-modification **::=** =buffer-name> slot-value-pair*
slot-value-pair **::=** slot-name bound-slot-value
bound-slot-value **::=** a bound variable or any Lisp value
request **::=** +buffer-name> isa chunk-type request-spec*
request-spec **::=** *{*slot-modifier*}* slot-value-pair
request-parameter **::=** a Lisp keyword naming a request parameter provided by the buffer specified
buffer-clearing **::=** −buffer-name>
variable **::=** a symbol which starts with the character =
output **::=** !output! [output-value]
output-value **::=** any Lisp value or a bound-variable
bound-variable **::=** a variable which is used in the buffer-test conditions of the production (including a variable which names the buffer that is tested in a buffer-test or is bound with an explicit binding in the production

The Order of Rule Applications

In the current translation scheme, the order of the rule applications does not match the semantics as described in the ACT-R theory (see section 2.1.3). Consider the following rules in a compact notation:

```
=b1>
  isa foo
  s1  v1
==>
=b1>
```

```
  s1  v2
=b2>
  s   x
```

and

```
=b1>
  isa foo
  s1  v2
==>
=b2>
  s   y
=b1>
  s1  v3 % for termination
```

In the semantics of ACT-R, if the first rule matches, all the buffer modifications are performed first. After that, the procedural module can search for the next matching rule, which is the second one (due to the result of the first rule). This rule then would overwrite the value `x` in the `s` slot of buffer `b2` with `y`.

```
buffer(b1,C),
  chunk(C,foo),
  chunk_has_slot(C,s1,v1)
==>
buffer_change(b1,chunk(_,_,[(s1,v2)])),
buffer_change(b2,chunk(_,_,[(s,x)])).

buffer(b1,C),
  chunk(C,foo),
  chunk_has_slot(C,s1,v2)
==>
buffer_change(b2,chunk(_,_,[(s,y)])),
buffer_change(b1,chunk(_,_,[(s1,v3)])). % this is for
    termination
```

For the query

```
?- chunk(c,foo), chunk_has_slot(c,s1,v1), chunk(c2,bar),
    chunk_has_slot(c2,s,v), buffer(b2,c2), buffer(b1,c).
```

the result would be

```
buffer(b1,c)
buffer(b2,c2)
chunk(c2,bar)
chunk(c,foo)
chunk_has_slot(c2,s,x)
chunk_has_slot(c,s1,v3)
```

I.e., the buffer `b2` holds the chunk with value `x`. This is due to the fact, that after the first rule has modified the buffer `b1`, the second rule matches and will be fired immediately, which then changes the value of `b1` to `y`. Afterwards, the second action of the first rule will be executed and the `s` slot of the chunk in buffer `b2` will be overwritten with `x`.

Hence, somehow the fact that the procedural module is busy and cannot fire another rule has to be modeled. This can be achieved by a simple phase constraint `fire` that will be added after the complete execution of a rule and will be removed as soon as a rule is executed.

For every production rule, the rule

```
{ buffer_tests } ==> { actions }
```

has to be changed to a simpagation rule

```
{ buffer_tests } \ fire <=> { actions }, fire.
```

It is important that the adding of `fire` is the last action of a rule.

The example would be modified as follows:

```
buffer(b1,C),
  chunk(C,foo),
  chunk_has_slot(C,s1,v1)
  \ fire
<=>
buffer_change(b1,chunk(_,_,[(s1,v2)])),
buffer_change(b2,chunk(_,_,[(s,x)])),
fire.
```

```
9  buffer(b1,C),
10   chunk(C,foo),
11   chunk_has_slot(C,s1,v2)
12   \ fire
13 <=>
14 buffer_change(b2,chunk(_,_,[(s,y)])),
15 fire.
```

The test query

```
1 ?- chunk(c,foo), chunk_has_slot(c,s1,v1), chunk(c2,bar),
     chunk_has_slot(c2,s,v), buffer(b2,c2), buffer(b1,c), fire.
```

yields

```
1 buffer(b1,c)
2 buffer(b2,c2)
3 chunk(c2,bar)
4 chunk(c,foo)
5 chunk_has_slot(c,s1,v3)
6 chunk_has_slot(c2,s,y)
7 fire
```

The fire constraint has to be added at the end of the query. This ensures the correct semantics and a completely constructed buffer system.[6]

In appendix B.1 a minimal executable example is provided.

Bound and Unbound Variables

According to the ACT-R production rule grammar in listing 4.4, unbound variables can only appear on the left hand side of a rule. Hence, no new variables are introduced on the right hand side. Since all the elements in the buffer store are completely described and ground, every variable on the LHS of a rule will be bound to a ground value. This simplifies the rule selection and application process a lot, since every value of the calculation is known

[6] This was actually the reason why in the first example without the `fire` constraint the buffer `b1` was created at the end of the query: If it would be created at an earlier point, the first rule would have matched immediately and the computation would have yielded a different result.

after the matching. This enables simple implementations of arithmetic tests, for example (see section 4.3.3).

Duplicate Slot Tests

A problem that has not been addressed yet is that ACT-R allows buffer tests like the following:

```
=buffer>
  isa  foo
  bar  spam
  bar  spam
```

In the logical reading, this would signify that $buffer$ Holds $c \ \wedge \ c$ Isa $foo \ \wedge \ c \xrightarrow{bar} spam \ \wedge \ c \xrightarrow{bar} spam$ which is equivalent to the test with only one check of the `bar` slot since $x \wedge x$ can be reduced to x. However, in CHR the following rule head resulting from the simple translation scheme does not match:

```
buffer(buffer,C),
  chunk(C,foo),
  chunk_has_slot(C,bar,spam),
  chunk_has_slot(C,bar,spam)
```

Because there are no two identical `chunk_has_slot` constraints in the store, the rule cannot fire. I.e. the list of slot tests in ACT-R is interpreted as a set of conditions, whereas CHR rules operate on a multi-set of constraints. However, the semantics of CHR can be changed to a set-based semantics. The first approach to implement a set-based semantics is to simply eliminate the duplicate test:

```
buffer(buffer,C),
  chunk(C,foo),
  chunk_has_slot(C,bar,spam)
```

However, this does not do the trick for all possible cases. For example, suppose a test

```
=buffer>
  isa  foo
```

```
3   bar  spam
4   bar  =x
```

where the second test refers to a variable (or even both tests are variables). This problem can be solved by adding a guard to the rule from the simple translation:

```
1 buffer(buffer,C),
2 chunk(C,foo),
3 chunk_has_slot(C,bar,spam)
4 ==> spam == X | ...
```

Since `X` must be bound after the matching,[7] the test will always give the correct result. This also works for the first example, where the test would be `spam == spam`, which is always true and hence could be reduced to the rule with the second test eliminated and no guard.[8] It also works if the two slot-tests were contradictory, for example:

```
1 bar  spam
2 bar  eggs
```

describes a rule that never can fire. The translation models exactly this behaviour:

```
1 buffer(buffer,C),
2 chunk(C,foo),
3 chunk_has_slot(C,bar,spam)
4 ==> spam == eggs | ...
```

since the built-in syntactic-equality test of Prolog does never return true for those two constants.

This approach is essentially the same as Frühwirth suggests in chapter 6 of the accompanying slides to his book [Frü09]: For each multi-headed rule, new rule variants are added. Those variants are produced by a pairwise unification of the constraints in the head which eliminates one of the unifiable head constraints [Frü10]. For example, consider the rule:

```
1 a(1,Y), a(X,2) ==> b(X,Y).
```

[7] see section 4.3.3
[8] true guards can always be reduced

The constraints `a(1,Y)` and `a(X,2)` can be unified to `a(1,2)`. Therefore the following rule is added:

```
1 a(1,2) ==> b(1,2).
```

Since in ACT-R all variables in the head of a rule have to be bound, this scheme can be simplified to a simple guard-check and the elimination of duplicate slot tests. Hence, no rule variants have to be added, but the rules containing duplicate slot tests are modified.

Slot Modifiers

In ACT-R, slot-tests can be preceded by *slot modifiers*. Those modifiers allow to specify tests like inequality (−) or arithmetic comparisons (`<, >, <=, >=`) of the slot value of a chunk with the specified variable or value. Since the slots in a chunk store are always fully defined with ground values, those tests are decidable.

If no slot modifier is specified in a slot test, the default modifier = is used, that states that the chunk in the specified buffer must have the specified value in the specified slot. This default semantics has been used in the previous sections when translating simple ACT-R rules to CHR and is performed automatically by the matching of CHR.

To translate the other slot modifiers to CHR, another CHR mechanism can be used: Guards. Since the allowed modifiers are all default built-in constraints,[9] a slot test with a modifier

```
1 ...
2 =buffer>
3   ...
4   ~slot  val
5 ...
6 ==>
```

where ~ stands for one of the modifiers in { `=,-,<,>,<=,>=` } can be translated as follows:

```
1 buffer(buffer,C),
2   ...
```

[9] i.e. Prolog predicates

```
  chunk_has_slot(C,slot,V),
  ...
==>
  V # val |
  ...
```

where # is the placeholder for the built-in constraint that computes the test specified by ~ and V is a fresh variable that has not been used in the rule yet.

For arithmetic slot modifiers the values being compared have to be numbers. If a value is not a number, the arithmetic test will fail and the rule cannot be applied [Botb].

Note that slot tests with modifiers other than = do not bind variables, but only perform simple checks, like it is with guards in CHR. If `val` is an unbound variable and is never bound to a value on LHS, the default implementation throws a warning, and the rule will not match. Therefore, to handle this case, the rule translation scheme has to be extended by an additional guard check `ground(Val)`, where `Val` is the Prolog variable that replaces each occurrence of the variable `val`.

As with normal slot tests, it is important to mention that if there are several tests on the same slot, the `chunk_has_slot` constraint must appear only once on the LHS of the CHR rule, since every slot-value pair is unique in the constraint store. Ie., if the first slot test of a particular slot appears on the LHS of the ACT-R rule, a `chunk_has_slot` constraint has to be added to the LHS of the CHR rule. For every other occurrence of this slot in a slot test, only guard checks are added.

Example 4.7. *To clarify the details of the matching concept in ACT-R, here are some examples and their behaviour:*

```
=buffer>
  isa    foo
  -spam  =bar
```

will throw a warning when loading the model. When running it, the rule will never fire, since no chunk value will match the inequality to the unbound variable `bar`*.*

```
=buffer>
  isa    foo
  -spam  =bar
  eggs   =bar
```

will fire, if there is a chunk whose value in `eggs` *is different from the value in* `spam`*.*

```
=buffer>
  isa    foo
  spam  =bar
  spam  =eggs
```

matches for every value of the spam slot. The translation to CHR is:

```
buffer(buffer,C),
  chunk(C,foo),
  chunk_has_slot(C,spam,Bar)
==>
  Bar=Eggs | ...
```

In this case, a binding occurs in the guard, which is usually unwanted. However, since the `Eggs` *variable is unbound and therefore a fresh variable introduced in the guard, it is allowed and does not harm the matching process. It is even necessary to bind* `Eggs` *to* `Bar` *because of the semantics of the equivalent ACT-R rule. The following example is a little bit different: The rule*

```
=buffer>
  isa    foo
  spam  =bar
  spam  =eggs
  ham   =eggs
```

will match all chunks which have the same value in `spam` *and* `ham`*. This translates to*

```
buffer(buffer,C),
  chunk(C,foo),
  chunk_has_slot(C,spam,Bar),
  chunk_has_slot(C,spam,Eggs)
==>
  Bar==Eggs | ...
```

Note that in this case, no binding occurs, since both, `Bar` *and* `Eggs`*, already occur in the head of the rule and are bound to some value. Hence, the test in the guard can be reduced to a simple syntactic equality test without binding (==).*

Relation to Negation-as-Absence The concept of negation-as-absence as described in [Frü09, pp. 147 sqq.] is provided by many production rule systems. It enables the programmer to negate a fact in the sense that the fact is not explicitly in the store and therefore the rule is applicable – so the programmer asks for the absence of a particular fact.

At the first glance, the slot modifiers seem to implement this concept, but this is not the case: All negated slot tests in ACT-R can be reduced to simple built-in guard checks, because the chunks in the store are always described completely and the values are ground, so simple built-in checks work automatically. This also works for invalid slot-tests that ask for slots which are not offered by the chunk-type they ask for. Then there will never be a constraint matching the head of the rule due to type consistency. With these restrictions, ACT-R avoids the problems that come with negation-as-absence as they are explained in [Frü09, pp. 147 sqq.] and the translation of such tests to CHR is very simple.

Empty Slots

An important special case in the semantics of ACT-R production rules is that if there is a slot test specified, a potential chunk only matches if it really has a value in this slot. Chunks that have `nil` in a slot specified in a buffer test will not match the test. Hence, variables can not be used to test if two slots have the same value and the value is `nil`, since every positive slot test involving `nil` fails automatically [Botb, p. 164, section “Variables”, last sentence].

In CHR this special case can be handled by adding a guard for each variable occurring in a positive slot-test checking that this variable does not equal `nil`. For negated slot tests, this is not the case: The following rule matches also a chunk with an empty `spam` slot:

```
1 =buffer>
2   isa    foo
3   -spam  4
```

Outputs

The production system of ACT-R also provides methods to produce side-effects. In this work, only a subset of those methods is concerned: the outputs. Outputs can appear on the right hand side of an ACT-R rule:

```
=buffer>
  isa  foo
==>
  !output! (a1 a2 a3)
```

The argument of such an output call is a list of Lisp-symbols, so it is possible to hand variables or terms.

This mechanism can be translated to Prolog directly:

```
output([]).
output([X|Xs]) :-
  write(X),nl,
  output(Xs).
```

The `Xs` have to be Prolog terms. In ACT-R, function calls like `!eval!` or `!bind!` are allowed, but are ignored in this work.

4.4 Modular Organization

By now, all components of the implementation have been assumed to be in one file: buffer system, production rules, declarative memory, ... This leads to problems like name space pollution and duplicate code, since, e.g. both the buffer system and the declarative memory use different chunk stores, which have the same behaviour but different data. It would be practical to reuse this code in both modules, but keeping the stores separated. Additionally, the code is better readable and it is easier to add new modules if a program is distributed over multiple files. Finally, the ACT-R architecture already proposes a modular organization, so it seems likely to adopt this idea of a modular architecture.

The term *module* is highly overloaded: In ACT-R it describes independent parts of human cognition, whereas in the world of programming the term is used in a slightly different manner. In the following, implementational modules will always be named explicitly as *Prolog modules*.

Nevertheless, the modular organization of ACT-R with its independent modules can be implemented by defining a Prolog module for each ACT-R module and adding some other modules around them. In the following, the concept of Prolog modules is explained.

4.4.1 Prolog Modules

Defining a new module creates a new namespace for all CHR constraints and Prolog predicates, which is illustrated in the following example:

Example 4.8 (Prolog Modules and CHR)**.** *In this example, two modules* `mod1` *and* `mod2` *are defined, with partially overlapping constraints.* `mod2` *exports the constraint* `c`*. In the following, the behaviour and interaction of the modules is explored.*

Listing 4.5: Definition of Module 1

```
1 :- module(mod1,[]).
2 :- use_module(library(chr)).
3 :- use_module(mod2).
4 :- chr_constraint a/0, b/0.
5
6 a <=> c.
```

Listing 4.6: Definition of Module 2

```
1 :- module(mod2,[c/0]).
2 :- use_module(library(chr)).
3 :- use_module(mod1).
4 :- chr_constraint a/0, b/0, c/0.
5
6 a <=> b.
7 b <=> mod1:a.
```

In this definition, two new modules `mod1` *and* `mod2` *are created and only the* `c` *constraint of* `mod2` *is exported, indicated by the lists in the module definitions. The modules import each other.*

The CHR constraint `a` *in listing 4.5 is internally represented as* `mod1:a`*, so it lives in its own namespace and does not pollute other namespaces. The constraint can appear on the right hand side of rules of other modules, but has to be called explicitly with its full namespace identifier. In line 8 of listing 4.6, the presence of the local* `a` *constraint leads the rule to fire and* `mod2:a` *is replaced by* `mod2:b`*, which leads the rule in line 9 to fire and replaces the local* `mod2:b` *constraint by an external* `mod1:a` *constraint. So, external constraints can be called by their complete identifiers. However, on the left hand side of a rule, only the constraints local to the current module can appear.*

Exported constraints can only appear once in a program, since they can be called without their namespace definition, which is demonstrated in line 8 of listing 4.5, where `mod2:c` *is called in* `mod1` *without referring to* `mod2` *explicitly.*

4.4.2 Interface for Module Requests

The architecture of ACT-R provides an infrastructure for the procedural module to state requests to all the other modules. To implement this concept as general as possible, an interface has to be defined which allows the adding of new modules to the system by just implementing this interface. Later on, the interface will be refined.

Listing 4.7: Simple Interface *IModule*

```
1 module_request(+BufName,+Chunk,-ResChunk,-ResState)
```

The arguments of such a request are:

BufName The name of the requested buffer, e.g. `retrieval`.

Chunk A chunk specification that represents the arguments of the request. The form of the allowed chunk specifications and the semantics of the request are module-dependent. For example: `chunk(_,t,[(foo,bar),(spam,eggs)])` could describe a chunk, that should be retrieved from declarative memory.

The request provides the following result:

ResChunk The resulting chunk in form of a chunk specification. The actual result and its semantics depend on the particular module.

ResState The state of the buffer after the request. For example, if no matching chunk could be retrieved from declarative memory, the state would be `error`.

4.4.3 Requests by the Buffer System

Every module that can handle a request implements the interface in listing 4.7. When the buffer system gets a call `buffer_request(buffer,chunk-specification)`, it simply can call the `module_request` method of the corresponding module. This can be achieved by `ModName:module_request(...)`, where `ModName` is the name of the corresponding module.

