Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Programmiermethodik und Compilerbau

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

Masterarbeit an der Universität Ulm

Vorgelegt von:
Daniel Gall
daniel.gall@uni-ulm.de

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

Betreuer:
Prof. Dr. Thom Frühwirth

2013

“A Rule-Based Implementation of ACT-R Using Constraint Handling Rules”
Version of July 30, 2013

Typesetting: PDF-LaTeX 2_ε
Printed by: Communication- and Information Center (kiz), Ulm University

Abstract

This is the abstract of my master thesis.

Contents

1 Introduction

2 Description of ACT-R

In computational psychology, the approach to explore human cognition is to implement detailed computational models that enable computers to execute them and simulate human behaviour [Sun08]. 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.

On the other hand, psychology is experiencing a movement towards specialization [And+04], ie. there are a lot of independent, highly specialized fields that lack a more global view.

To implement consistent models of cognition, it is necessary to develop a theory that tries to put all those highly specialized components together and allows modelers to build their models on the basis of this theory. Cognitive architectures try to explain **FIXME: definition cognitive architecture from book** **FIXME: move to introduction and motivation**

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]

On top of the provided cognitive architecture, one can specify models for specific tasks. The cognitive architecture constrains the modeling to facilitate the modeling process. Thereby it ensures cognitive plausibility to some degree, since models are built upon a highly verified theory [TLA06, p. 29].

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 **FIXME: source**. 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 **??**, 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 **??**.

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 **??** 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.

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, eg. 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 **??** the general computational process is illustrated by showing 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 that may lead to the next rule matching 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. Ie., 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 they are not critical for the description of the facts and just for readability in the theory. The real description of a concept comes from its connections.

Chunks can have slots that are connected to other chunks or elements. Such an element can be regarded as a chunk without any slots. For instance, the fact that five plus to equals to seven can be modeled as a chunk that is connected to the numbers 5, 2 and 7 (see **??**). Notice that in the figure each slot has a individual name. This is necessary to distinguish the connections of the chunks, otherwise the summands were indistinguishable from the sum in the example.

Each chunk is associated with a chunk-type that determines which slots a chunk can have. For example, the fact in figure **??** has the type `addition-fact`. All chunks of this type must provide the slots `arg1`, `arg2` and `sum`.

For the chunk types there is no upper limit of slots they can define. However, Anderson et al. suggested to limit the number of slots to Miller's Number of 7 ± 2, for the reason of plausibility [**unknown**]. **FIXME: Find cite**

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 in the buffer that can be processed by the procedural module, which is described in the next section.

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 buffer. In the condition-part of a rule it is defined which kind of chunk with certain 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 7 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. Actions can be changes of values in the chunks 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 can have a special meaning. The actual semantics of a request is dependent 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 as its argument 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.

[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.

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. While that time, other modules may perform requests triggered in the action of the first 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.

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 request is independent of the execution of the production rule and after the request has been stated by the procedural module, it can begin with the next recognize-cycle.

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.

Chunks as Central Data Structure

As may have become clear 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 results of a request is a chunk placed into a buffer and even the procedural system tries to match the chunks in the buffers in the condition part and 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 and is in progress, the procedural module is *free* and therefore can select a matching rule according to the recognize-act-cycle. If the rule has been selected, the module is *busy* and cannot choose another rule to fire. Between selection and firing of a rule the module has to wait $50\,\mathrm{ms}$. 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, where the procedural module is free and cannot find a matching rule. If the request has been performed, it usually has a change of buffers as an effect. At this point of time, when the content of a buffer has changed, this could cause 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 can have different subgoals that have to be achieved to achieve 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 ACT-R, the goal module with the goal buffer is used for this purpose.

Working memory

The goal module and buffer are often referred to as *working memory* [And+04, p. 1041], but actually it can have another meaning as stated in [ARL96]: As usual in production systems, everything that is present to the production system and can match against the production rules 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 **??** 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 perceiving sounds in the environment.

Like with every other module, communication is achieved through the buffers of those 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, ie. that the whole visual field is processed massively parallel and the time of finding one green object surrounded by blue objects is constant, regardless of the number of blue objects, for example. If there are more than one object meeting the constraints, then one 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].

The visual system and its capabilities are described in detail in [Actb, unit 2] and also regards the implementational details of the system.

The Imaginal Module

The imaginal module 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). The visual-object buffer can hold only one chunk at once. A solution could be, to save all the information in 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, the best way would be 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 with every clearing of a buffer the chunk in that buffer is stored in declarative memory[3], a unlimited amount of data can be produced and remembered by stating retrieval requests later on.