Hence, the buffer system must know which buffer belongs to which module. The `buffer/2` constraint therefore has to be extended to a `buffer/3` constraint, that also holds the module name of its module:

```
1 buffer(BufName,ModName,Chunk)
```

A buffer request can now be handled as follows:

Listing 4.8: Retrieval Request in CHR

```
1  buffer(BufName, ModName, _) \ buffer_request(BufName, Chunk) <=>
2    % module is busy now
3    set_buffer_state(BufName,busy),
4    % clear buffer immediately
5    buffer_clear(BufName),
6    % send request
7    ModName:module_request(BufName, Chunk, ResChunk,ResState),
8
9    % Apply result of the request
10   % case 1: resulting state is error
11   (ResState=error,
12   buffer(BufName, ModName, nil),
13   set_buffer_state(BufName,error) ;
14
15   % case 2: no errors occured
16   ResState = free,
17   ResChunk = chunk(ResChunkName,_,_),
18   add_chunk(ResChunk),
19   buffer(BufName, ModName, ResChunkName),
20   set_buffer_state(BufName,free)).
```

When getting a buffer request, the buffer state is set to `busy` first. Then the buffer is cleared immediately, which leads the chunk in the buffer being added to the declarative memory. Then the module request is stated according to the interface. If the resulting state is `error`, then the buffer gets this state and the content of the buffer is `nil`, otherwise, if the resulting state is `free`, the result will be added to the buffer and the state will eventually be set to `free`. It is important to mention, that the received chunk will be created explicitly in the chunk store of the buffer system and therefore is a copy of a possibly existing identical chunk in the module that has been requested.

4.4.4 Components of the Implementation

As mentioned before, the implementation is divided into modules which provide and require interfaces. Figure 4.2 gives an overview of the architecture of the implementation as a component diagram. The main component is the *model* which consists of *user-defined rules*, that is the translated production rules as described in section 4.3.2 and the ACT-R core. The core consists of the *buffer system* (see section 4.3.1) which implements a *chunk-store* (see section 4.2) and a module *stdlisp* which is capable of handling default lisp functions as described in section 4.8. The interface *ICore* consists of the buffer interface – which basically offers the access to the buffers and its contents for the matching and the buffer actions – and the Lisp function interface as described in the aforementioned sections. Additionally, the core provides the constraints `fire/0` and `apply_rule/1` for reasons described in section 4.9.2 on page 104. Furthermore, the core uses the *scheduler* module which provides the *IScheduler* interface described in section 4.7.2 and defined in definition 4.5. The configuration and the *IConfiguration* and *IObserver* interfaces are described in section 4.8.2. Finally, the interface *IModule* which is implemented by every ACT-R module (and in this case especially by the *declarative module*) is described in a simplified version in listing 4.7 and in its final version in listing 4.10.

It is important to mention, that in the figure, each component which is included in another component signifies that the corresponding Prolog file is textually included to the parent component. The interfaces in the figure indicate that the component is a Prolog module which is included by the use of `use_module`. The constraints defined in such a standalone component are not accessible for the other components. An exception is the component of the *user-defined rules* which – in the current version – textually includes the *core*.

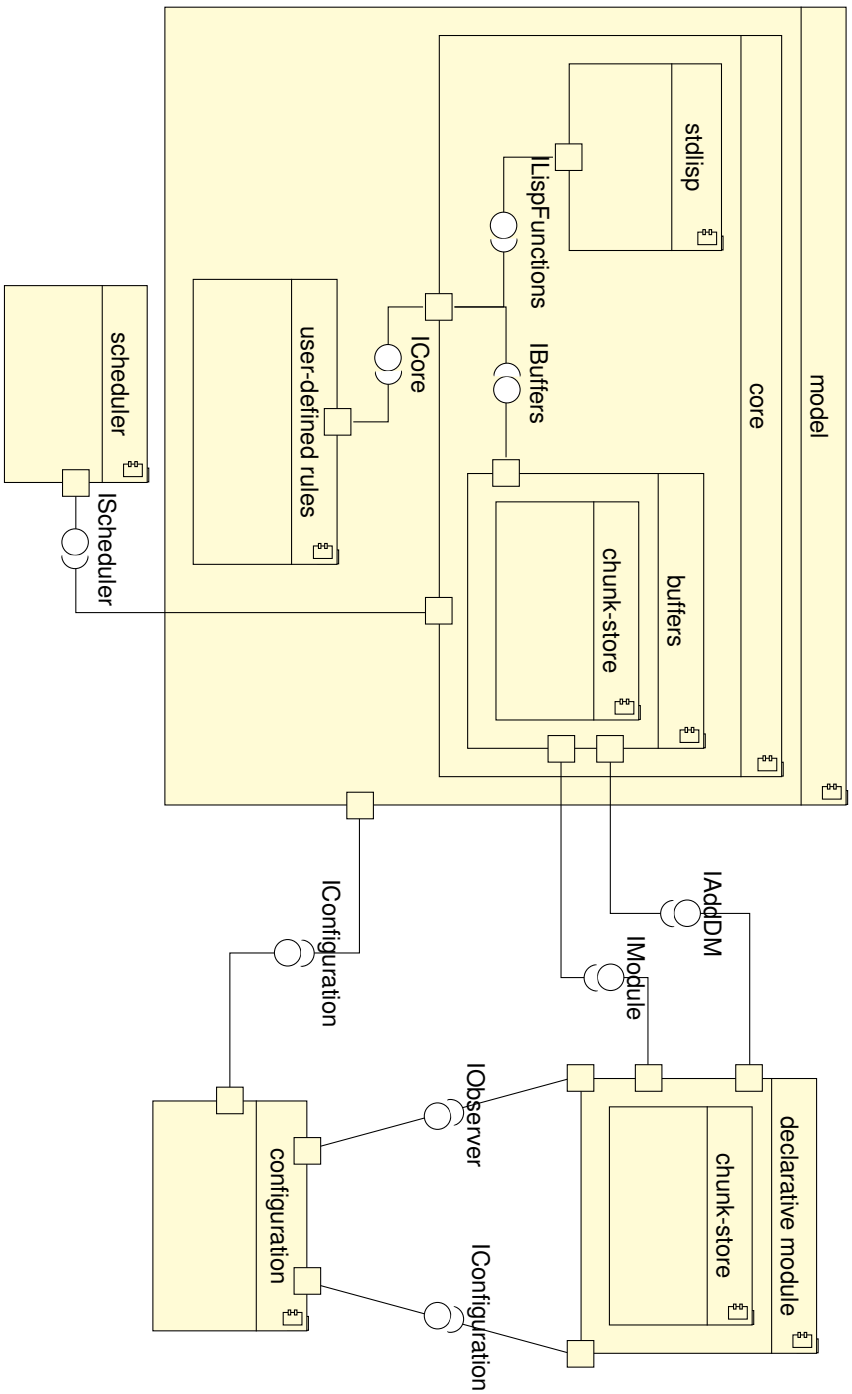


Figure 4.2: UML-Component Diagram of the implementation

4.5 Declarative Module

The Declarative Module is a *chunk store*, that additionally implements the *module* interface. Therefore some rules to handle requests that find certain chunks in the chunk store have to be implemented. The chunk-store is separated from the buffer system which has its own chunk-store. Hence, all chunks requested by the buffer system from the declarative module are copies and therefore changes on those chunk do not affect the chunks in the declarative memory directly.

4.5.1 Global Method for Adding Chunks

Since the declarative module is very important to the whole theory it is kind of a special module. Therefore, its Prolog module exports a predicate `add_dm` that is just a wrapper for its chunk store. Hence, with the command `add_dm` chunks can be added to the chunk store of the declarative module from every part of the program. This is e.g. used for the clearing of buffers in the buffer system, where the chunk of a buffer is written to declarative memory when the buffer is cleared. This is the implementation of the command:

```
1 add_dm(ChunkDef) <=> add_chunk(ChunkDef).
```

4.5.2 Retrieval Requests

A retrieval request gets an incomplete chunk specification as input and returns a chunk whose slots match the provided chunk pattern.

Chunk Patterns

The chunk patterns are transmitted in form of chunk specifications as defined in section 4.2.2. Since those specifications may be incomplete, variables are considered as place-holders for values in the result. The result always is a complete and ground chunk specification, because every chunk in a chunk store has to be defined completely; empty slots are indicated by the value nil.

Example 4.9. *In this example, some possible requests are discussed.*

1. *Request:*

```
chunk(foo,bar,_)
```

   A chunk with name `foo`*, type* `bar` *and arbitrary slot values is requested.*

2. *Request:*

```
chunk(_,bar,_)
```

   This request is satisfied by every chunk of type `bar`*.*

3. *Request:*

```
chunk(_,t,[(foo,bar),(spam,eggs)])
```

   The most common case of requests is a specification of the type and a (possibly incomplete) number of slot-value pairs for that type. If a type does not provide a specified slot, the request is invalid and no chunk will be returned.

Finding Chunks

In this section, a CHR constraint `find_chunk/3` will be defined which produces a `match_set/1` constraint for each chunk that matches a specified pattern. Eventually, the match set will be collected and returned.

```
find_chunk(N1,T1,Ss), chunk(N2,T2) ==>
  unifiable((N1,T1),(N2,T2),_),
  nonvar(Ss) |
  test_slots(N2,Ss),
  match_set([N2]).

find_chunk(N1,T1,Ss), chunk(N2,T2) ==>
  unifiable((N1,T1),(N2,T2),_),
  var(Ss) |
  test_slots(N2,[]),
  match_set([N2]).

find_chunk(_,_,_) <=> true.
```

First, each chunk in the store whose name and type are unifiable with the specified name and type, will be part of the initial match set. If in the chunk specification the name and type are variables, each chunk will match. For the unification test, the `unifiable/3` predicate of Prolog is used, because the unification should not be performed but only tested. If name and type match the pattern, then the slots have to be tested by `test_slots/2`.

The rule in line 7 is for chunk specifications that do not specify the slots. In this case, no slots have to be tested. If all chunks have been tested or no chunk matches at all, the process is finished (rule in line 13).

After adding each matching candidate to the match set whose name and type have already been checked, the match set is pruned from chunks that have non-matching slot values:

```
1  test_slots(_,[]) <=> true.
2
3  chunk_has_slot(N,S,V1), match_set([N]) \
       test_slots(N,[(S,V2)|Ss]) <=>
4    unifiable(V1,V2,_) |
5    test_slots(N,Ss).
6
7  chunk_has_slot(N,S,V1) \ test_slots(N,[(S,V2)|_]),
       match_set([N]) <=>
8      \+unifiable(V1,V2,_) |
9      true.
10
11 test_slots(N,_) \ match_set([N]) <=> true.
```

The first rule is the base case, where no slots have to be tested any more and the test is finished and has been successful.

In line 3, the rule is applicable if there is at least one slot `(S,V2)` that has to be tested and a $HasSlot$ relation of the kind $\texttt{N} \xrightarrow{\texttt{S}} \texttt{V1}$ with the slot `S` to be tested, that is still in the match set, so no conflicting slot has been found yet. If the values `V1` and `V2` are unifiable, i.e. both are the same constant or at least one is a variable, the test passes and the chunk `N` remains in the match set and the rest of the slot tests are performed.

The third rule in line 8 is applied if the guard of the first rule did not hold, so the values `V1` and `V2` are not unifiable, but there is a connection $\texttt{N} \xrightarrow{\texttt{S}} \texttt{V1}$ and in the request it has been specified that the value of slot `S` has to be `V2`. In this case, $\texttt{V1} \neq \texttt{V2}$, so the test

fails and the chunk `N` has to be removed from the match set, since one of its slots does not match.

If the last two rules cannot be applied, the chunk does not provide a slot that, however, has been specified in the request. Hence, the chunk does not match and the last rule therefore deletes it from the match set.

If those rules have been applied exhaustively, only the matching chunks will remain in the match set: If there was an outstanding slot test, one of the rules would be applicable and the chunk would be removed from the match set, if the test fails (and the test would also be removed, because it has been performed). If the test is successful, the chunk will remain part of the match set, but the test will be removed. So the match set is correct and complete.

However, since the match set is distributed over a set of `match_set` constraints, it would be desirable to collect all those matches in one set. This can be triggered by the constraint `collect_matches/1`, which returns the complete match set in its arguments as soon as there is only one `match_set` left:

```
1 collect_matches(_) \ match_set(L1), match_set(L2) <=>
2   append(L1,L2,L),
3   match_set(L).
4
5 collect_matches(Res), match_set(L) <=> Res=L.
6
7 collect_matches(Res) <=> Res=[].
```

The first rule merges two match sets to one single merge set containing a list with all the chunks of the former sets, if the `collect_matches` trigger is present.

In the second rule, if there are no two match sets to merge in the store any more, the result of the `collect_matches` operation is the remaining match set. The same applies for the last rule, where no match set is in the store and therefore the result is empty. Note that this implies that the rules have to be applied from top to bottom, left to right.[10]

The symbolic layer does not implement any rule stating which chunk will be returned if there is more than one chunk in the match set. In this implementation, the first chunk in the list is chosen. The module request is now implemented as follows:

[10] This is called the refined operational semantics of CHR

```
module_request(retrieval,chunk(Name,Type,Slots),ResChunk,
    ResState) <=>
  find_chunk(Name,Type,Slots),
  collect_matches(Res),
  first(Res,Chunk),
  return_chunk(Chunk,ResChunk),
  get_state(ResChunk,ResState).
```

where `first(L,E)` gets a list `L` and returns its first element `E` or `nil` if the list is empty.

With `return_chunk/2` and `get_state/2`, the actual results of the request are computed: By now, the variable `Chunk` holds the name of the chunk to return, but in the specification of the module request, a complete chunk specification is demanded. `return_chunk/2` is defined as a default method of a chunk store that gets a chunk name as its first argument and returns a chunk specification created from the values in the chunk store as its second argument.

The resulting state of the request is computed as follows: If the result chunk is `nil`, no chunk was in the match set, so the state of the declarative module will be `error`. In any other case, the state is `free` after the request has been performed.

4.5.3 Chunk Merging

An important technique used in ACT-R's declarative memory module is chunk merging: If a chunk enters the chunk store and all of its slots and values are the same as of a chunk already in the store, then those two chunks get merged. Both names refer to the merged chunk.

If a chunk entering the declarative memory has the same name as a chunk that already is in the store, the new chunk first gets a new name. Then, if its slots are the same as the slots of the old chunk, they are merged and the new name is deleted.

```
chunk(Name,Type) \ add_chunk(chunk(Name,Type,Slots)) <=>
  Type \== chunk |
  add_chunk(chunk(Name:new,Type,Slots)).

% first check, if identical chunk exists
add_chunk(chunk(C,T,S)) ==>
```

```
7   T \== chunk |
8   check_identical_chunks(chunk(C,T,S)).
```

The rules are added before the empty slot initialization in listing 4.1. The first rule handles the case where a chunk with an already allocated name is added to the store. Then it gets a new name (which basically is just the old name extended by `:new`) and tries to add this new chunk to the memory.

The second rule adds an identity check for each chunk to be added except for primitive elements, since their slots are identical for every name, because every primitive element has no slots. Nevertheless they have to be distinguishable by their names and therefore cannot be merged.

The identity check is implemented as follows:

```
1  check_identical_chunks(nil) <=> true.
2  chunk(NameOld,Type),
       check_identical_chunks(chunk(NameNew,Type,Slots)) ==>
3    check_identical_chunks(chunk(NameNew,Type,Slots),NameOld).
4  check_identical_chunks(_) <=> true.
```

The rule in line 3 adds for each identity check and each chunk in the store a pairwise identity check which is performed by the following rules:

```
1  chunk(NameOld, Type) \ check_identical_chunks(chunk(NameNew,
       Type, []), NameOld) <=>
2    identical(NameOld,NameNew).
3
4  chunk(NameOld, Type), chunk_has_slot(NameOld, S, V) \
       check_identical_chunks(chunk(NameNew, Type, [(S,V)|Rest]),
       NameOld) <=>
5    check_identical_chunks(chunk(NameNew,Type,Rest),NameOld).
6
7  chunk(NameOld, Type), chunk_has_slot(NameOld, S, VOld) \
       check_identical_chunks(chunk(_, Type, [(S,VNew)|_]),
       NameOld) <=>
8    VOld \== VNew |
9    true.
10 remove_duplicates @ identical(N,N) <=> true.
```

```
11
12 % abort checking for identical chunks if one has been found
13 cleanup_identical_chunk_check @ identical(NameOld,NameNew) \
       check_identical_chunks(chunk(NameNew,_,_),NameOld) <=> true.
```

The first rule is the base case, where no slots have to be tested any more. Then the chunks are identical and an `identical/2` constraint is added to the store for those two chunks.

The next rule applies for an identity check for a slot-value pair that is successful, whereas the third rule handles the opposite case and aborts the check for this pair of chunks without adding an `identical` constraint.

In the fourth rule redundant information is removed and the last rule aborts the identity check as soon as one matching chunk has been found.

Note that this implementation assumes that the chunk specification of the chunk entering the declarative memory is complete. If it is incomplete, the correct semantics of the adding is that all unspecified slots are empty so their values equal `nil`. Nevertheless, this implementation would merge the chunk with a chunk that has values in those unspecified slots. This could be improved by completing the chunk specifications before checking for identical chunks in the store.

For the easier use of the identical constraint at other points, transitive identities can be reduced to the only real chunk in the store (the one with the `chunk` constraint):

```
1 reduce @ chunk(C1,_), identical(C1,C2) \ identical(C2,C3) <=>
2   identical(C1,C3).
```

Additionally, identical chunks should not be added to the store explicitly, since they are just a copy of another chunk and their name is a reference to this actual chunk. The helper chunks with the artificial `:new` suffix are also not added to the store and their identical information is deleted in the end, since their nomenclature was only store internal, so only the original name must be kept:

```
1 % do not add chunks with :new tag, remove identical information
2 add_chunk(chunk(C:new,T,S)), identical(C,C:new) <=> true.
3 % do not add identical chunk, but keep identical information
4 identical(C1,C2) \ add_chunk(chunk(C2,T,S)) <=> true.
```

In the declarative module, the chunk search must be extended to find merged chunks by both names. This can be achieved as follows:

```
1 identical(C1,C2) \ find_chunk(C2,T2,Ss2) <=>
2   ground(C2) |
3   find_chunk(C1,T2,Ss2).
```

so the chunk search of a chunk that is only a pointer to another chunk can be reduced to the search of this chunk.

The process of chunk merging is described in [Botb, pp. 217, 71] and [Actb, p. 3]. The treatment of identical names is not described there, but has been tested in the original implementation: ACT-R handles those identical names similarly to this approach by adding a sequential number to the identical name. For example, the chunk with name `a` would be called `a-0` if there already was a chunk `a` in the store.

4.6 Initialization

In the examples, the models had to be run by stating complex queries which create all necessary buffers and add all chunk types and chunks to the declarative memory manually. In the original ACT-R implementation, the command `run` is used to run a model. This behaviour can be transferred easily to the CHR implementation by adding a `run` constraint and a rule for this constraint, that performs all the initialization work.

```
1 init @
2   run <=>
3   add_buffer(retrieval,declarative_module),
4   add_chunk_type(...),
5   add_dm(...),
6   goal_focus(...),
7   fire.
```

This example assumes that there are implementation of the called methods that perform the partial initialization work. A typical initialization routine could create all buffers at first, then add all needed chunk-types (note, that this might include the artificial chunk-type `chunk` as defined in section 4.2.2 (see page 38). Then the initial declarative memory chunks are added. The `goal-focus` method selects a created chunk and copies it to the

declarative memory. The adding of the `fire` constraint eventually starts the computation after the initialization is finished as described in section 4.3.3 (see page 53).

4.7 Timing in ACT-R

So far, the execution order of the production rules has been controlled by the `fire` constraint – a phase constraint which so far has simulated the occupation of the production system while executing a rule.

However, certain buffer actions like buffer requests may take some time to finish. The procedural system is free to fire the next rule after all actions have been started[11] and the requests are performed in parallel to that. Additionally, for the simulation it may be interesting to explore how much time certain actions have taken, especially when it comes to the subsymbolic layer.

Those aspects cannot be implemented easily using the current approach with phase constraints. Hence, the idea of introducing a central scheduling unit is a possible solution of those requirements: The unit has a serialized ordered list of events with particular timings. If a new event is scheduled, the time when it is executed must be known. The scheduling unit inserts the event at the right position of the list preserving the ordered time condition. Figure 4.3(a) illustrates this approach.

The system periodically removes the first event from the queue, sets the current time to the time of the event and executes it which may lead to new events in the queue. The queue organizes the right order of the events automatically. Figure 4.3(b) shows the effect of the removal to the queue and the system time. With this approach, the simulation of a parallel execution of ACT-R can be achieved: Each buffer action on the RHS of a rule just schedules an event that actually performs the action at a specified time (the current time plus its duration). The central part of the scheduler is a *priority queue* which manages a list of events and returns them by time. It is described in the next section.

4.7.1 Priority Queue

A *priority queue* is an abstract data structure that serves objects by their priority. It provides the following abstract methods:

[11] see chapters 2.1.3 and 2.1.7

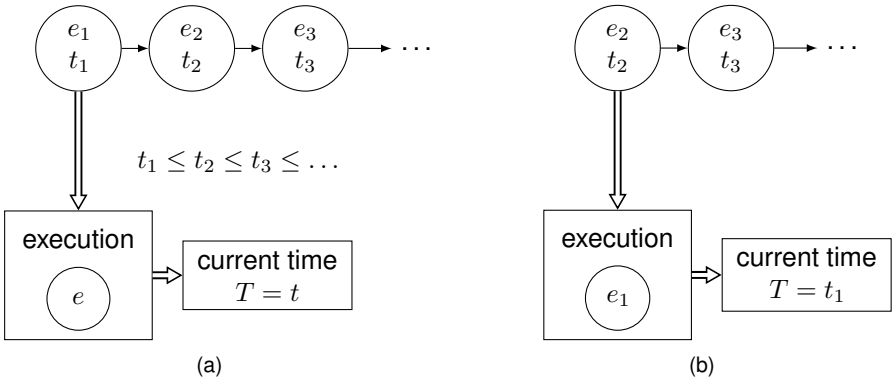


Figure 4.3: Scheduling with an event queue. (a) The scheduler stores the events e_i with their corresponding times t_i in a queue which is sorted by the times. Currently, the event e is executed and the time T is t. (b) If the event e has been executed completely, the next event – the first event e_1 of the queue – is removed. The current time is set to the time t_1 and the event is executed.

enqueue with priority An object with a particular priority is inserted to the queue.

dequeue highest priority The object with the highest priority is removed from the queue and returned.

Objects

In the implementation of the scheduler, the priority queue contains objects of the form `q(Time,Priority,Event)`. The priority of such a queue object is composed from the `Time` and the `Priority`. An `Event` is a Prolog goal, i.e. a built-in or a CHR constraint. The order between the elements is defined as follows:

Definition 4.4 (ordered time-priority condition)**.** *`q(T1,P1,E1)` $\prec$ `q(T2,P2,E2)`, i.e. the object `q(T1,P1,E1)` is the predecessor of the object `q(T2,P2,E2)`, if `T1` $<$ `T2`. In the case that `T1 = T2`, then `q(T1,P1,E1)` $\prec$ `q(T2,P2,E2)` if `P1` $>$ `P2`. So, events with smaller times will be returned first. If two events appear at the same time, it is possible to define a priority and the event with the higher priority will be returned first.*

Representation of the Queue

The representation of the priority queue is inspired by [Frü09, pp. 38 sqq.]: An order constraint `A --> B` is introduced, which states that `A` will be returned before `B` (or `B` is

the direct successor of A). The beginning of the queue will be defined by a start symbol s, so the first real element is the successor of s. A possible queue could be:

```
    s     --> q(1,0,e1)
q(1,0,e1) --> q(3,7,e2)
q(3,7,e2) --> q(3,2,e3)
```

This queue achieves the ordered time-priority condition, since the queue objects are in the correct order according to their times and priorities. It is also consistent in a sense that it has no gaps and no object has more than one successor.

In general, some rules to make such a queue consistent can be defined, i.e. every object only has one successor and the queue achieves the time-priority condition:

```
A --> A <=> true.

_ --> s <=> false.

A --> B, A --> C <=>
  leq(A,B),
  leq(B,C) |
  A --> B,
  B --> C.
```

The first rule states, that if an object is its own successor, this information can be deleted. The second rule states, that nothing can be the predecessor of the start symbol. The last rule is the most important one: If one object has two successors, then these connections have to be divided into two connections according to the defined time-priority condition. This condition can be implemented as follows:

```
leq(s,_).

% Time1 < Time2 -> event with time1 first, ignore priority
leq(q(Time1,_,_), q(Time2,_,_)) :-
  Time1 < Time2.

% same time: event with higher priority first
leq(q(Time,Priority1,_), q(Time,Priority2,_)) :-
  Priority1 >= Priority2.
```

The first predicate states that the start symbol is less than every other object. The other two rules directly implement the time-priority condition as defined in definition 4.4.

Note that two objects are considered the same, iff their time, priority and event are syntactically the same. If an object is altered in one of the `-->` constraints, it has to be edited in every other occurrence in the list to avoid gaps.

Another important property is, that the list does not have any gaps, so it must be possible to track the queue from every element backwards to the start symbol. This condition is not achieved by the rules above, but since a priority queue only offers two mechanisms to modify it, a lot of those problems can be avoided:

add_q(Time,Priority,Event) Enqueues an object with the specified properties. I.e., a new `q(Time,Priority,Event)` object will be created and the following constraint will be added: `s --> q(Time,Priority,Event).` The rules presented above will move the element to the correct position since it conflicts potentially with the former successor of the start symbol `s`. This leads to a linear, serialized list achieving the time-priority condition without gaps.

```
1 add_q(Time,Priority,Evt) <=>
2   s --> q(Time,Priority,Evt).
```

de_q(X) Dequeues the first element of the queue according to the time-priority condition and binds its value to `X`.

```
1  % queue with more than one element
2  de_q(X), s --> A, A --> B <=>
3    X = A,
4    s --> B.
5
6  % queue with only one element
7  de_q(X), s --> A <=>
8    X = A.
9
10 % empty queue
11 de_q(X) <=> X = nil.
```

The first object is just the successor of `s`, since the list has been constructed preserving the correct order and the property, that everything starts at `s`. If the first object has a successor, this object is the new first object. If there are no order constraints left, the queue is empty and `de_q` returns `nil`.

Adding Events at Second Position

In this implementation, another default method is added for the priority queue to insert an event which is the direct successor of the first event and hence avoids the default enqueuing methods. It sets the time and priority automatically according to the desired position of the new event:

after_next_event(E) Adds the event `E` to the queue after the first event without destroying the consistency and the time-priority condition of the queue.

To implement this method, the time and priorities have to be set in a way that the time-priority condition does hold:

```
1 s --> q(Time,P1,E1) \ after_next_event(Evt) <=>
2   NP1 is P1 + 2, % increase priority of first event, so it
      still has highest priority
3   P is P1 + 1, % priority of event that is added, ensured that
      it is higher than of the former second event (because it
      is P1+1)
4   de_q(_), % remove head of queue
5   add_q(Time,NP1,E1), % add head of queue again with new
      priority. Will be first again, because it has old prio
      (which is higher than prio of all successors)
6   add_q(Time,P,Evt). % add new event. Will be < than Prio of
      head but it is ensured that it is higher than prio of
      second event
```

If the first event is `q(Time,P1,E1)` and a new event `Evt` has to be added after this event, the times of the two events are the same, the priority of the first event is `P1 + 2` and the priority of the new event is `P1 + 1`. The first event is removed from the queue and would be added with its new priority to the queue again, as it is with the new event.

This is correct: The first event will be the first event again, because its old priority was higher than any other priority at that point of time. Since the new priority is even higher than that, no other event from the queue will have a higher priority. The new event also has a higher priority than every other event in the queue, but a lower priority than the first event, so it will be added after the first event.

By the `de_q` and `add_q` actions it is ensured that no garbage of the old event remains in the queue and the events are added correctly through an official method of the queue.

4.7.2 Scheduler

The scheduler component is an own module that manages events by feeding a priority queue and controls the recognize-act cycle. It also holds the current time of the system.

Current Time

The current time can be saved in a `now/1` constraint. It is important that there is only one such constraint and that time increases monotonically. Other modules can access the time only by a `get_now/1` constraint that just returns the current time saved in the `now/1` constraint. The current time cannot be set from outside, but is determined by the last event dequeued from the priority queue.

Interface to the Scheduler

Definition 4.5 (IScheduler)**.** *The* IScheduler *interface provides the following methods:*

Queue Management *The methods* `add_q/3`, `de_q/1` *and* `after_next_event/1` *which are described above provide access to the queue of the scheduler.*

Timings *The setting and getting of the time by* `get_now/1` *and* `now/1`*. The latter one should only be used once in the initialization. The setting of the current time is managed by the scheduler automatically.*

Start next cycle *The constraint* `nextcyc/0` *is only used to initialize the first cycle and should not be used at a later point, since the management of the recognize-act-cycle is managed by the scheduler.*

No rule *If no rule was applicable, the procedural module can trigger the* `no_rule/0` *constraint which has effects to the next conflict resolution event, as mentioned before.*

Recognize-Act Cycle

As described before, the procedural module can only fire one rule at a time. When executing the RHS of a rule, all its actions are added to the scheduler with the time point when their execution is finished. For example: If on the RHS of the firing rule a retrieval request has to be performed, an event will be added to the priority queue with the time

$Now + Duration$, so the chunk retrieved from the declarative memory will be written to the retrieval buffer at this time point.

After all events of the RHS have been added to the scheduler, the procedural module is free again and therefore the next rule can fire. The event of firing will be added to the priority queue as well by the `fire` constraint at the end of each production rule. The rule will be added at the current time point, but with low priority, since it has to be ensured that every action of the previously executed rule has been performed yet (in the sense that a corresponding event has been added at the specified time point). In many cases, no other rule will be applicable at this time, because most of the meaningful production rules have to wait for the results of the preceding production rule.

Many of the buffer actions are performed immediately, so for buffer modifications or clearings, an event with the current point of time is added to the queue. They are performed in a certain order defined by their priority as shown in table 4.1. The request actions are performed at last, but they perform a buffer clearing immediately and then start their calculation which can take some time.

Table 4.1: Priorities of buffer actions

Priority	Action
100	Buffer modification
50	Module requests
10	Buffer clearings

If no rule is applicable, the next time for a possible application is after having performed the next event. So, the next `fire` event is scheduled directly after the first event in the queue. This simulates the behaviour that the procedural module stays ready to fire the next rule, without polling at every time point if a rule is applicable, but only reacting on changes to the buffer system.

The following enumeration summarizes the recognize-act cycle with a scheduler:

1. The next event is removed from the queue, the current time is set to the time of the event and the event is performed.
   a) The dequeued event is a `fire` constraint: The rule that matches all its conditions is fired and removes the `fire` constraint.
   b) The actions of the rule are scheduled in the queue. Modifications and Clearings have the current time point, requests have a time point in the future depending on the module.

c) The last action of the rule is to add a `fire` constraint to the queue with the current time point and a very low priority. This simulates that the procedural module is free again, after all in-place actions of a rule have been performed, so the next matching rule may fire.
d) There are two possibilities:
   i. *The next rule matches:* It will be performed like the last rule.
   ii. *No rule matches:* The next possible time for a rule to fire is when something in the buffers has changed. This cannot happen until the next event has been performed. So the next `fire` event will be added to the queue by `after_next_event` which has been described above.

2. Go to point 1. This is performed until there are no events in the queue.

The following parts are necessary to implement this cycle:

Start Next Cycle The constraint `nextcyc` leads the system to remove the next event from the queue and perform it. Performing is done by a `call_event` constraint:

```
% After an event has been performed, nextcyc is triggered.
%This leads to the next event in the queue to be performed.
nextcyc <=> de_q(Evt), call_event(Evt).
```

Call an Event Event calling just takes a queue element and sets the current time to the time of the event and performs a Prolog `call`. Additionally, a message is printed to the screen. After the event has been executed, the next cycle is initiated.

If the queue element is `nil` (i.e. no event has been in the queue), the computation is finished and the current time is removed.

```
% no event in queue -> do nothing and remove current time
call_event(nil) \ now(_) <=> write('No more events in queue.
    End of computation.'),nl.
call_event(q(Time,Priority,Evt)), now(Now) <=>
  Now =< Time |
  now(Time),
  write(Now:Priority),
  write(' ... '),write('calling event: '), write(Evt),nl,
  call(Evt),
  nextcyc.
```

Changing the Buffer System For each buffer action, add a `do_buffer_action` constraint, that actually performs the code specified in the former action. Modify the action as follows:

```
% Schedule buffer_action
buffer_action(BufName, Chunk) <=>
  get_now(Now),
  Time is Now + Duration,
  add_q(Time, Priority, do_buffer_action(BufName, Chunk)).
```

with appropriate values for `Duration` and `Priority`.

Production Rules As in the last version of the production system, each rule has the following structure:

```
rule @
  {conditions} \ fire <=>
    {actions},
    conflict_resolution.
```

where `conflict_resolution/0` just schedules the next `fire` event and is defined as:

```
conflict_resolution <=>
  get_now(Now),
  add_q(Now,0,fire).
```

As the last production rule, there has to be:

```
no-rule @
  fire <=>
    no_rule.
```

which removes the `fire` constraint if still present and states that no rule has been fired, because otherwise the firing of a rule would have removed the `fire` constraint. If no rule was applicable, a new `fire` event is scheduled after the next event:

```
1 no_rule <=>
2   write('No rule matches -> Schedule next conflict resolution
        event'),nl,
3   after_next_event(do_conflict_resolution).
```

4.8 Lisp Functions

In ACT-R, it is possible to call Lisp functions in the model definitions, for example to add chunk-types and declarative memory or to set configuration variables. Since those functions cannot be called in the CHR/Prolog environment, there has to be a mechanism to implement missing functions.

4.8.1 General Translation

Lisp functions are lists (indicated by round braces `(` and `)`), which have the functor as their first element and the function arguments in the rest of the list. Hence, each Lisp function `(f arg1 arg2 ...)` called in the model definition of an ACT-R model is translated into the adding of a CHR constraint `lisp_f([arg1, arg2, ...])`. For all used functions, there has to be a default implementation in the module `std_lisp`. In this implementation, the concept is reduced to simple procedural calls ignoring return values, which is adequate for most of the pure ACT-R models without the experiment environment. Here is an example implementation of the Lisp function `(chunk-type name slot1 slot2 ...)` which adds a new chunk-type to the system:

```
1 lisp_chunktype([Type|Slots]) <=>
2   % add type to buffer system (local module)
3   add_chunk_type(Type,Slots),
4   % add type to declarative module
5   declarative_module:add_chunk_type(Type,Slots).
```

Note that this function adds the chunk-types both to the declarative module and the buffer system, since types are global to the system. Hence, every module with a chunk store should be added to the implementation of this Lisp function.

4.8.2 Configuration Variables

In ACT-R, configuration variables which control the behaviour of the architecture are set with the Lisp function `(sgp :Var Val)`, which sets the variable **`Var`** to `Val`. To handle such configuration variables, a configuration module can be added to the CHR implementation that offers the *IConfiguration* interface to set variables and to ask for their values:

Definition 4.6 (IConfiguration)**.** *The interface* IConfiguration *provides the following methods to access the configuration store:*

Set configuration variables *By calling `set_conf(+Var,+Val)` the configuration variable **`Var`** is set to the value `Val`.*

Get the value of a configuration variable *The call of `get_conf(+Var,-Val)` binds the value of the configuration variable **`Var`** to the argument `Val`.*

The `(sgp ...)` call can then be translated to:

```
1 lisp_sgp([]) <=> true.
2 lisp_sgp([:,Var,Val|Rest]) <=>
3   set_conf(Var,Val),
4   lisp_sgp(Rest).
```

The Observer Pattern

Sometimes it is necessary for a module to get informed about the change of a configuration variable. Therefore, the configuration module implements the *Observable* interface, which offers an abstract method `add_config_observer(Module,Var)`. Thereby a module can register at the configuration module for update notifications of a particular variable. If the variable is changed by a `set_conf` call, all observers of this variable can be notified by the following code:

```
1 % collect observers
2 observer(Module,Var), set_conf(Var,_) ==> notify(Module,Var).
3
4 % actually set variable
5 set_conf(Var,Val), configuration(Var,_) <=>
6   configuration(Var,Val),
```

```
7   notify_all(Var). % variable has been set -> perform pending
        notifications
8  notify_all(Var), notify(Module,Var) ==>
9   Module:update. % perform all pending notifications
10 notify_all(_) <=> true. % no notifications pending, clean up
```

The constraint `observer(Module,Var)` states that the module `Module` observes the variable **`Var`**. The observer modules have to declare a `update/0` constraint which may trigger some action related to the change of a configuration variable.

4.9 Subsymbolic Layer

So far, a translation scheme and a framework implementing the symbolic concepts of ACT-R has been presented. This section extends this implementation by the subsymbolic layer as described in section 2.2.