[2] As described in section 2.1.2, one should stick to 7 ± 2 slots for each chunk.
[3] see section 2.1.3

It is important to mention, that it takes the imaginal module $.2\,\mathrm{ms}$ 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].

The imaginal module is described in [Actb, unit 2].

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, eg. "‘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 **??** of this type *count-fact* model the facts that 3 is the successor of 2 and 4 is the successor of 3.

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, the goal chunk could have a slot, that always holds the current number that has been counted to. This leads to a goal chunk as illustrated in figure **??**, where the current number is 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

Now the rules to implement counting can be defined as:

count-rule If the goal is to count and 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 $m = 3$.

[4] like every module can only handle one request at a time

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 function of module requests: The request states a (potentially partial) chunk definition and the corresponding module puts the result of the request in a fully defined chunk 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.

The count-rule will be applicable as long as there are *count-facts* in the declarative memory. Figure **??** illustrates the counting process.

In this example, the rules have been defined in a very informal way. In the following chapters that deal with implementation, a formalization of such rules will be discussed, that defines clearly, what kinds of rules are allowed and 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 **??**, 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.

FIXME: figures

2.2 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 Serialism: 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 needing the result of the computation can fire and computation is continued.

2.3 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].

2.3.1 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 two chunks that encode addition facts for the same two arguments (let them be 5 and 2), but with different sums (6 and 7), for example. This could be the case, if a child learned the wrong fact, for example. 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 of being retrieved: If the activation A_i is less than a certain *threshold* τ, then 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*.

The base-level activation is a value associated with each chunk and depends on how often a chunk has been practiced and when the last retrieval of the chunk has been performed. 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 a 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 that is 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:

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

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]. Figure **??** illustrates the addition-fact $5 + 2 = 7$ with the corresponding quantities introduced in the last equation.

The values for W_j reflect how many sources of activation are in the current context. A source of activation is a chunk in the goal buffer or in all buffers[5], 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 becomes

[5]This is called the current context: Usually it means the set of all chunks in all buffers, but there are definitions in literature, that only call the chunk in the goal buffer current context.

a particular connection between a chunk from the context with a chunk from declarative memory.

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:

- 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, sentences constructed from the same persons and locations, but that were not in the original set of sentences. Figure **??** represents an example chunk network of the studied sentences (based on [AR99, fig. 1].

"The term *fan* refers to the number of facts associated with a particular concept" [AR99, p. 186]. In figure **??**, some facts are shown 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 was the retrieval time for a particular fact about that concept.

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 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 **??** illustrates the activation spreading process.

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 retrieval time of a chunk:

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

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

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

where τ is the retrieval threshold as mentioned in section 2.3.1, 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.

2.3.2 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 now has the goal to add two numbers, it somehow has to decide which strategy it will choose, since both strategies match the context.

In ACT-R, 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 are more than one matching rules. 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 nth application [And07, pp. 160–161]. This leads to 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 every time and it is not clear which production rule will be strengthened by the reward. In [And07, p. 161] an example is described, where a monkey receives a squirt of juice a second after he presses a button. The question now 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 is the reward this particular rule receives. 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, eg., 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 the definition of the reward for a production rule, rules also can get a negative reward, if their selection was too long ago. If one wants to penalize all rules since the last reward, a rule with reward 0 can be triggered, which leads to all rules applied before being rewarded with a negative amount of reward.

2.4 Learning

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

Table 2.1: ACT-R's Taxonomy of Learning [And07, pp. 92–95]

	Declarative	Procedural
Symbolic	Fact learning	Skill acquisition
Subsymbolic	Strengthening	Conditioning

2.4.1 Symbolic Layer

Symbolic learning somehow influences the objects of the symbolic layer, ie. chunks and production rules, in a way that new objects are created or objects are merged. Those learning possibilities are described in the following.

Fact Learning

Skill acquisition

2.4.2 Subsymbolic Layer

The concepts introduced in section 2.3 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[6]. Additionally, associative weights are learned from the current context. These processes adapt to the problems a particular human mind is confronted with and 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).

[6]This is the concept of base-level learning as described in section 2.3.1

[Whi, chapter 4]

3 Constraint Handling Rules

4 Transformation of ACT-R to CHR

After the comprehensive but at some point informal overview of the ACT-R theory in chapter 2, this chapter presents a possible implementation of the described concepts of the ACT-R theory in CHR.

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, ie. 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 [**actr_reference**] 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-Rs 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 and to exchange information between modules and to state requests, this section first deals with this central 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 then. 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 bases upon a relation that represents such a network.