4.9.1 Activation of Chunks

The activation of a chunk is a numerical value that determines if a chunk is retrieved and how long is the latency of the retrieval. The next sections describe how the concept can be implemented in CHR.

Base-Level Learning

One part of the activation value is the base-level activation of a chunk. The value is learned by the system and depends on practice as stated in equation (2.2):

$$B_i = \ln\left(\sum_{j=1}^{n} t_j^{-d}\right)$$

Presentations of a Chunk To determine the base-level value of a chunk, the time points when it has been practiced have to be known. A chunk is considered as practiced when it enters the declarative memory either explicitly by calling `add_dm` or implicitly by a

buffer clearing. Additionally, if a chunk is merged with a chunk that enters the declarative memory, the original chunk is strengthened (so the chunk that has been in the declarative memory before is considered as presented).

Hence, every time a chunk enters the declarative memory, the time of this event has to be stored. Therefore, a `presentation/2` that holds the chunk name and the time of a presentation is introduced. To simplify the use of this constraint, the procedural constraint `present/1`, that stores the presentation of a chunk at the current time, can be implemented as follows:

```
chunk(Name,Type) \ present(chunk(Name,Type,_)) <=>
  getNow(Time),
  presentation(Name,Time).
```

The `add_dm` command is extended by a presentation event:

```
add_dm(ChunkDef) <=>
  add_chunk(ChunkDef),
  present(ChunkDef).
```

If a chunk that is identical to a chunk already stored enters declarative memory, the original chunk is being strengthened by:

```
identical(C1,C2) \ present(chunk(C2,_,_)) <=>
  present(C1).
```

Calculating the Base-Level Value Somewhere in the process of a request, the activation of a set of chunks has to be calculated. To trigger the calculation of the activation of one particular chunk, the trigger constraint `calc_activation(Chunk,Activation)` which gets a chunk name and binds its activation value to the second argument is introduced:

```
presentation(C,PTime), calc_activation(C,A) ==>
  get_now(Now),
  Time is Now - PTime,
  base_level_part(C,Time,_,A).

calc_activation(_,_) <=> true.
```

These two rules produce for every presentation of the chunk a `base_level_part` with the name of the chunk and the time since a particular presentation has happened. Additionally, two unbound variables are given to the part constraint: The first is a variable that will hold an intermediate result and the second is the actual activation value that is bound to the return value of the `calc_activation` constraint.

Each of those `base_level_part` constraints are part of the result which is the activation value of their chunk. The following rule converts the time t_j to the value t_j^{-d} as stated in equation (2.2):

```
% if A and B not set: set B to Time^(-D). Time is the time
    since the presentation of this base_level_part
base_level_part(_,Time,B,A) ==>
  var(A),
  var(B),
  Time =\= 0 |
  get_conf(bll,D), % decay parameter
  B is Time ** (-D).
```

The result is bound to the variable `B`. The rule can only be applied if the intermediate result `B` and the result `A` are not bound to any value. As soon as the intermediate result has been calculated, two `base_level_part` constraints can be merged by adding their `B` values up according to the base-level learning equation (2.2):

```
% collect base level parts and add them together. Only if Bs
    are set
base_level_part(C,_,B1,A), base_level_part(C,_,B2,A) <=>
  nonvar(B1),
  nonvar(B2),
  var(A) |
  B is B1+B2,
  base_level_part(C,_,B,A).
```

If this rule is applied to exhaustion, only one `base_level_part` constraint will remain. Hence, below this rule, a rule for this single constraint can be introduced:

```
% if B is set, A is not set and there are no more
    base_level_parts of this chunk: calculate actual base level
    activation and store it in A. Only possible if B is =\= 0.
```

```
base_level_part(_,_,B,A) <=>
  var(A),
  nonvar(B),
  B =\= 0 |
  A is log(B).
```

Note that the rule has to be executed after the last rule cannot be applied anymore, otherwise it would take an intermediate result as the actual result.

With this set of rules, the base-level activation can be calculated according to its definition. There are some special cases that can be handled implementation specifically (all cases where the guards prevent the rules from firing are such cases that may need some kind of special treatment).

Fan Values In the calculation process, the fan value fan_j of a chunk j may be needed. This value is the number of chunks where j appears in the slots plus one for the chunk itself. So, a chunk that does not appear in any slots has the fan value 1, a chunk that appears in the slots of another chunk has the value 2, etc. This can be achieved in CHR as follows:

```
chunk(C,_) ==> fan(C,1).

chunk_has_slot(_,_,C), chunk(C,_) ==> fan(C,1).

fan(C,F1), fan(C,F2) <=>
  F is F1+F2,
  fan(C,F).
```

The first rule adds a fan of 1 for each chunk. The next rule adds a fan of 1 for a chunk `C` for each slot where `C` is the value. In the last rule, two fan values for one particular chunk are summed up to a single fan value. Note that this has to be considered when deleting a chunk: For every chunk in the slots of the deleted chunk, the fan value must be decreased by one.

As noted before in section 4.2.2 on page 38, primitive elements are stored as chunks of the type `chunk` that has no slots. This is important for the fan calculation as it is presented here, since the fan value depends on the presence of a `chunk` constraint

to calculate the proper fan value. Otherwise, the fan value of primitive elements would always be off by one.

Associative Weights The activation calculation also depends on a contextual component, the associative weights. For a value S_{ji}, which describes the associative strength from a chunk j to a chunk i, the following rules apply:

```
1 fan(J,F), chunk(I,_), chunk_has_slot(I,_,J) \ calc_sji(J,I,Sji)
    <=>
2   I \== J |
3   Sji is 2 - log(F).
4
5 calc_sji(_,_,Sji) <=> Sji=0.
```

The `calc_sji(J,I,Sji)` gets a chunk `J` and a chunk `I`, calculates their associative weight from `J` to `I` and binds it to `Sji`. The first rule can be applied if `J` appears in the slots of chunk `I`. Then, the associative weight is calculated according to equation (2.4):

$$S_{ji} = S - \ln(\text{fan}_j)$$

The value of S is assumed to be 2 in this rule; a configurable constant for S as described in section 4.8.2 can be introduced easily.

If `J` does not appear in the slots of chunk `I`, then $S_{\texttt{JI}} := 0$ by definition (line 5).

Calculating the Overall-Activations To calculate the overall-activations, the constraint `calc_activations/2` is introduced. It gets a list of chunks and a context and initiates the computation of the overall activation values of the chunks in the list regarding the context. Remember that for the associative weights, the current context plays a role, since all associative weights from the chunks (the js) in the context to the chunk whose activation is calculated are summed up (see section 2.2.1 for details).

```
1 calc_activations([],_) <=>
2   true.
3
4 calc_activations([C|Cs],Context) <=>
5   calc_activation(C,B),
6   calc_activations(Cs,Context),
```

```
7   context(C,Context,Assoc),
8   length(Context,N),
9
10  Assoc1 is 1/N * Assoc,
11  A is B + Assoc1,
12  max(C,A).
```

The first rule is the base-case that simply finishes the calculation. The second rule takes the first chunk in the list and calculates its base-level activation `B` by `calc_activation`. The next line triggers the computation for the rest of the chunks in the list (using the same context).

For the chunk `C` the context component, i.e. the associative weight, is calculated by the call of `context(C,Context,Assoc)` in line 7, which binds the associative weight to `Assoc`. I.e. the result of $\sum_{j \in C} S_{ji}$ (C is the context as represented by `Context`) is bound to the variable `Assoc`.

Afterwards, the attentional weighting W_j is considered in lines 8 and 10. The variable `Assoc1` now holds the value of $W_j \cdot \sum_{j \in C} S_{ji}$. The overall activation of chunk `C` is – as defined in equation (2.3) – calculated in line 11 by adding the base-level activation to the sum of the associative weightings of the chunk. Eventually, in line 12, the activation of this chunk is added to the set of potential maximum candidates of all activation values. The maximum then is calculated as follows:

Listing 4.9: Calculate highest activation of all matching chunks

```
1  max(_,A1) \ max(_,A2) <=>
2    A1 >= A2 |
3    true.
```

This deletes all potential candidates for the maximal activation value that have a smaller value than one of the other candidates. In [Frü09, pp. 19 sqq.] this algorithm is presented in more detail.

Threshold and Maximum The request is only successful, if there is a matching chunk that has an activation higher than a specified threshold. In the following, it is assumed that the threshold is saved in a `threshold/1` constraint.

In the last section, the chunk with the highest activation among all matching chunks is calculated and stored in a `max/2` constraint (there is only one `max` constraint after the rule in listing 4.9 has been applied to exhaustion). The constraint `get_max/2` triggers the final maximum computation and binds the chunk and its activation to its parameters:

```
1  get_max(MN,MA), max(N,A), threshold(RT) <=>
2    A >= RT |
3    MN=N,
4    MA=A.
5
6  get_max(MN,MA), max(_,A), threshold(RT) <=>
7    A < RT |
8    MN=nil,
9    % set activation to threshold
10   MA=RT,
11   write('No chunk has high enough threshold'),nl.
12
13 get_max(MN,MA), threshold(RT) <=>
14   MN=nil,
15   % set activation to threshold if no chunk matches
16   MA=RT,
17   write('No chunk matches.'),nl.
```

The first rule is applied in case of the activation of the maximal chunk being higher than the threshold. Then, the chunk and its activation are simply returned.

In the second rule, the matching chunk with the highest activation does not pass the threshold. Then no chunk can be returned, so the result of the `get_max` request is `nil`. The activation of this empty chunk is set to the threshold, because the retrieval latency, i.e. the time the retrieval request takes depends on the threshold in case that no chunk could be found. So this value is used later.

The last rule will only be applied if both of the other rules did not match. This is the case, if there has no `max` constraint been put to the store, which only occur, if no chunk matched the request. This also leads to an empty chunk as a result.

New Module Request Interface Requests that regard the subsymbolic layer need some additional information and return some additional results: First, somehow the *Context*, i.e. all chunks that are in the values of the buffer chunks, have to be passed

from the buffer system to the requested module (in this case the declarative module, but it may be possible that there are other modules which need the context). Additionally, the requested module has to return the time it takes, since every request may take a different time that is only known by the module, but has to be considered by the scheduler of the procedural module.

The interface from section 4.4.2 changes to:

Listing 4.10: The final version of the interface *IModule*

```
module_request(+BufName, +ChunkDef, +Context, -ResChunk,
   -ResState, -ResTime)
```

where `Context` is a list of the chunk names that are in the current context (as defined in section 2.2.1 on page 21) and `ResTime` is bound to the time the request will take. As described in section 4.7.2 on page 85, the buffer actions are divided in two phases: The first only schedules the second phase at the time when the action is finished, whereas the second phase actually performs the action, i.e. the changes to the buffers.

A similar method as for the buffer modifications or clearings is applied for buffer requests: They are divided into three rules with three trigger constraints – `buffer_request/2`, `start_request/2` and `do_buffer_request/2` – similar to the division into two rules as already described in section 4.7.2 on page 85 for the other actions. The difference is that, since a request may take a certain time, the first rule schedules the start of the request with the priority `0`, so all other types of buffer actions have been performed. The request is actually performed immediately when calling `start_request`, but the effects to the requested buffer are applied not until `do_buffer_request` is called, which is scheduled at the result time (`ResTime`) of the request. This is due to the dependence of the latency of a request on the activation values of the matching chunks, so the request has to be performed in advance to calculate the correct time it will take, so the result has to be known before the request ends in simulation time. This yields the following code in the buffer system:

Listing 4.11: Schedule the start of a buffer request

```
1 % just add start_request event with current time
2 buffer_request(BufName, Chunk) <=>
3   get_now(Now),
4   add_q(Now,0,start_request(BufName,Chunk)).
```

Listing 4.12: Execute the buffer request and schedule time the result is applied

```
1  % start the request
2  buffer(BufName, ModName, _) \ start_request(BufName, Chunk) <=>
3    write('Started buffer request '),
4    write(BufName),nl,
5    get_now(Now),
6    buffer_state(BufName,busy),
7    do_buffer_clear(BufName), % clear buffer immediately
8    get_context(Context),
9    ModName:module_request(BufName, Chunk, Context,
         ResChunk,ResState,RelTime),
10   performed_request(BufName, ResChunk, ResState), % save result
         of request
11   Time is Now + RelTime,
12   add_q(Time, 0, do_buffer_request(BufName, Chunk)).
```

In line 8 the current context is calculated and bound in form of a list of chunk names to the variable `Context`.

Listing 4.13: Apply the results of the buffer request

```
1  % apply the results
2  do_buffer_request(BufName, _), buffer(BufName, ModName, _),
       buffer_state(BufName,_), performed_request(BufName,
       ResChunk, ResState) <=>
3    write('performing request: '),write(BufName),nl,
4    (ResState=error,
5    buffer(BufName, ModName, nil),
6    buffer_state(BufName,error) ;
7
8    ResState = free,
9    ResChunk = chunk(ResChunkName,_,_),
10   add_chunk(ResChunk),
11   buffer(BufName, ModName, ResChunkName),
12   buffer_state(BufName,free)).
```

Note that a buffer request has the lowest priority of all buffer actions, as shown in table 4.1. If the request was performed before the buffer modifications and clearings have taken effect, the wrong context would be used for the activation calculations.

Adapting the Retrieval Request With the now defined methods, the overall activation of a chunk can be calculated. The module request for the retrieval buffer as introduced in listing 4.8 can be adapted as follows:

Listing 4.14: Retrieval request with subsymbolic calculations

```
1  module_request(retrieval, chunk(Name,Type,Slots), Context,
       ResChunk, ResState, RelTime) <=>
2    find_chunk(Name,Type,Slots),
3    collect_matches(Res),
4    % trigger activation calculation for matching chunks
       regarding given context
5    calc_activations(Res,Context),
6
7    % find threshold for maximum check
8    get_conf(rt,RT),
9    threshold(RT),
10
11   get_max(MaxChunk,MaxAct),
12
13   % Return resulting chunk, state and time
14   return_chunk(MaxChunk,ResChunk),
15   get_state(ResChunk,ResState),
16   calc_time(MaxAct,RelTime).
```

Find matching chunks (lines 2 and 3) First of all, all matching chunks are searched and saved in a list called `Res`, similarly to the symbolic approach in listing 4.8.

Calculate Activations of the matching chunks (line 5) The activations of the matching chunks in the list `Res` are computed regarding the context that has been handed over by the request.

Get the threshold (lines 8 and 9) The current threshold is retrieved from the configuration and a `threshold` constraint is placed in the store. The next steps will need a threshold constraint present.

Find chunk with highest activation (line 11) The chunk with the highest activation is saved in `MaxChunk`, its activation value in `MaxAct`.

Return a chunk specification (line 14) The request is supposed to return a complete chunk specification in the variable `ResChunk`. This is achieved by `return_chunk`.

Return the resulting state of the buffer (line 15) The resulting state `ResState` of the requested retrieval buffer is `free`, if a matching chunk has been found and `error` if no chunk matches the request:

```
1 get_state(nil,error).
2 get_state(_,free).
```

Return the time the request takes (line 16) The time depends on the activation of the chunk. If no matching chunk has been found, the activation is assumed to be the threshold value. This is already achieved by the maximum calculation as described above. According to equation (2.5), the resulting time is computed as follows:

```
1 calc_time(Act,ResTime) :-
2   get_conf(lf,F),
3   ResTime=F*exp(-Act).
```

Configuration of the Retrieval

ACT-R offers some configuration variables which influence the retrieval process. In the CHR implementation, some of those configuration variables are implemented in the system according to the configuration infrastructure as described in section 4.8.2. In ACT-R, configuration variables are set with the command `(sgp :Var Val)` which sets the value of the variable **`Var`** to `Val`.

Turn on the subsymbolic layer The variable `esc` that can be set to `t` for *true* and `nil` for *false* controls the activation of the subsymbolic layer for retrieval requests. In CHR, the declarative module observes the value of this variable by the observer interface presented in section 4.8.2. Depending on the value of the variable, the constraint `subsymbolic/0` is present or not:

```
1  % subsymbolic layer turned on -> add subsymbolic constraint
2  set_subsymbolic(t), subsymbolic <=>
3    subsymbolic.
4  set_subsymbolic(t) <=>
5    subsymbolic.
6
7  % subsymbolic layer turned off -> remove subsymbolic
       constraint
8  set_subsymbolic(nil), subsymbolic <=>
9    true.
10 set_subsymbolic(nil) <=>
11   true.
```

For each rule only involved in the subsymbolic layer, the `subsymbolic` constraint is added to its kept head, so the rule only fires if the subsymbolic layer is turned on.

Retrieval threshold The retrieval threshold θ is set by the configuration variable `rt`.

Latency factor The latency factor F is set by the configuration variable `lf`.

Proportion of θ and F The configuration system automatically sets the latency factor if the retrieval threshold is set. If an individual value for F should be set, then the automatically set value has to be overwritten after the threshold has been set.

```
1 latency-factor-by-threshold @
2 set_conf(rt,RT) ==>
3   LF is 0.35*exp(RT),
4   set_conf(lf,LF).
```

Decay parameter The decay parameter d is set by the configuration variable `bll`.

4.9.2 Conflict Resolution and Production Utility

If there are competing strategies that match a current state, the production system selects the rule with the highest production utility to fire. This process is called *conflict resolution* in the terminology of production rule systems and is described in the following.

Conflict Resolution

In [Frü09, pp. 151 sqq.] a general implementation of conflict resolution in CHR is described. This approach can be easily adapted to the needs of ACT-R. Replace every production rule

```
1 {buffer tests} \ fire <=> {guard} | {actions},
     conflict_resolution.
```

with two rules

Listing 4.15: Translation scheme for production rules regarding conflict resolution

```
1 delay-name @
2   fire, {buffer tests} ==> {guard} | conflict_set(name).
3 name @
4   {buffer tests}, apply_rule(name) <=> {guard} | {actions},
       conflict_resolution.
```

The first rule adds the matching rule to a conflict set without computing anything, the second rule actually performs the calculations as soon as the `apply_rule` constraint is present.

At the end, add a rule

```
1 no-rule @
2   fire <=> conflict_set([]), choose.
```

As soon as the `fire` constraint is present (so the recognize cycle/conflict resolution process begins), each matching rule adds a `conflict_set/1` constraint with its name. The last rule finishes the recognize cycle by deleting the fire constraint *after* all rules that match had their chance to add a `conflict_set` constraint. Then an empty `conflict_set` constraint, indicated by `[]`, is added. At the end of this phase, the constraint store contains a bunch of `conflict_set` constraints that represent the matching rules plus an empty `conflict_set` constraint. If no rule matches, there is only an empty `conflict_set` constraint in the store.