Definition 4.1 (chunk-store)**.** *A* chunk-store Σ *is a tuple* $(C, E, \mathcal{T}, HasSlot, Isa)$*, where* C *is a set of chunks and* E *a set of primitive elements, with* $C \cap E = \emptyset$*.* $V = C \cup E$ *are the* values *of* Σ *and* $\mathcal{T}$ *a set of chunk-types. A chunk-type* $(T, S) \in \mathcal{T}$ *is a tuple with a type name* T *and a set of slots* S*. The set of all slot-names is* $\mathcal{S}$*.*

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

- $c \;\; Isa \;\; T \Leftrightarrow$ *chunk* c *is of type* T*.*
- $(c, s, v) \in HasSlot \Leftrightarrow v$ *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*".*

The function $slots : C \to \mathcal{S} \times V$ *is defined as* $slots(c) = \{(s, v) | (c, s, v) \in HasSlot\}$ *and* $slots : \mathcal{T} \to \mathcal{S}$ *as* $slots((T, S)) = S$*.*

A chunk-store is type-consistent*, if* $\forall (c, s, v) \in HasSlot : c \ Isa \ (T, S) \Rightarrow \exists s' \in S : s = s'$ *and* $\forall c \in C : \ c \ Isa \ (T, S) \Rightarrow \forall s \in S : \exists (c', s', v) \in HasSlot : c = c', s = s'$*.*

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

$\mathtt{chunk-type(nameslot_1 slot_2 \ldots slot_n)}$ *adds the type* $T = (\mathtt{name}, \{\mathtt{slot1}, \ldots, \mathtt{slotn}\})$ *to the store, ie.* $\mathcal{T}' = \mathcal{T} \cup \{T\}$*.*

$\mathtt{add-chunk(nameisatypeslot_1 val_1 \ldots slot_n val_n)}$ *adds a new chunk to the store, ie.* $C' = C \cup \{\mathtt{name}\}$*,* $Isa' = Isa \cup (\mathtt{name}, (\mathtt{type}, slots(\mathtt{type}))$ *and* $HasSlot' = \bigcup_{i=1}^{n} (\mathtt{slot_i}, \mathtt{val_i}) \cup HasSlot$*. 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 `add-chunk` *method. Since it is claimed by the definition of* $HasSlot$ *that for all slots* s *of a chunk* c *there must be a* $(c, s, v) \in HasSlot$*, in implementations the unspecified slots are initialized as empty slots, represented by the empty value* `nil`*.*

FIXME: add other methods. describe empty chunks?

Example 4.1. *The addition-fact chunk in figure* **??** *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:

$$
\begin{aligned}
(\{a\}, \{2, 5, 7\},\\
\{(addition - fact, \{arg1, arg2, sum\})\},\\
\{(a, arg1, 5), (a, arg2, 5), (a, sum, 7)\},\\
\{(a, addition - fact)\}).
\end{aligned}
$$

$slots(a) = \{(arg1, 5), (arg2, 2), (sum, 7)\}$ *and* $slots((addition - fact, \{arg1, arg2, sum\}) = \{arg1, arg2, sum\}$*. Hence, the store is type-consistent.*

Example 4.2. **FIXME: refer to empty chunks**

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 $HasSlot$, 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[1].

For the chunks:

```
:- 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 $T = (\texttt{t}, S)$.

The $HasSlot$ relation is represented by the `chunk_has_slot(c,s,v)` constraint, which really is just a direct translation of an element $(c, s, v) \in HasSlot$.

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 known by the brain.

[1]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 `chunk_type_has_slot(t,s)` in the constraint store.

Additionally, from the definition of a chunk store it is known, that the $HasSlot$ and the Isa relations have to be left-complete. **FIXME: correct expression! correct definitions!** Therefore, for every chunk `c` in the store, exactly one `isa(c,t)` constraint has to be in the store. For each `chunk_type_has_slot(t,s)` constraint, a `chunk_has_slot(c,s,v)` constraint has to be defined. If one wants to express, that a chunk has an empty slot, he might use `nil` for the value to indicate that. Note that `nil` must not be a chunk name or chunk-type name.

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.

The straight-forward definition of a data type for chunk specifications is just to use the specification like in definition 4.2: Since `(name isa type slot_1 val_1 \dots slot_n val_n)` is just a list in LISP and specifies a chunk uniquely, a similar Prolog term can be used:

```
:- chr_type chunk_def ---> nil; chunk(any, any, slot_list).
:- chr_type list(T) ---> []; [T | list(T)].
% a list of slot-value pairs
:- chr_type slot_list == list(pair(any,any)).
:- chr_type pair(T1,T2) ---> (T1,T2).
```

This definition states that a chunk is either `nil`, ie. an empty chunk, or a term `chunk(Name, Type, SVP)`, where `SVP` is a list of slot-value pairs. 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 default methods can be implemented as follows:

Listing 4.1: rules for `add_chunk`

```
% empty chunk will not be added
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:

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

% 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(Nam
<=>
  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),
  chunk(V,chunk), % no chunk for slot value found => add chunk of type chunk

do_add_chunk(_) <=> false.
```

FIXME: add very simple solution

Listing 4.2: rules for `add_chunk_type`

```
add_chunk_type(CT, []) <=>
  chunk_type(CT).
add_chunk_type(CT, [S|Ss]) <=>
  chunk_type_has_slot(CT, S),
  add_chunk_type(CT, Ss).
```

FIXME: add duplicate handling

Checking Type-Consistency

FIXME: missing

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* $(B, \Sigma, Holds)$*, where* B *is a set of buffers,* $\Sigma = (C, E, \mathcal{T}, HasSlot, Isa)$ *a type-consistent chunk-store and* $Holds \subseteq B \times (C \cup \{\mathtt{nil}\})$ *a unique and complete relation, that assigns every buffer at most one chunk that it holds. If a buffer* b *is empty, ie. it does not hold a chunk, then* $(b, \mathtt{nil}) \in Holds$*.*

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

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[2].

Destructive Assignment and Consistency

The demand of $Holds$ being unique is a form of destructive assignment as described in [Frü09, p. 32], ie. 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:

```
set_buffer(b, c) \ buffer(b, _) <=> buffer(b, c).
```

This rule ensures that only one `buffer` 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 at a later point. **FIXME: add reference**

In addition, if a new chunk is introduced in 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. This process is discussed later when talking about buffer requests in section **??**.

Buffer States

Another formal detail of the buffer system is that buffers can have various states: *busy*, *free* and *error*. 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

[2] $\forall b \in B\ \exists c \in (C \cup \{\mathtt{nil}\}) : (b, c) \in Holds$

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:

```
(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*, ie. a specification of slot-value pairs that are checked against the chunk in the specified buffer or a *buffer query*, ie. 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, ie. its right hand side will be applied. The LHS is a conjunction of buffer tests, ie. there is no specific order for the tests [**actr_reference**].

Example 4.3 (counting example – left hand side)**.** *The left hand side of 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
==>
  ... )
```

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 Modification

Buffer Request

Buffer Clearing

Example 4.4 (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
)
```

FIXME: add description

Direct Translation of Buffer Tests

An ACT-R production rule of the form

```
(p name
  =buffer1>
    isa     type1
    slot11 val11
    ...
    slot1n val1n
  ...
  =bufferk>
    isa     typek
    slotk1 valk1
    ...
    slotkm valkm
==>
```

```
... )
```

states formally, that:

If $buffer1Holdsc \wedge cIsatype1 \wedge slot_{1,1} = val_{1,1} \wedge \ldots$ **FIXME: extend** 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,slot11,val11),
  ...
  chunk_has_slot(C1,slot1n,val1n),
  ...
  buffer(bufferk,Ck),
  chunk(Ck,typek),
  chunk_has_slot(Ck,slotk1,valk1),
  ...
  chunk_has_slot(Ck,slotkm,valkm)
==>
  ...
```

This rule checks the buffer system for the existence of a buffer specified in the request holding a certain 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.

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: **FIXME: geht besser**

- 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 `chunk` or `chunk_has_slot` constraints 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. This corresponds to the semantics of a ACT-R production rule with variables on the LHS.

FIXME: ist die begründung schlüssig? evtl section über variablen hier einfügen **FIXME: add some words about**

Example 4.5 (counting example in CHR – simple)**.** *The production rule in example 4.4 can be translated to:*

```
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     number =m
  +retrieval>
    isa     count-fact
    first   =m
)
```

4.3.3 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)
...
==> ...
```

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 [**actr_reference**].

```
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-name slot-value
request-parameter ::= a Lisp keyword naming a request parameter provided by th
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 t
```

```
variable which names the buffer that is tested in a buffer-test or dynamic-buffer-test) c
an explicit binding in the production
```

Some of the details in this grammar that have not been discussed yet are presented in the following.