The last rule also triggers the choosing process by adding the constraint `choose`. The following rules handle the choosing process:

```
1 conflict_set(_) \ conflict_set([]) <=> true.
2
3 find-max-utility @ production_utility(P1,U1),
     production_utility(P2,U2), conflict_set(P1) \
     conflict_set(P2) <=>
4   U1 >= U2 |
5   true.
```

The first rule deletes the empty `conflict_set` constraint, if there has been an applicable production rule which is recognized by other `conflict_set` constraints present.

The second rule assumes, that for each production rule `p` in the procedural memory there is a `production_utility(p,u)` constraint that holds the utility value `u` of the production `p`. If there are two `conflict_set` constraints in the store, the one with the higher utility value will be kept and the other will be removed from the store.

If the rules have been applied to exhaustion, there is only one `conflict_set` constraint in the store – either an empty one ore one with the name of a rule. The following rules handle the choosing process:

```
choose, conflict_set([]) <=>
  no_rule.

choose @ choose, conflict_set(P) <=>
  P \== [] |
  get_now(Now),
  Time is Now + 0.05,
  add_q(Time,0,apply_rule(P)).
```

The first rule is only applicable, if there were no matching rules and the empty conflict-set, indicated by the `conflict_set([])` constraint, is still present. Then this fact is indicated by a `no_rule` constraint.

In the second rule, the firing of the last remaining rule is scheduled 50 ms from the current time, as it is described in section 2.1.3. The event is the `apply_rule(P)` constraint, which leads the second rule in listing 4.15 to fire, which performs the actions of the rule.

When the chosen production rule is applied, its actions are performed and eventually a `conflict_resolution` constraint is added to the store. This constraint leads the next conflict resolution event to being scheduled:

```
now(Time) \ conflict_resolution <=> add_q(Time,-10,fire).
```

The event is scheduled at the current time with very low priority, i.e. a priority lower than all requests, so all the actions of the rule have had the chance to be executed. If there was no matching rule, the `no_rule` constraint is in the store. This leads the next conflict resolution event to be scheduled after the next event (which may lead to a change of the system state):

```
no_rule <=> after_next_event(fire).
```

Note that the described method of implementing the conflict resolution process of ACT-R, matches exactly the description in the reference manual:

> “The procedural module will automatically schedule conflict-resolution events. The first one is scheduled at time 0 and a new one is scheduled after each production fires. If no production is selected during a conflict-resolution event

> then a new conflict-resolution event is scheduled to occur after the next change occurs." [Botb, p. 156]

In [Frü09], the `conflict_set` and `apply_rule` constraints also have the values of the variables that have been bound in the matching of the rule in the collecting phase:[12]

```
1  delay-name @
2    fire,
3    {buffer tests}
4    ==> {guard} |
5    conflict_set(rule(name, {Variables in the head of the rule})).
6  name @
7    {buffer tests},
8    apply_rule(rule(name, {Variables in the head of the rule}))
9    <=> {guard} |
10   {actions}, conflict_resolution.
```

This is due to the fact that during the conflict resolution process other rules may have changed the constraint store and the rule might not be applicable anymore. However, in ACT-R the procedural module is a serial bottleneck, so no rules that change the state of the buffer system can be applied during the conflict resolution. Additionally, in [Frü09] the rules that were in the conflict set but have not been applied, remain in the conflict set for the next cycle and are applied, if they have at some point the highest priority in the set and are still applicable (ensured by the bound variables in `apply_rule`). In ACT-R, this does not play a role, since the recognize-act cycle is defined serially and only one rule is applied in each cycle. In the next cycle, all rules are checked again for matching heads and the computations are performed on the new values. Note that the problem of trivial non-termination of propagation rules described in [AFS13, p. 5], which has to be considered when changing the operational order of rule applications in CHR, does not play a role for ACT-R, since it does not implement propagation rules.

Computing the Utility Values

As described in section 2.2.2, rules can have a certain amount of reward that can be distributed among all rules that have been applied since the last reward has been distributed. The reward a rule can distribute, can be saved in a `reward/2` constraint. If a

[12] The example is slightly modified from the original in [Frü09]: In the original, the name of the rule does not play a role and the priorities are known in advance

rule is applied, the time of application can be saved in a `to_reward/2` constraint which states that the rule in the constraint has been applied and therefore receives a part of the next reward as soon as it occurs:

```
1 apply_rule(P) ==> P \== [] | get_now(Now), to_reward(P,Now).
```

Note that the `apply_rule/1` constraint from the last section is used to determine when a rule is fired. When a rule that can distribute a reward is applied, the reward is triggered:

```
1 apply_rule(P), reward(P,R) ==>
2   P \== [] |
3   trigger_reward(R).
```

This will reward all rules that have a `to_reward/2` constraint in the store which leads to a new production utility value:

```
1 trigger_reward(R) \ production_utility(P,U),
     to_reward(P,FireTime) <=>
2   calc_reward(R,FireTime,Reward),
3   get_conf(alpha,Alpha),
4   NewU is U + Alpha*(Reward-U),
5   production_utility(P,NewU).
6
7 calc_reward(R,FireTime,Reward) :-
8   get_now(Now),
9   Reward is R - (Now-FireTime).
```

Note that in line 4 the utility is adapted by the utility learning rule as shown in equation (2.7). The reward the rule receives is calculated by the Prolog predicate in lines 7 sqq.: The more time passed since the rule application, the less is the reward of the rule.

In the end, there has to be a rule that cleans up the reward trigger, after all the rules have been rewarded. This ensures, that the next `to_reward` constraints are not immediately consumed by the last reward trigger:

```
1 trigger_reward(_) <=> true.
```

Configuration of the Conflict Resolution

There are some methods to influence the conflict resolution process using the ACT-R command `(spp ...)` that sets values for individual production rules. This command is translated into CHR as described in section 4.8. Additionally, there are some configuration variables that can be set – as described in section 4.8.2 – by the command `(sgp ...)`:

Setting the utility of a rule Utilities can be set statically by the user by the ACT-R command `(spp P :u U)` which sets the utility of the rule `P` to `U`.

Setting the reward of a rule The amount of reward a rule can distribute is set by the user. By default, no rules have a reward to trigger. With the ACT-R command `(spp P :reward R)`, the reward of the rule `P` is set to `R`. This command is also implemented among the standard lisp methods.

Default values and turning the utility mechanisms off At the moment, there is no command in the CHR implementation that allows to turn off the conflict resolution mechanisms: The system sets a default utility value for each production rule and no rewards in the initialization process (the rule `init @ run <=> ...` as described in section 4.6). The user is able to set other initial utility values and rewards for particular rules. If no rewards are set, the utility values will remain at their initial values. Hence, if the user does not set any utilities or rewards, each rule will have the same utility value that does not change. If there are competing rules, the conflict resolution process will pick one. This can be regarded as turned off utility mechanisms.

Learning rate The learning rate α is set by the command `(sgp :alpha A)` that sets the configuration variable `alpha` to `A`.

Public Methods of the Conflict Resolution

Since the scheduler has to have access to the constraints `fire/0` and `apply_rule/1` to control the timing conflict resolution, those constraints are added to the *ICore* interface as described in section 4.4.4. Additionally, the constraint `set_production_utility/2` is added to the *ICore* interface, because the utilities may be set from outside the procedural module initially.

4.10 Compiler

With the implementation of the basic ACT-R concepts, a simple compiler has been created to automate the translation of Lisp ACT-R rules to CHR rules according to the translation

schemes defined in section 4.3. The compiler has been built using Prolog and CHR. However, it still lacks suitable translation of some of the details in the procedural module presented in section 4.3.3.

4.10.1 Basic Idea

The basic idea of the compiler is that it gets an ACT-R model definition written in the Lisp-like ACT-R syntax and produces the correspondent CHR rules. The model should be executable by just loading the translation and typing in the query `run`. Hence, the translated model should somehow load the CHR framework simulating ACT-R, for instance the modules, the buffer system, the scheduler, etc.

4.10.2 Compiling

The compiler consists of three parts: A tokenizer, a parser and an actual translation component. Those three parts are described individually in the following sections.

Tokenizer

The tokenizer gets the file with the model definition as input and does some preprocessing on it: From the input – a sequence of characters – it builds a list of tokens. Tokens are separated by white-space (or by special characters) and can be one of the following character sequences in the input:

Special Characters are individual tokens and are defined as all allowed characters that are not letters, numbers or white-space. In ACT-R, the symbols `!, (, ), +, -, =, >, ?` are considered special characters. Some special characters of ACT-R are missing in this list, but are not implemented in the compiler yet.

Keywords are treated like a special character, but contain more than one symbol. In ACT-R, one of the keywords is **==>**, separating the LHS from the RHS of a production rule.

Identifiers Every character sequence which starts with a letter and contains only letters or numbers. An identifier ends, if a special character or a white-space is read from the input. The `-` character is an exception, because it can be used within identifiers, although it is a special character.
Example: `count, addition-fact, chunk1`

Numbers Every word that only contains digits `0-9`. As for every token, a number ends, if a white-space, a special character or a keyword occurs.

Example 4.10 (Tokenizing some input)**.** *The input*

```
(p name
  =buffer>
    s  v
==>
  +buffer>
    s  new-value
  -buffer2>
)
```

produces the output list

```
['(',p,name,=,buffer,>,s,v,==>,+,buffer,>,s,new-value,
   -,buffer2,>,')']
```

The basic idea for the implementation of such a tokenizer is described in [LM, pp. 19 sqq.]. It basically follows a deterministic finite automaton with the state transitions as illustrated in figure 4.4.

In Prolog, text input can be handled as a list of ASCII character codes. To classify those characters according to the list above, some predicates and facts can be used:

```
digit(D) :- 46 < D, D < 59.
special_char(33). %!
special_char(40). %(
...
```

The tokenizer gets a list of such character codes and returns a list of tokens:

```
getTokens([], []).

getTokens([X|Xs], Ts) :-
  white_space(X),
  getTokens(Xs,Ts).

```

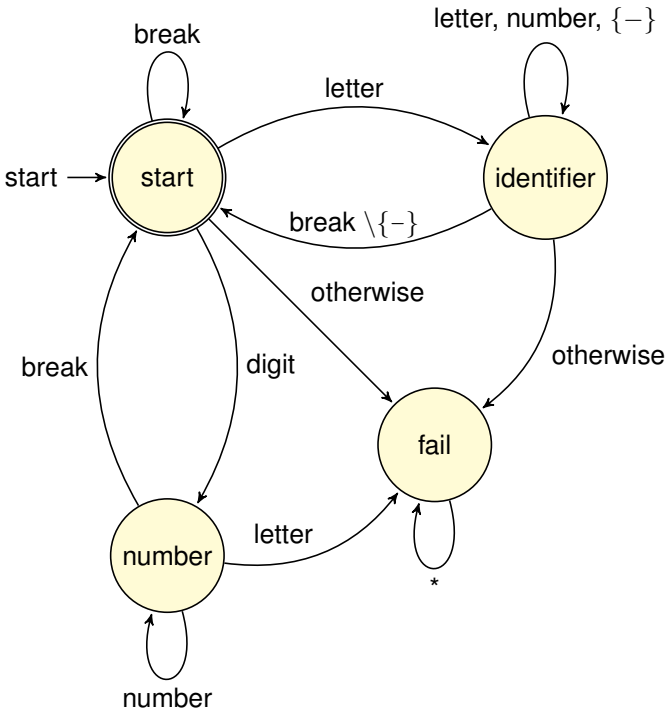


Figure 4.4: The finite automaton implemented by the tokenizer. The labels of the arrows are sets of symbols, e.g. *letter* denotes all characters that are letters. The class *break* is the union of the classes *white-space*, *keyword* and *special character*.

```
7 getTokens(In, [T|Ts]) :-
8   getToken(start, In, Rest, T), !,
9   getTokens(Rest, Ts).
```

The first predicate is the base case. The second rule just drops the white-space and calls the tokenizing predicate again with the rest of the list. The last predicate only applies if the other predicates failed. It starts the inspection of the symbols by calling the predicate `getToken(Q,In,Rest,T)` which gets the `start` state as argument `Q` and the current input list `In` and returns the rest of the list and the result token `T`, after some input has been processed. The `getToken` predicate starts in the start state and then decides deterministically which of the state transitions is chosen by inspecting the first character from the input list. It then remains in the state according to the state transition diagram in figure 4.4 until it reads a character that forces it to leave the state. Then it binds the remaining input to the list `Rest` and all the characters that have been consumed in this state to the result token `T`. For the identifiers, the following implementation is used:

```
1  getToken(start, [X|Xs],Rest,LT) :-
2    letter(X),
3    % find rest of word
4    getToken(identifier, Xs,Rest,[X],T),
5    % convert uppercase letters to lowercase
6    downcase_atom(T,LT).
7
8  getToken(identifier, [C|R],Rest,S,T) :-
9    (letter(C);
10     digit(C);
11     minus(C)),
12   getToken(identifier,R,Rest,[C|S],T).
13
14 getToken(identifier,Rest,Rest,S,W) :-
15   % tokens are accumulated in wrong order
16   reverse(S,S1),
17   atom_chars(W,S1).
```

The first predicate checks, if the list starts with a letter. It then changes the state to `identifier`, because only identifiers start with a letter. The predicate `getToken/5` has an accumulator list as an additional argument to the original `getToken/4` predicate. It remains in the state identifier, if it reads a letter, a digit or the symbol - and accumulates the read symbol in a list (second predicate). If the second predicate cannot be applied any more, because the input list starts with a symbol that is not allowed for an identifier, the third predicate binds this remaining input to the list `Rest` and returns the reversed accumulated characters as token. The first predicate then changes the case of the token to lowercase and returns it to `getTokens`, which changes the state to `start` and begins the next cycle, until the complete input has been consumed. The implementation for digits is analogous.

The arrow keyword is handled separately by a predicate which, as shown in the following listing, consumes the three characters **==>** at once. This predicate has to be the first to be checked.

```
1  getToken(start, [61,61,62|Rest], Rest, '==>').
```

Note that the processing fails, if a not-allowed symbol is read from input, so all of the `getToken` predicates fail.

Parser

The parser takes a list of tokens as input and returns a parse tree. It is implemented by the use of Definite Clause Grammar Rules (DCG rules), which allow to directly write grammar-like rules like:

```
1 s --> [a], s, [b].
2 s --> [].
```

This program recognizes the language $\{a^n b^n \mid n \leq 0\}$. The query `s([a,a,b,b],[])` returns `true`, whereas `s([a,b,b],[])` returns `false`. An introduction to DCG rules is given in [Ogb].

Hence, the grammar in section 4.3.3 can be translated almost directly to such DCG rules. However, this only returns if the input list is a word of the defined grammar. To get a parse-tree out of the parsing process, the rules can be extended according to the following scheme:

```
1 a(a(B,C)) --> b(B), c(C).
2 b(b(B)) --> ...
3 c(c(C)) --> ...
```

I.e., the path taken by the program is accumulated in an argument as a term. A result of the example could be `a(b(...),c(...))`.

Translation Component

In the translation component, CHR rules are generated from the parse-tree according to the methods discussed in this chapter. For the output, a writer that has been presented in the exercises of the *Rule-Based Programming* lecture at the University of Ulm by Amira Zaki [Chr] has been used.

The translation component also organizes the complete parsing process: It loads the input file, calls the tokenizer and the parser and then recursively analyzes the resulting parsing tree. Whenever an ACT-R production rule has been processed completely, it produces a CHR rule using the writer.

For the variables in the rules, it uses a symbol table for each rule to ensure that the same variable is always translated to the same Prolog variable in the result.

4.10.3 Limitations of the Current Implementation

The current implementation of the compiler can be improved at some points. First of all, it can only process a subset of the grammar that actually is a subset of the simplified grammar shown in section 4.3.3. For example, it is only able to parse the negation slot-modifier, the others are not implemented yet, but can be added easily in the future.

The formulation of the grammar also differs from the original definition, so for future work, it would be advantageous to assimilate the two grammars. Additionally, lists have been implemented very complicated in the DCG rule grammar and could be simplified by using actual Prolog lists instead of constructs like `tests --> test, tests; test`, which make the parse tree very complicated. Instead, the tree could be simplified to `tests([`t_1`, ... , `t_n`])`.

Another problem is that duplicate slot tests as shown in section 4.3.3 on page 57 cannot be handled by the compiler yet.

The recursive approach of the translation component can get very confusing the more rules are added, because a lot of help data has to be accumulated and passed between the different recursion levels (sometimes upwards, sometimes downwards). Another approach could be to store the data that is parsed at a certain level in the constraint store. Eventually, some CHR rules put the parts together in the end.

Finally, the compiler does not produce any error messages, but only fails silently if there are syntactical problems in the input. For usability, it would be helpful to add meaningful error messages.

5 Example Models

In this chapter, two ACT-R models are translated into CHR and the results are discussed. The examples are taken from *The ACT-R Tutorial* [Actb].

5.1 The Counting Model

The first model is an extension of the example presented in section 2.1.6. The code has been published in unit 1 of *The ACT-R Tutorial* [Actb] as shipped with the source code of the vanilla ACT-R 6.0 implementation. Basically, the counting process is modeled by a set of static facts that a person has learned and retrieves for the counting process from declarative memory. The following model definition is discussed:

Listing 5.1: ACT-R code of the counting example

```
(define-model count

  (chunk-type count-order first second)
  (chunk-type count-from start end count)

  (add-dm
   (b ISA count-order first 1 second 2)
   (c ISA count-order first 2 second 3)
   (d ISA count-order first 3 second 4)
   (e ISA count-order first 4 second 5)
   (f ISA count-order first 5 second 6)
   (first-goal ISA count-from start 2 end 4)
   )

  (P start
     =goal>
        ISA         count-from
```

```
18          start       =num1
19          count       nil
20     ==>
21        =goal>
22          count       =num1
23        +retrieval>
24          ISA         count-order
25          first       =num1
26    )
27
28    (P increment
29        =goal>
30          ISA         count-from
31          count       =num1
32         - end        =num1
33        =retrieval>
34          ISA         count-order
35          first       =num1
36          second      =num2
37     ==>
38        =goal>
39          count       =num2
40        +retrieval>
41          ISA         count-order
42          first       =num2
43        !output!      (=num1)
44    )
45
46    (P stop
47        =goal>
48          ISA         count-from
49          count       =num
50          end         =num
51     ==>
52        -goal>
53        !output!      (=num)
54    )
55
56    (goal-focus first-goal)
57  )
```

First of all, in line 1 the model definition is initiated and a model name is given. Line 3 sq. add the two necessary chunk-types: A chunk-type for the goal-chunks and a chunk-type for the actual declarative data. These chunk-type definitions are global to the system, i.e. they are added to all chunk stores. The goal-chunks have the slots `start` and `end` that encode from which number the counting process should start and where it should end. The value in the slot `count` saves the current number of the counting process, analogously as in section 2.1.6. Then the chunks are added from line 6 to 13. They are representations of the order of the natural numbers from one to six. The last chunk is the goal-chunk. Note that only the start and end numbers are specified, the current number will be set to `nil`.

The first production rule `start` in line 15 sqq. can only be applied if the goal has an empty `count` slot, but an actual value in the `start` slot. This will be valid for the initial state as explained later. The production states a request for the first count-fact with the start number in its `first` slot and sets the current number in the goal to the start number. The other slots in the goal buffer remain the same.

In line 28 sqq. the main rule `increment` is defined. It assumes that a count-order fact which matches the current number in the goal has been retrieved. Additionally, the counting process must not end with this number, indicated by the negated slot test on the slot `end`. The action part then states a new request for the next count-fact and increments the current number. Once the rule is applicable, it will be applicable as long there are count-facts in the declarative memory and the specified end has not been reached yet.

The last production rule `stop` in line 46 is applicable, as soon the `increment` rule cannot be applied anymore due to the fact that the specified end of the counting process has been reached. Then, the last number will be printed and the goal buffer will be cleared.

The last function call in line 56 is an initialization method, which simply puts the previously defined chunk `first-goal` into the goal buffer. This leads to an initial state where the rule `start` is applicable. The output of the model is:

```
1      0.000   GOAL            SET-BUFFER-CHUNK GOAL FIRST-GOAL
2      0.000   PROCEDURAL      CONFLICT-RESOLUTION
3      0.050   PROCEDURAL      PRODUCTION-FIRED START
4      0.050   PROCEDURAL      CLEAR-BUFFER RETRIEVAL
5      0.050   DECLARATIVE     START-RETRIEVAL
6      0.050   DECLARATIVE     RETRIEVED-CHUNK C
7      0.050   DECLARATIVE     SET-BUFFER-CHUNK RETRIEVAL C
```

```
8       0.050   PROCEDURAL        CONFLICT-RESOLUTION
9       0.100   PROCEDURAL        PRODUCTION-FIRED INCREMENT
10  2
11      0.100   PROCEDURAL        CLEAR-BUFFER RETRIEVAL
12      0.100   DECLARATIVE       START-RETRIEVAL
13      0.100   DECLARATIVE       RETRIEVED-CHUNK D
14      0.100   DECLARATIVE       SET-BUFFER-CHUNK RETRIEVAL D
15      0.100   PROCEDURAL        CONFLICT-RESOLUTION
16      0.150   PROCEDURAL        PRODUCTION-FIRED INCREMENT
17  3
18      0.150   PROCEDURAL        CLEAR-BUFFER RETRIEVAL
19      0.150   DECLARATIVE       START-RETRIEVAL
20      0.150   DECLARATIVE       RETRIEVED-CHUNK E
21      0.150   DECLARATIVE       SET-BUFFER-CHUNK RETRIEVAL E
22      0.150   PROCEDURAL        CONFLICT-RESOLUTION
23      0.200   PROCEDURAL        PRODUCTION-FIRED STOP
24  4
25      0.200   PROCEDURAL        CLEAR-BUFFER GOAL
26      0.200   PROCEDURAL        CONFLICT-RESOLUTION
27      0.200   ------            Stopped because no events left to
        process
```

The source file in listing 5.1 can be translated by the provided CHR compiler which yields the following result:

Listing 5.2: Auto-generated CHR code of the counting example

```
1  :- include('actr_core.pl').
2  :- chr_constraint run/0, fire/0.
3
4  delay-start @
5    fire,
6    buffer(goal,_,A),
7      chunk(A,count-from),
8      chunk_has_slot(A,start,B),
9      chunk_has_slot(A,count,nil)
10 ==>
11   B\==nil |
12   conflict_set(start).
13
```

```
start @
  buffer(goal,_,A),
    chunk(A,count-from),
    chunk_has_slot(A,start,B),
    chunk_has_slot(A,count,nil)
  \ apply_rule(start)
<=>
  B\==nil |
  buffer_change(goal,chunk(_,_,[ (count,B)])),
  buffer_request(retrieval,chunk(_,count-order,[ (first,B)])),
  conflict_resolution.

delay-increment @
  fire,
  buffer(goal,_,A),
    chunk(A,count-from),
    chunk_has_slot(A,count,C),
    chunk_has_slot(A,end,D),
  buffer(retrieval,_,B),
    chunk(B,count-order),
    chunk_has_slot(B,first,C),
    chunk_has_slot(B,second,E)
==>
  C\==nil,
  D\==C,
  E\==nil |
  conflict_set(increment).

increment @
  buffer(goal,_,A),
    chunk(A,count-from),
    chunk_has_slot(A,count,C),
    chunk_has_slot(A,end,D),
  buffer(retrieval,_,B),
    chunk(B,count-order),
    chunk_has_slot(B,first,C),
    chunk_has_slot(B,second,E)
  \ apply_rule(increment)
<=>
  C\==nil,
```

```
54    D\==C,
55    E\==nil |
56    buffer_change(goal,chunk(_,_,[ (count,E)])),
57    buffer_request(retrieval,chunk(_,count-order,[ (first,E)])),
58    output(C),
59    conflict_resolution.
60
61  delay-stop @
62    fire,
63    buffer(goal,_,A),
64      chunk(A,count-from),
65      chunk_has_slot(A,count,B),
66      chunk_has_slot(A,end,B)
67  ==>
68    B\==nil |
69    conflict_set(stop).
70
71  stop @
72    buffer(goal,_,A),
73      chunk(A,count-from),
74      chunk_has_slot(A,count,B),
75      chunk_has_slot(A,end,B)
76    \ apply_rule(stop)
77  <=>
78    B\==nil |
79    buffer_clear(goal),
80    output(B),
81    conflict_resolution.
82
83  init @
84  run <=> true |
85    set_default_utilities([stop,increment,start]),
86    add_buffer(retrieval,declarative_module),
87    add_buffer(goal,declarative_module),
88    lisp_chunktype([chunk]),
89    lisp_chunktype([count-order,first,second]),
90    lisp_chunktype([count-from,start,end,count]),
91    lisp_adddm([[b,isa,count-order,first,1,second,2],
92      [c,isa,count-order,first,2,second,3],
93      [d,isa,count-order,first,3,second,4],
```

```
94      [e,isa,count-order,first,4,second,5],
95      [f,isa,count-order,first,5,second,6],
96      [first-goal,isa,count-from,start,2,end,4]]),
97    lisp_goalfocus([first-goal]),
98    now(0),
99    conflict_resolution,
100   nextcyc.
101
102 no-rule @
103 fire <=>
104   true |
105   conflict_set([]),
106   choose.
```

First of all, it is interesting to note, that there are some more CHR rules than production rules in the original code in listing 5.1. One very obvious extension is the rule `init` in line 83. This rule initializes the model according to section 4.6. First, it sets the default utilities for each of the production rules and creates the used buffers in the buffer system. Then, the Lisp functions from the original model definitions are called in their CHR versions, according to section 4.8. Those are the methods which create chunk-types and add the initial declarative knowledge to the declarative module. Note that also the artificial chunk-type `chunk` with no slots is created. The last Lisp call moves the `first-goal` chunk to the goal buffer. Finally, the current time is set to zero, a conflict resolution event is scheduled in the queue (according to section 4.9.2) and the recognize-act cycle is started by `nextcyc`, which will dequeue the first event from the scheduler as described in section 4.7.2.

Furthermore, each production rule from the original module has two correspondent CHR rules in the translation due to the conflict resolution process described in section 4.9.2. The first rule delays the execution by adding the rule – if applicable – to the conflict set as soon as a new conflict resolution event (represented by the CHR constraint `fire`) has been triggered. The second rule actually performs the actions of the production rule if it has been chosen by the conflict resolution process (indicated by the constraint `apply_rule/1`). After the rule application, each rule schedules the next conflict resolution event by adding `conflict_resolution` to the store. The guards of the production rules check that each tested slot actually has a value (so its value is not `nil`) as described in section 4.3.3 (see page 62). In the rule `increment`, for example, a negated slot test is translated to a guard check according to the pattern presented in section 4.3.3 (see page 59).

The rule `no-rule` in line 102 is the last rule tested in the collecting process of the conflict resolution. It removes the `fire` constraint and triggers the choosing process after it has added an empty rule to the conflict set. This is important for the choosing process to detect if no rule was applicable in this conflict resolution phase.

The output of the model is the following (pretty printed by hand):

```
1  ?- run.
2  0       ... calling event: do_conflict_resolution
3              going to apply rule start
4  0.05    ... calling event: apply_rule(start)
5              firing rule start
6  0.05    ... calling event:
      do_buffer_change(goal,chunk(_G17029,_G17030,[ (count,2)]))
7  0.05    ... calling event:
      start_request(retrieval,chunk(_G17476,count-order,[
      (first,2)]))
8              Started buffer request retrieval
9              clear buffer retrieval
10 0.05    ... calling event: do_conflict_resolution
11             No rule matches -> Schedule next conflict resolution
                  event
12 1.05    ... calling event:
      do_buffer_request(retrieval,chunk(_G17476,count-order,[
      (first,2)]))
13             Retrieved chunk c
14             Put chunk c into buffer
15 1.05    ... calling event: do_conflict_resolution
16             going to apply rule increment
17 1.1:0   ... calling event: apply_rule(increment)
18             firing rule increment
19 output:2
20 1.1     ... calling event:
      do_buffer_change(goal,chunk(_G33694,_G33695,[ (count,3)]))
21 1.1     ... calling event:
      start_request(retrieval,chunk(_G34139,count-order,[
      (first,3)]))
22             Started buffer request retrieval
23             clear buffer retrieval
24 1.1     ... calling event: do_conflict_resolution
```

```
25            No rule matches -> Schedule next conflict resolution
                  event
26  2.1    ... calling event:
    do_buffer_request(retrieval,chunk(_G34139,count-order,[
    (first,3)]))
27            Retrieved chunk d
28            Put chunk d into buffer
29  2.1    ... calling event: do_conflict_resolution
30            going to apply rule increment
31  2.15   ... calling event: apply_rule(increment)
32            firing rule increment
33  output:3
34  2.15   ... calling event:
    do_buffer_change(goal,chunk(_G25849,_G25850,[ (count,4)]))
35  2.15   ... calling event:
    start_request(retrieval,chunk(_G26294,count-order,[
    (first,4)]))
36            Started buffer request retrieval
37            clear buffer retrieval
38  2.15   ... calling event: do_conflict_resolution
39            going to apply rule stop
40  2.1999 ... calling event: apply_rule(stop)
41            firing rule stop
42  output:4
43  2.1999 ... calling event: do_buffer_clear(goal)
44            clear buffer goal
45  2.1999 ... calling event: do_conflict_resolution
46            No rule matches -> Schedule next conflict resolution
                  event
47  3.15   ... calling event:
    do_buffer_request(retrieval,chunk(_G26294,count-order,[
    (first,4)]))
48            performing request: retrieval
49            Retrieved chunk e
50            Put chunk e into buffer
51  3.15   ... calling event: do_conflict_resolution
52            No rule matches -> Schedule next conflict resolution
                  event
53            No more events in queue. End of computation.
```

Note that the output is very similar to the original output, especially the order of rule applications and events. However, the timings are not accurate yet, since some constants and special cases are different from the original implementation. This can be fixed and does not really harm the theory.

To demonstrate the subsymbolic layer, the original code is extended as follows:

```
(define-model count
  (sgp :esc t)

  (chunk-type count-order first second)
  (chunk-type goal-chunk goal start end count)

  (add-dm
  ...
  (d ISA count-order first 3 second 4)
  (d1 ISA count-order first 3 second 5)
  ...
  (second-goal ISA goal-chunk goal training1)
  )

  (P train1
    =goal>
      ISA           goal-chunk
      goal          training1
  ==>
    =goal>
      goal          training2
    +retrieval>
      ISA           count-order
      first         3
      second        4
  )

  (P train2
    =goal>
      ISA           goal-chunk
      goal          training2
    =retrieval>
      ISA           count-order
```

```
34      first          3
35      second         4
36    ==>
37      =goal>
38        ISA          goal-chunk
39        start        2
40        end          4
41        goal         count
42      -retrieval>
43    )
44
45    (P start
46        { defined as in listing 5.1 }
47    )
48
49    (P increment
50      { defined as in listing 5.1 }
51    )
52
53    (P incrementx
54      =goal>
55          ISA          goal-chunk
56          goal         count
57          count        =num1
58        - end          =num1
59      =retrieval>
60          ISA          count-order
61          first        =num1
62          second       =num2
63    ==>
64      -goal>
65      !output!       (wrong)
66    )
67
68    (P stop
69      { defined as in listing 5.1 }
70    )
71
72    (goal-focus second-goal)
73
```

```
74    (spp increment :u 8 incrementx :u 0)
75    (spp stop :reward 15)
76  )
```

The subsymbolic layer is turned on and chunk `d1` which encodes a false count fact is added. Additionally, there are two training rules which just retrieve the fact `d` to increase its activity and then reset the state to the initial state of the model in listing 5.1. The goal is set to a chunk, which leads the first training rule to match at first. Furthermore, a broken rule which matches the same context as the original `increment` rule is added. The initial utility of the correct `increment` rule is set to 8, whereas the corrupted rule gets an initial value of 0. Additionally, the reward the rule `stop` can distribute is set to 15, so reaching the final state is rewarded and all rules which lead to that state get a certain amount of this reward. The full code can be found in appendix B.2.

When executing the resulting CHR model, this yields the following output (times have been cut after four digits):

```
1  ?- run.
2  0      ... calling event: do_conflict_resolution
3             going to apply rule train1
4  0.05   ... calling event: apply_rule(train1)
5             firing rule train1
6  0.05   ... calling event:
      do_buffer_change(goal,chunk(_G26126,_G26127,[
      (goal,training2)]))
7  0.05   ... calling event:
      start_request(retrieval,chunk(_G26573,count-order,[
      (first,3), (second,4)]))
8             Started buffer request retrieval
9             clear buffer retrieval:nil
10 0.05   ... calling event: do_conflict_resolution
11            No rule matches -> Schedule next conflict resolution
                 event
12 0.0974 ... calling event:
      do_buffer_request(retrieval,chunk(_G26573,count-order,[
      (first,3), (second,4)]))
13            performing request: retrieval
14            Retrieved chunk d
15            Put chunk d into buffer
16 0.0974 ... calling event: do_conflict_resolution
```

```
17            going to apply rule train2
18  0.1474 ... calling event: apply_rule(train2)
19            firing rule train2
20  0.1474 ... calling event:
       do_buffer_change(goal,chunk(_G44291,goal-chunk,[ (start,2),
       (end,4), (goal,count)]))
21  0.1474 ... calling event: do_buffer_clear(retrieval)
22            clear buffer retrieval:d
23  0.1474 ... calling event: do_conflict_resolution
24            going to apply rule start
25  0.1974 ... calling event: apply_rule(start)
26            firing rule start
27  0.1974 ... calling event:
       do_buffer_change(goal,chunk(_G26238,_G26239,[ (count,2)]))
28  0.1974 ... calling event:
       start_request(retrieval,chunk(_G26683,count-order,[
       (first,2)]))
29            Started buffer request retrieval
30            clear buffer retrieval:nil
31  0.1974 ... calling event: do_conflict_resolution
32            No rule matches -> Schedule next conflict resolution
                  event
33  0.2575 ... calling event:
       do_buffer_request(retrieval,chunk(_G26683,count-order,[
       (first,2)]))
34            performing request: retrieval
35            Retrieved chunk c
36            Put chunk c into buffer
37  0.2575 ... calling event: do_conflict_resolution
38            going to apply rule increment
39  0.3075 ... calling event: apply_rule(increment)
40            firing rule increment
41  output:2
42  0.3075 ... calling event:
       do_buffer_change(goal,chunk(_G44015,_G44016,[ (count,3)]))
43  0.3075 ... calling event:
       start_request(retrieval,chunk(_G44460,count-order,[
       (first,3)]))
44            Started buffer request retrieval
45            clear buffer retrieval:c
```

```
0.3075 ... calling event: do_conflict_resolution
           No rule matches -> Schedule next conflict resolution
               event
0.3657 ... calling event:
   do_buffer_request(retrieval,chunk(_G44460,count-order,[
   (first,3)]))
           performing request: retrieval
           Retrieved chunk d
           Put chunk d into buffer
0.3657 ... calling event: do_conflict_resolution
           going to apply rule increment
0.4157 ... calling event: apply_rule(increment)
           firing rule increment
output:3
0.4157 ... calling event:
   do_buffer_change(goal,chunk(_G68222,_G68223,[ (count,4)]))
0.4157 ... calling event:
   start_request(retrieval,chunk(_G68667,count-order,[
   (first,4)]))
           Started buffer request retrieval
           clear buffer retrieval:d
0.4157 ... calling event: do_conflict_resolution
           going to apply rule stop
0.4657 ... calling event: apply_rule(stop)
           firing rule stop
           triggered reward for rule: stop
           triggered reward for rule: increment
           triggered reward for rule: increment
           triggered reward for rule: start
           triggered reward for rule: train2
           triggered reward for rule: train1
           reward triggered by rule stop
output:4
0.4657 ... calling event: do_buffer_clear(goal)
           clear buffer goal:first-goal
0.4657 ... calling event: do_conflict_resolution
           No rule matches -> Schedule next conflict resolution
               event
```

```
77 0.5029 ... calling event:
       do_buffer_request(retrieval,chunk(_G36175,count-order,[
       (first,4)]))
78             performing request: retrieval
79             Retrieved chunk e
80             Put chunk e into buffer
81 0.5029 ... calling event: do_conflict_resolution
82             No rule matches -> Schedule next conflict resolution
                   event
83             No more events in queue. End of computation.
```

Due to the training of chunk `d` which increased the activation of this chunk according to the base-level learning equation, the other matching chunk `d1` is not being retrieved in the computational process. Furthermore, the rule `increment` is applied instead of `incrementx`, although both are matching the context, due to the higher intitial utility value of `increment`. The reward of the rule `stop` leads to the following utilities at the end of the computation (taken from the final CHR store):

```
1 production_utility(train1,6.916852474603892)
2 production_utility(train2,6.9363461811027785)
3 production_utility(start,6.946346181102779)
4 production_utility(increment,10.480367881320248)
5 production_utility(stop,7.0)
6 production_utility(incrementx,0)
```

All rules except from `increment` and `incrementx` have been initialized with an utility value of 5. The rules `start`, `increment`, `stop` and the training productions have a higher utility value than in the beginning, whereas the other rules have their initial values. This is the case because only the rules `start` and `increment` have lead to the final state and therefore have received the reward distributed by the `stop` rule.

Note that the times in the output may vary from the execution of the same model in the vanilla implementation, since some constants are set differently.

5.2 Modeling a Taxonomy of Animals and Their Properties

In this example, a taxonomy of categories and properties, illustrated in figure 5.1, is modeled [Actb, unit 1, pp. 24 sqq.]. The goal chunks of this model represent queries like "Is a canary a bird?" or "Is a shark dangerous?".

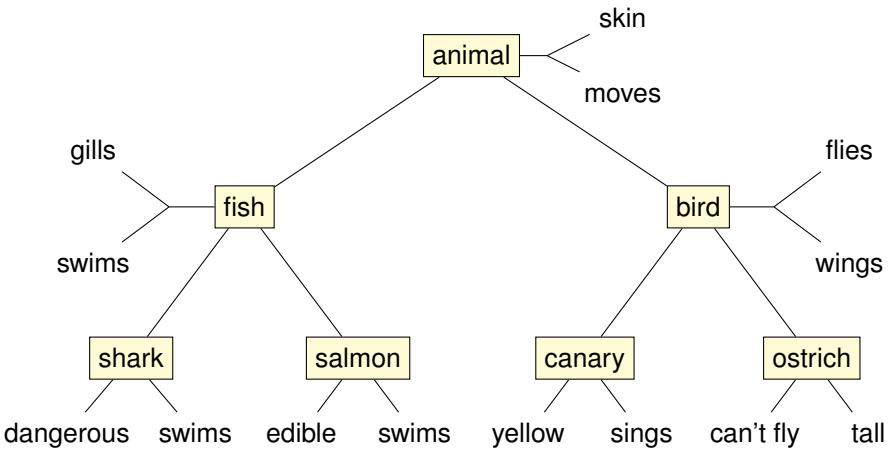


Figure 5.1: An example taxonomy of categories and properties. The categories are marked by rectangles, the pure text nodes in the tree are properties. For example, a *shark* is member of the category *fish* and has the direct properties *dangerous* and *swims* and inherits the properties *gills*, *swims*, *moves* and *skin* from its parent categories.

Chunks which encode a property of an object are of the form illustrated in figure 5.2(a): Each property chunk has a slot *object* which encodes the name of the object. The slot *attribute* holds the name of the attribute like for example *dangerous* or *locomotion*. In the *value* slot, the value of the attribute is defined. The value of the attribute *locomotion* is *swimming* in this example. The chunk in figure 5.2(b) encodes the membership of the object *shark* in the category *fish*.

To answer a query like "Is a canary an animal?" which cannot be answered directly from a *property* chunk, the model must support a recursive search in the tree. The full model code and translation can be found in appendix B.3. The model uses the concept of duplicate slot tests as described in section 4.3.3 (see page 57). However, the compiler cannot handle those tests in the current version and hence the compiled model has to be edited using the pattern from section 4.3.3:

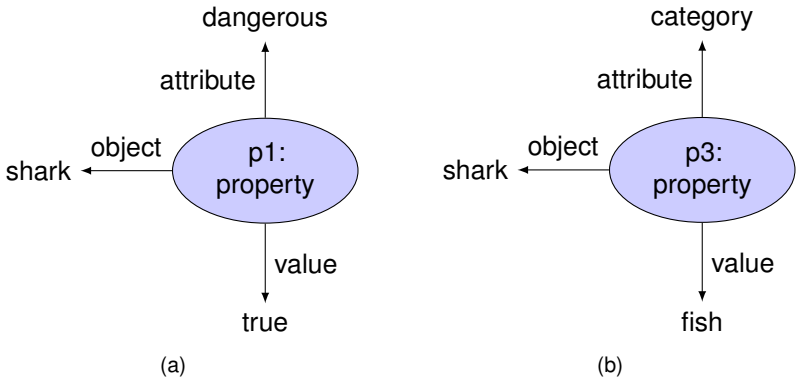


Figure 5.2: Two chunks of the type *property* encoding properties of the object *shark* as defined in figure 5.1. (a) This chunk encodes the fact, that a shark is dangerous. (b) The membership of an object in a category is also encoded by a *property* chunk. The attribute for this category membership property is *category* and the value encodes the name of the *category*.

```
=retrieval>
      ISA          property
      object       =obj1
      attribute    category
      value        =obj2
    - value        =cat
```

has to be translated to:

```
chain-category @
  buffer(goal,_,A),
    chunk(A,is-member),
    chunk_has_slot(A,object,C),
    chunk_has_slot(A,category,D),
    chunk_has_slot(A,judgment,pending),
  buffer(retrieval,_,B),
    chunk(B,property),
    chunk_has_slot(B,object,C),
    chunk_has_slot(B,attribute,category),
    chunk_has_slot(B,value,E)
  \ apply_rule(chain-category)
<=>
```

```
14   C\==nil,
15   D\==nil,
16   E\==nil,
17   E\==D |
18   ...
```

I.e., the duplicate slot test for the slot `value` is reduced to one single head constraint (`chunk_has_slot(B,value,E)`) and a guard, which states that `=obj2` $\neq$ `=cat` (the built-in check `E \== D`). After this small change, the model can be run with various queries, i.e. goal chunks as described in [Actb, unit 1, pp. 24 sqq.] and appendix B.3.

6 Conclusion

The goal of this thesis was to investigate the cognitive architecture *ACT-R* to develop a translation scheme and an implementation of its fundamental concepts using Constraint Handling Rules. It has been shown that the fundamental aspects of the system can be implemented very elegantly in CHR and the translation process can be automated. Hence, this work enables modelers to translate their ACT-R models without changing them and they are executable immediately. Furthermore, with the translation of the rules, modelers can directly utilize the various analysis methods of CHR programs and analyze the implications of their models formally. The behaviour of the models and the framework can be adapted easily to individual needs by just changing some rules for example for some special cases of the base-level learning or the initialization. Since the framework consists of a manageable set of elegant rules, even deep changes of the underlying cognitive architecture beyond the default parameters are possible.

Another goal was to facilitate the comparison of rule-based systems. As described before in the description of the procedural module (see chapter 4.3), the production rules of ACT-R avoid a lot of common problems of production rule systems like negation-as-absence and local variables. This is due to the – compared to other rule-based systems – simple architecture of the basic rules. For instance, the system does not allow to introduce new variables on the right hand side of a rule. Hence, all checks on the left hand side of an ACT-R production rule are simple matchings or comparisons of known values. This leads to very simple guard checks in the CHR translation; a lot of rules can even be reduced to a simple matching problem which is already supported automatically by CHR. The actions of a production rule are very limited, since they only affect the buffer system, i.e. an implementation only has to offer a limited amount of actions which can then be translated very easily.

One of the most complicated aspects of ACT-R implementations is the timing and the scheduling of conflict-resolution and module request events. However, even those concepts can be implemented elegantly using trigger constraints. The conflict resolution of ACT-R can be implemented by automated rule compilation, which produces two CHR rules out of one production rule. I.e., there is no unreasonable blow-up in the number of rules.

Thus, the implementation offers an elegant and adaptable cognitive architecture which sticks very close to the fundamental aspects of the ACT-R theory and reference implementation. It facilitates model analysis and comparison of the ACT-R production rule system to other rule-based formalisms.

Future Work This work has presented some of the fundamental concepts of ACT-R and mechanisms of translating them to CHR. However, there are some parts of the theory and common implementations which have not been regarded in this thesis. Simple ACT-R production rules without modified slot tests can be translated correctly. At the moment, the compiler lacks of translation methods for some allowed modifiers. Rules with duplicate slot tests cannot be translated correctly. Section 4.10.3 gives a more detailed overview of the problems of the compiler.

ACT-R furthermore provides an experiment environment which allows the modeler to create graphical frontends which can be used by both ACT-R models and humans. This environment is called *ACT-R GUI Interface (AGI)* [Bota] and communication is implemented by a network interface. The perceptual/motor modules usually are used in combination with the AGI and therefore have not been implemented, yet. The implementation is also lacking the imaginal module, since the fundamental concepts can be shown using only the declarative and the goal module.

Additionally, only a subset of the production rule grammar has been implemented, yet, so modification requests, strict harvesting, explicit variable bindings and evaluation functions which may produce side-effects have been ignored in the current implementation and may be part of future versions.

In future work, the concepts of skill acquisition, i.e. the creation of new procedural knowledge, especially the concept of production rule compilation, may be investigated in addition. Furthermore, the possibilities of the analysis of ACT-R rules and the evaluation of experiments may be inspected.

Bibliography

[Abo] *About ACT-R*. URL: http://act-r.psy.cmu.edu/about/ (visited on 08/19/2013).

[Acta] *The ACT-R Homepage*. URL: http://act-r.psy.cmu.edu/ (visited on 03/22/2013).

[Actb] *The ACT-R Tutorial*. 2012. URL: http://act-r.psy.cmu.edu/actr6/units.zip.

[AFS13] Slim Abdennadher, Ghada Fakhry, and Nada Sharaf. "Implementation of the Operational Semantics for CHR with User-defined Rule Priorities". In: *CHR '13: Proc. 10th Workshop on Constraint Handling Rules*. Ed. by Henning Christiansen and Jon Sneyers. K.U.Leuven, Department of Computer Science, Technical report CW 641, 07/2013, pp. 1–12.

[And+04] John R. Anderson, Daniel Bothell, Michael D. Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. "An Integrated Theory of the Mind". In: *Psychological Review* 111.4 (2004), pp. 1036–1060. ISSN: 0033-295X. DOI: 10.1037/0033-295X.111.4.1036. URL: http://doi.apa.org/getdoi.cfm?doi=10.1037/0033-295X.111.4.1036 (visited on 04/24/2013).

[And07] John R. Anderson. *How can the human mind occur in the physical universe?* English. Oxford University Press, 2007. ISBN: 978-0-19-539895-3.

[AR99] John R. Anderson and Lynne M. Reder. "The fan effect: New results and new theories". In: *JOURNAL OF EXPERIMENTAL PSYCHOLOGY GENERAL* 128 (1999), 186–197. URL: http://www.andrew.cmu.edu/user/reder/publications/99_jra_lmr_2.pdf (visited on 05/05/2013).

[ARL96] John R. Anderson, Lynne M. Reder, and Christian Lebiere. "Working memory: Activation limitations on retrieval". In: *Cognitive psychology* 30.3 (1996), 221–256. URL: http://citeseerx.ist.psu.edu/viewdoc/

download?doi=10.1.1.5.6397&rep=rep1&type=pdf (visited on 04/24/2013).

[AS00] John R. Anderson and Christian D. Schunn. “Implications of the ACT-R learning theory: No magic bullets”. In: *Advances in instructional psychology: Educational design and cognitive science*. Ed. by R. Glaser. Vol. 5. Hillsdale, NJ: Lawrence Erlbaum Associates, 2000, pp. 1–33.

[BG10] Marcello Balduccini and Sara Girotto. “Formalization of psychological knowledge in answer set programming and its application”. In: *Journal of Theory and Practice of Logic Programming (TPLP)* 10.4-6 (2010), 725–740. URL: http://journals.cambridge.org/production/action/cjoGetFulltext?fulltextid=7834603 (visited on 04/24/2013).

[Bota] Dan Bothell. *ACT-R 6.0 AGI Manual*. Department of Psychology, Carnegie Mellon University. Pittsburgh, Pennsylvania 15213.

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

[Chr] *The CHR Homepage*. URL: http://www.constraint-handling-rules.org (visited on 08/10/2013).

[Frü09] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 08/2009. ISBN: 9780521877763. URL: http://www.constraint-handling-rules.org.

[Frü10] Thom Frühwirth. *CHR – a common platform for rule-based approaches*. 2010. URL: http://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/chr-book-slides-chap6.pdf (visited on 08/26/2013).

[Jaca] *Benefits of jACT-R (part of the FAQ section of the homepage)*. URL: http://jactr.org/node/50 (visited on 08/13/2013).

[Jacb] *The Homepage of jACT-R*. URL: http://jactr.org/ (visited on 08/12/2013).

[Java] *About ACT-R: The Java Simulation & Development Environment*. URL: http://cog.cs.drexel.edu/act-r/about.html (visited on 08/13/2013).

[Javb] *ACT-R: The Java Simulation & Development Environment – Homepage*. URL: http://cog.cs.drexel.edu/act-r/ (visited on 08/12/2013).

[LM] James Lu and Jerud J. Mead. *Prolog – A Tutorial Introduction*. Computer Science Department, Bucknell University. Lewisburg, PA 17387.

[New90] Allen Newell. *Unified theories of cognition*. Cambridge, MA, USA: Harvard University Press, 1990. ISBN: 0-674-92099-6.

[Ogb] Anne Ogborn. *Using Definite Clause Grammars in SWI-Prolog*. URL: http://www.pathwayslms.com/swipltuts/dcg/ (visited on 08/12/2013).

[PS07] Luís Moniz Pereira and Ari Saptawijaya. "Modelling morality with prospective logic". In: *Procs. 13th Portuguese Intl.Conf. on Artificial Intelligence (EPIA'07), LNAI*. Springer, 2007.

[RT05] M Rutledge-Taylor. "Can ACT-R realize 'Newell's dream'?" In: *Proceedings of the 27th annual meeting of the Cognitive Science Society*. 2005.

[RW72] R. A. Rescorla and A. W. Wagner. "A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement". In: *Classical Conditioning II: Current Research and Theory*. Ed. by A. H. Black and W. F. Prokasy. New York: Appleton-Century-Crofts, 1972. Chap. 3, pp. 64–99.

[Sne+10] Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. "As Time Goes By: Constraint Handling Rules – A Survey of CHR Research between 1998 and 2007". In: *Theory and Practice of Logic Programming* 10.1 (2010), pp. 1–47. DOI: 10.1017/S1471068409990123.

[Sun08] Ron Sun. "Introduction to Computational Cognitive Modeling". In: *The Cambridge Handbook of Computational Psychology*. Ed. by Ron Sun. New York: Cambridge University Press, 2008, pp. 3–19. URL: http://www.cogsci.rpi.edu/~rsun/folder-files/sun-CHCP-intro.pdf (visited on 07/15/2013).

[SW06] Terrence C. Stewart and Robert L. West. "Deconstructing ACT-R". In: *Proceedings of the Seventh International Conference on Cognitive Modeling*. 2006, 298–303. URL: http://actr.psy.cmu.edu/papers/641/stewartPaper.pdf (visited on 04/24/2013).

[SW07] Terrence C. Stewart and Robert L. West. “Deconstructing and reconstructing ACT-R: Exploring the architectural space”. In: *Cognitive Systems Research* 8.3 (09/2007), pp. 227–236. ISSN: 13890417. DOI: 10.1016/j.cogsys.2007.06.006. URL: http://linkinghub.elsevier.com/retrieve/pii/S1389041707000253 (visited on 04/24/2013).

[Swi] *The SWI-Prolog Homepage*. URL: http://www.swi-prolog.org/ (visited on 08/10/2013).

[TL03] Niels A. Taatgen and Frank J. Lee. “Production compilation: A simple mechanism to model complex skill acquisition”. In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 45.1 (2003), 61–76. URL: http://hfs.sagepub.com/content/45/1/61.short (visited on 04/05/2013).

[TLA06] Niels A. Taatgen, C. Lebiere, and J.R. Anderson. “Modeling Paradigms in ACT-R”. In: *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation.* Cambridge University Press, 2006, pp. 29–52. URL: http://act-r.psy.cmu.edu/papers/570/SDOC4697.pdf (visited on 04/05/2013).

[Whi] Jacob Whitehill. *Understanding ACT-R – an Outsider’s Perspective*. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.184.8589&rep=rep1&type=pdf (visited on 03/22/2013).

A CD Content

Thesis The thesis can be found in the folder `thesis`. It contains the LaTeX source files. The base document is `thesis.tex`, the individual chapters can be found in `chapters`. To compile the files, a lot of packages are needed. For example, the graphics are produced with Ti*k*Z, so make sure, all necessary packages are installed on your system.

CHR-ACT-R The source of the implementation presented in this work can be found in the directory `CHR-ACT-R/src`. It is separated into two parts: the compiler and the framework.

- The compiler is in the directory `compiler`. It can be started by consulting `actr2chr.pl` in SWI-Prolog. The query `compile_file(f).` compiles the file `f`. There are several example model files in the directory of the compiler which all start with the prefix `example_`.
- The framework can be found in the directory `core`. To load a model, the compiled model file has to be consulted in SWI-Prolog. There are several compiled example models all starting with the prefix `example_`. Make sure that the models are in the same folder as the framework. The query `run.` runs the model.

B Executable Examples

This appendix provides some of the examples that appeared in the work with a minimal environment that represents the current context where the examples appeared.

B.1 Rule Order

In this example, the need of a phase constraint (`fire/0`) is illustrated. For the query

```
?- chunk(c,foo), chunk_has_slot(c,s1,v1), chunk(c2,bar),
   chunk_has_slot(c2,s,v), buffer(b2,c2), buffer(b1,c).
```

the example yields a wrong result. If uncommenting the rules with the fire constraint instead of the rules without, the result will be correct. See section 4.3.3 on page 53.

Listing B.1: Rule order example

```
:- use_module(library(chr)).

:- chr_type chunk_def ---> nil; chunk(any, any, slot_list).
:- chr_type list(T) ---> []; [T | list(T)].
:- chr_type slot_list == list(pair(any,any)). % a list of
   slot-value pairs
:- chr_type pair(T1,T2) ---> (T1,T2).

:- chr_type lchunk_defs == list(chunk_def).

:- chr_constraint buffer/2, buffer_change/2, alter_slots/2,
   alter_slot/3, chunk/2, chunk_has_slot/3,fire.

% Handle buffer_change
```

```
13 buffer(BufName, OldChunk) \ buffer_change(BufName,
       chunk(_,_,SVs)) <=>
14   alter_slots(OldChunk,SVs).
15
16 alter_slots(_,[]) <=> true.
17 alter_slots(Chunk,[(S,V)|SVs]) <=>
18   alter_slot(Chunk,S,V),
19   alter_slots(Chunk,SVs).
20
21 alter_slot(Chunk,Slot,Value), chunk_has_slot(Chunk,Slot,_) <=>
22   chunk_has_slot(Chunk,Slot,Value).
23
24 alter_slot(Chunk,Slot,Value) <=>
25   false. % since every chunk must be described completely, Slot
         cannot be a slot of the type of Chunk
26   %chunk_has_slot(Chunk,Slot,Value).
27
28 % first example without fire:
29
30  buffer(b1,C),
31    chunk(C,foo),
32    chunk_has_slot(C,s1,v1)
33  ==>
34  buffer_change(b1,chunk(_,_,[(s1,v2)])),
35  buffer_change(b2,chunk(_,_,[(s,x)])).
36
37  buffer(b1,C),
38    chunk(C,foo),
39    chunk_has_slot(C,s1,v2)
40  ==>
41  buffer_change(b2,chunk(_,_,[(s,y)])),
42  buffer_change(b1,chunk(_,_,[(s1,v3)])).
43
44 % example with fire (uncomment it and add comments to the rules
       above)
45
46 % buffer(b1,C),
47 %   chunk(C,foo),
48 %   chunk_has_slot(C,s1,v1)
49 %   \ fire
```

```
% <=>
% buffer_change(b1,chunk(_,_,[(s1,v2)])),
% buffer_change(b2,chunk(_,_,[(s,x)])),
% fire.
%
% buffer(b1,C),
%   chunk(C,foo),
%   chunk_has_slot(C,s1,v2)
%   \ fire
% <=>
% buffer_change(b2,chunk(_,_,[(s,y)])),
% buffer_change(b1,chunk(_,_,[(s1,v3)])),
% fire.
```

B.2 Subsymbolic Layer

Listing B.2: Counting example with subsymbolic layer and training

```
:- include('actr_core.pl').
:- chr_constraint run/0, fire/0.

delay-train1 @
  fire,
  buffer(goal,_,A),
    chunk(A,goal-chunk),
      chunk_has_slot(A,goal,training1)
==>
  conflict_set(train1).

train1 @
  buffer(goal,_,A),
    chunk(A,goal-chunk),
      chunk_has_slot(A,goal,training1)
  \ apply_rule(train1)
<=>
  true |
  buffer_change(goal,chunk(_,_,
    [ (goal,training2)])),
```

```
  buffer_request(retrieval,
    chunk(_,count-order,
      [ (first,3),
        (second,4)])),
  conflict_resolution.

delay-train2 @
  fire,
  buffer(goal,_,A),
    chunk(A,goal-chunk),
      chunk_has_slot(A,goal,training2),
  buffer(retrieval,_,B),
    chunk(B,count-order),
      chunk_has_slot(B,first,3),
      chunk_has_slot(B,second,4)
==>
  true |
  conflict_set(train2).

train2 @
  buffer(goal,_,A),
    chunk(A,goal-chunk),
      chunk_has_slot(A,goal,training2),
  buffer(retrieval,_,B),
    chunk(B,count-order),
      chunk_has_slot(B,first,3),
      chunk_has_slot(B,second,4)
  \ apply_rule(train2)
<=>
  true |
  buffer_change(goal,chunk(_,goal-chunk,
    [ (start,2),
      (end,4),
      (goal,count)])),
  buffer_clear(retrieval),
  conflict_resolution.

delay-start @
  fire,
  buffer(goal,_,A),
```

```
61      chunk(A,goal-chunk),
62        chunk_has_slot(A,goal,count),
63        chunk_has_slot(A,start,B),
64        chunk_has_slot(A,count,nil)
65  ==>
66    B\==nil |
67    conflict_set(start).
68
69  start @
70    buffer(goal,_,A),
71      chunk(A,goal-chunk),
72        chunk_has_slot(A,goal,count),
73        chunk_has_slot(A,start,B),
74        chunk_has_slot(A,count,nil)
75    \ apply_rule(start)
76  <=>
77    B\==nil |
78    buffer_change(goal,
79      chunk(_,_,[ (count,B)])),
80    buffer_request(retrieval,
81      chunk(_,count-order,[ (first,B)])),
82    conflict_resolution.
83
84  delay-increment @
85    fire,
86    buffer(goal,_,A),
87      chunk(A,goal-chunk),
88        chunk_has_slot(A,goal,count),
89        chunk_has_slot(A,count,C),
90        chunk_has_slot(A,end,D),
91    buffer(retrieval,_,B),
92      chunk(B,count-order),
93        chunk_has_slot(B,first,C),
94        chunk_has_slot(B,second,E)
95  ==>
96    C\==nil,
97    D\==C,
98    E\==nil |
99    conflict_set(increment).
100
```

```
increment @
  buffer(goal,_,A),
    chunk(A,goal-chunk),
      chunk_has_slot(A,goal,count),
      chunk_has_slot(A,count,C),
      chunk_has_slot(A,end,D),
  buffer(retrieval,_,B),
    chunk(B,count-order),
      chunk_has_slot(B,first,C),
      chunk_has_slot(B,second,E)
  \ apply_rule(increment)
<=>
  C\==nil,
  D\==C,
  E\==nil |
  buffer_change(goal,
    chunk(_,_,[ (count,E)])),
  buffer_request(retrieval,
    chunk(_,count-order,[ (first,E)])),
    output(C),
    conflict_resolution.

delay-incrementx @
  fire,
  buffer(goal,_,A),
    chunk(A,goal-chunk),
      chunk_has_slot(A,goal,count),
      chunk_has_slot(A,count,C),
      chunk_has_slot(A,end,D),
  buffer(retrieval,_,B),
    chunk(B,count-order),
      chunk_has_slot(B,first,C),
      chunk_has_slot(B,second,E)
==>
  C\==nil,
  D\==C,
  E\==nil |
  conflict_set(incrementx).

incrementx @
```

```
141   buffer(goal,_,A),
142     chunk(A,goal-chunk),
143       chunk_has_slot(A,goal,count),
144       chunk_has_slot(A,count,C),
145       chunk_has_slot(A,end,D),
146   buffer(retrieval,_,B),
147     chunk(B,count-order),
148       chunk_has_slot(B,first,C),
149       chunk_has_slot(B,second,E)
150   \ apply_rule(incrementx)
151 <=>
152   C\==nil,
153   D\==C,
154   E\==nil |
155   buffer_clear(goal),
156   output(wrong),
157   conflict_resolution.
158
159 delay-stop @
160   fire,
161   buffer(goal,_,A),
162   chunk(A,goal-chunk),
163     chunk_has_slot(A,goal,count),
164     chunk_has_slot(A,count,B),
165     chunk_has_slot(A,end,B)
166 ==>
167   B\==nil |
168   conflict_set(stop).
169
170 stop @
171   buffer(goal,_,A),
172     chunk(A,goal-chunk),
173       chunk_has_slot(A,goal,count),
174       chunk_has_slot(A,count,B),
175       chunk_has_slot(A,end,B)
176   \ apply_rule(stop)
177 <=>
178   B\==nil |
179   buffer_clear(goal),
180   output(B),
```

```
  conflict_resolution.

init @
  run <=>
    now(0),
    set_default_utilities([stop, incrementx, increment, start,
       train2, train1]),
    add_buffer(retrieval,declarative_module),
    add_buffer(goal,declarative_module),
    lisp_chunktype([chunk]),
    lisp_sgp([:,esc,t]),
    lisp_chunktype([count-order,first,second]),
    lisp_chunktype([goal-chunk,goal,start,end,count]),
    lisp_adddm([
      [b,isa,count-order,first,1,second,2],
      [c,isa,count-order,first,2,second,3],
      [d,isa,count-order,first,3,second,4],
      [d1,isa,count-order,first,3,second,5],
      [e,isa,count-order,first,4,second,5],
      [f,isa,count-order,first,5,second,6],
      [first-goal,isa,goal-chunk,goal,training1]]),
    lisp_goalfocus([first-goal]),
    lisp_spp([increment,:,u,8,incrementx,:,u,0]),
    lisp_spp([stop,:,reward,15]),
    conflict_resolution,
    nextcyc.

no-rule @
  fire<=>
    conflict_set([]),
    choose.
```

B.3 Semantic Model

Listing B.3: ACT-R production rules for the semantic model

```
(define-model semantic

```

```
(chunk-type property object attribute value)
(chunk-type is-member object category judgment)

(add-dm
 (shark isa chunk) (dangerous isa chunk)
 (locomotion isa chunk) (swimming isa chunk)
 (fish isa chunk) (salmon isa chunk)
 (edible isa chunk) (breathe isa chunk)
 (gills isa chunk) (animal isa chunk)
 (moves isa chunk) (skin isa chunk)
 (canary isa chunk) (color isa chunk)
 (sings isa chunk) (bird isa chunk)
 (ostrich isa chunk) (flies isa chunk)
 (height isa chunk) (tall isa chunk)
 (wings isa chunk) (flying isa chunk)
 (true isa chunk) (false isa chunk)
 (p1 ISA property object shark attribute dangerous value true)
 (p2 ISA property object shark attribute locomotion value
    swimming)
 (p3 ISA property object shark attribute category value fish)
 (p4 ISA property object salmon attribute edible value true)
 (p5 ISA property object salmon attribute locomotion value
    swimming)
 (p6 ISA property object salmon attribute category value fish)
 (p7 ISA property object fish attribute breathe value gills)
 (p8 ISA property object fish attribute locomotion value
    swimming)
 (p9 ISA property object fish attribute category value animal)
 (p10 ISA property object animal attribute moves value true)
 (p11 ISA property object animal attribute skin value true)
 (p12 ISA property object canary attribute color value yellow)
 (p13 ISA property object canary attribute sings value true)
 (p14 ISA property object canary attribute category value bird)
 (p15 ISA property object ostrich attribute flies value false)
 (p16 ISA property object ostrich attribute height value tall)
 (p17 ISA property object ostrich attribute category value bird)
 (p18 ISA property object bird attribute wings value true)
 (p19 ISA property object bird attribute locomotion value
    flying)
 (p20 ISA property object bird attribute category value animal)
```

```
 (g1 ISA is-member object canary category bird judgment nil)
 (g2 ISA is-member object canary category animal judgment nil)
 (g3 ISA is-member object canary category fish judgment nil))

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


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

(P chain-category
   =goal>
      ISA         is-member
      object      =obj1
```

```
      category    =cat
      judgment    pending
   =retrieval>
      ISA         property
      object      =obj1
      attribute   category
      value       =obj2
    - value       =cat
==>
   =goal>
      object      =obj2
   +retrieval>
      ISA         property
      object      =obj2
      attribute   category
)

 (P fail
   =goal>
      ISA         is-member
      object      =obj1
      category    =cat
      judgment    pending

    ?retrieval>
      state       error
==>
   =goal>
      judgment    no
)


(goal-focus g1)
)
```

Listing B.4: Translated production rules for the semantic model in CHR

```
:- include('actr_core.pl').
:- chr_constraint run/0, fire/0.

```

```
4
5  delay-initial-retrieve @
6    fire,
7    buffer(goal,_,A),
8      chunk(A,is-member),
9        chunk_has_slot(A,object,B),
10       chunk_has_slot(A,category,C),
11       chunk_has_slot(A,judgment,nil)
12 ==>
13   B\==nil,
14   C\==nil |
15   conflict_set(initial-retrieve).
16
17 initial-retrieve @
18   buffer(goal,_,A),
19     chunk(A,is-member),
20       chunk_has_slot(A,object,B),
21       chunk_has_slot(A,category,C),
22       chunk_has_slot(A,judgment,nil)
23   \ apply_rule(initial-retrieve)
24 <=>
25   B\==nil,
26   C\==nil |
27   buffer_change(goal,
28     chunk(_,_,[ (judgment,pending)])),
29   buffer_request(retrieval,
30     chunk(_,property,
31       [ (object,B),
32         (attribute,category)])),
33   conflict_resolution.
34
35 delay-direct-verify @
36   fire,
37   buffer(goal,_,A),
38     chunk(A,is-member),
39       chunk_has_slot(A,object,C),
40       chunk_has_slot(A,category,D),
41       chunk_has_slot(A,judgment,pending),
42   buffer(retrieval,_,B),
43     chunk(B,property),
```

```
      chunk_has_slot(B,object,C),
      chunk_has_slot(B,attribute,category),
      chunk_has_slot(B,value,D)
==>
  C\==nil,
  D\==nil |
  conflict_set(direct-verify).

direct-verify @
  buffer(goal,_,A),
    chunk(A,is-member),
      chunk_has_slot(A,object,C),
      chunk_has_slot(A,category,D),
      chunk_has_slot(A,judgment,pending),
  buffer(retrieval,_,B),
    chunk(B,property),
      chunk_has_slot(B,object,C),
      chunk_has_slot(B,attribute,category),
      chunk_has_slot(B,value,D)
  \ apply_rule(direct-verify)
<=>
  C\==nil,
  D\==nil |
  buffer_change(goal,
    chunk(_,_,[ (judgment,yes)])),
  conflict_resolution.

delay-chain-category @
  fire,
  buffer(goal,_,A),
    chunk(A,is-member),
      chunk_has_slot(A,object,C),
      chunk_has_slot(A,category,D),
      chunk_has_slot(A,judgment,pending),
  buffer(retrieval,_,B),
    chunk(B,property),
      chunk_has_slot(B,object,C),
      chunk_has_slot(B,attribute,category),
      chunk_has_slot(B,value,E)
==>
```

```
84    C\==nil,
85    D\==nil,
86    E\==nil,
87    E\==D |
88    conflict_set(chain-category).
89
90  chain-category @
91    buffer(goal,_,A),
92      chunk(A,is-member),
93        chunk_has_slot(A,object,C),
94        chunk_has_slot(A,category,D),
95        chunk_has_slot(A,judgment,pending),
96    buffer(retrieval,_,B),
97      chunk(B,property),
98        chunk_has_slot(B,object,C),
99        chunk_has_slot(B,attribute,category),
100       chunk_has_slot(B,value,E)
101   \ apply_rule(chain-category)
102 <=>
103   C\==nil,
104   D\==nil,
105   E\==nil,
106   E\==D |
107   buffer_change(goal,
108     chunk(_,_,[ (object,E)])),
109   buffer_request(retrieval,
110     chunk(_,property,
111       [ (object,E),
112         (attribute,category)])),
113   conflict_resolution.
114
115 delay-fail @
116   fire,
117   buffer(goal,_,A),
118     chunk(A,is-member),
119       chunk_has_slot(A,object,B),
120       chunk_has_slot(A,category,C),
121       chunk_has_slot(A,judgment,pending),
122   buffer_state(retrieval,error)
123 ==>
```

```
124   B\==nil,
125   C\==nil |
126   conflict_set(fail).
127
128 fail @
129   buffer(goal,_,A),
130     chunk(A,is-member),
131       chunk_has_slot(A,object,B),
132       chunk_has_slot(A,category,C),
133       chunk_has_slot(A,judgment,pending),
134   buffer_state(retrieval,error)
135   \ apply_rule(fail)
136 <=>
137   B\==nil,
138   C\==nil |
139   buffer_change(goal,
140     chunk(_,_,[ (judgment,no)])),
141   conflict_resolution.
142
143 init @
144   run <=>
145     set_default_utilities([fail, chain-category, direct-verify,
          initial-retrieve]),
146     add_buffer(retrieval,declarative_module),
147     add_buffer(goal,declarative_module),
148     lisp_chunktype([chunk]),
149     lisp_chunktype([property,object,attribute,value]),
150     lisp_chunktype([is-member,object,category,judgment]),
151     lisp_adddm([
152       [shark, isa, chunk],
153       [dangerous, isa, chunk],
154       { ... every primitive element is created as chunk of type
            chunk. Can be omitted, since it is done automatically
            }
155       [p1, isa, property, object, shark, attribute, dangerous,
            value, true],
156       [p2, isa, property, object, shark, attribute, locomotion,
            value, swimming],
157       [p3, isa, property, object, shark, attribute, category,
            value, fish],
```

```
158         [p4, isa, property, object, salmon, attribute, edible,
               value, true],
159         [p5, isa, property, object, salmon, attribute,
               locomotion, value, swimming],
160         [p6, isa, property, object, salmon, attribute, category,
               value, fish],
161         [p7, isa, property, object, fish, attribute, breathe,
               value, gills],
162         [p8, isa, property, object, fish, attribute, locomotion,
               value, swimming],
163         [p9, isa, property, object, fish, attribute, category,
               value, animal],
164         [p10, isa, property, object, animal, attribute, moves,
               value, true],
165         [p11, isa, property, object, animal, attribute, skin,
               value, true],
166         [p12, isa, property, object, canary, attribute, color,
               value, yellow],
167         [p13, isa, property, object, canary, attribute, sings,
               value, true],
168         [p14, isa, property, object, canary, attribute, category,
               value, bird],
169         [p15, isa, property, object, ostrich, attribute, flies,
               value, false],
170         [p16, isa, property, object, ostrich, attribute, height,
               value, tall],
171         [p17, isa, property, object, ostrich, attribute,
               category, value, bird],
172         [p18, isa, property, object, bird, attribute, wings,
               value, true],
173         [p19, isa, property, object, bird, attribute, locomotion,
               value, flying],
174         [p20, isa, property, object, bird, attribute, category,
               value, animal],
175         [g1, isa, is-member, object, canary, category, bird,
               judgment, nil],
176         [g2, isa, is-member, object, canary, category, animal,
               judgment, nil],
177         [g3, isa, is-member, object, canary, category, fish,
               judgment, nil]]),
```

```
178     lisp_goalfocus([g1]), % choose one of g1, g2, g3
179     now(0),
180     conflict_resolution,nextcyc.
181
182 no-rule @
183   fire <=>
184     conflict_set([]),
185     choose.
```

Name: Daniel Gall Matrikelnummer: 645463

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Daniel Gall