The Order of Rule Applications

Bound and Unbound Variables

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[3], a slot test with a modifier

```
...
=buffer>
  ...
  ~slot  val
...
==>
```

where ~ stands for a modifier in { =, -, <, >, <=, >= }can be translated as follows:

```
buffer(buffer,C),
  ...
  chunk_has_slot(C,slot,V),
```

[3]ie. Prolog predicates

```
  ...
==>
  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 [**actr_reference**].

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 amended with 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 4.6. *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 |
  ...
```

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

will match all chunks which have the same value in `spam` *and* `ham`. **FIXME: CHR translation???**

Empty Slots

An important special case in the semantics of ACT-R production rules is, that if there is a slot test specified, then 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 [**actr_reference**].

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:

```
=buffer>
  isa     foo
  -spam   4
```

matches also a chunk with an empty `spam` slot (`nil` in its `spam` slot).

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

The `X` have to be Prolog terms.

In ACT-R, function calls like `!eval!` or `!bind!` are allowed, but they are ignored in this work.

4.4 Modular Organization

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 4.7 (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 an interaction of the modules is explored.*

Listing 4.3: Definition of Module 1

```
:- module(mod1,[]).
:- use_module(library(chr)).

:- use_module(mod2).

:- chr_constraint a/0, b/0.

a <=> c.
```

Listing 4.4: Definition of Module 2

```
:- module(mod2,[c/0]).
:- use_module(library(chr)).

:- use_module(mod1).

:- chr_constraint a/0, b/0, c/0.

a <=> b.
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 CHR constraint `a` *in listing 4.3 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 4.4, 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 4.3, 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.

Listing 4.5: Simple Interface Module

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

The arguments of such a request are:

BufName The name of the requested buffer, eg. `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`.

This interface will be extended later on.

4.4.3 Components of the Implementation

uml component diagram + discussion

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.

4.5.1 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 4.8. *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, that 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, for each chunk in the store, whose name and type is 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.

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])
4  \ test_slots(N,[(S,V2)|Ss]) <=>
5    unifiable(V1,V2,_) |
6    test_slots(N,Ss).
7
8  chunk_has_slot(N,S,V1)
9  \ test_slots(N,[(S,V2)|_]), match_set([N]) <=>
10     \+unifiable(V1,V2,_) |
11     true.
12
13 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 applies, if there is at least one slot `(S,V2)` that has to be tested and a $HasSlot$ relation of the kind $\mathtt{N} \xrightarrow{\mathtt{S}} \mathtt{V1}$ 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, ie. the both are the same constant or at least one is a variable, then the test passes and the chunk `N` remains in the match set and the rest of the slot tests are performed.

The second rule 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 $\mathtt{N} \xrightarrow{\mathtt{S}} \mathtt{V1}$ and in the request it has been specified that the value of slot `S` has to be `V2`. In this case, $\mathtt{V1} \neq \mathtt{V2}$, 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 both of the first 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. **FIXME: make simpagation rule to simplification ru**

If those rules have been applied exhaustively, only the matching chunks will remain in the match set: If there would be 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 a `collect_matches/1` constraint, that gets the complete match set in its arguments:

```
collect_matches(_) \ match_set(L1), match_set(L2) <=>
  append(L1,L2,L),
  match_set(L).

collect_matches(Res), match_set(L) <=> Res=L.

collect_matches(Res) <=> Res=[].
```

The first rule merges two match sets to one singe 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 no two match sets are in the store, the result of the `collect_matches` operation is the 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[4].

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

```
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 was empty.

With `return_chunk/2` and `get_state/2`, the actual results of the request are computed:

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

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 and 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. **FIXME: add return chunk to default methods of chunk store**

The resulting state of the request is computed as follows: If the result chunk is `nil`, then 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.

4.5.2 Chunk Merging

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 to the CHR implementation easily by adding a `run` constraint and a rule for this constraint, that performs all the initialization work.

FIXME: modify count example from above

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 that simulated the occupation of the production system while a rule is executed.

However, certain buffer actions like buffer requests may take some time until they are finished. The procedural system is free to fire the next rule after all actions have been started[5] 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.

[5]see chapters 2.2 and 2.1.3

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 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 **??** illustrates this approach.

The system just removes such events from the queue and executes them, which leads to new events in the queue. The queue organizes the right order of the events.

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 heart of the scheduler is a *priority queue* which 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:

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 holds objects `q(Time,Priority,Event)`. The priority of such a queue object is composed from the `Time` and the `Priority`. 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)`, *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, there can be defined some rules to make such a queue consistent, ie. every object only has one successor and the queue achieves the time-priority condition:

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

_ --> s <=> false.

A --> B, A --> C <=>
  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, priority does not matter
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 is less than every other object. The other two rules implement the time-priority condition directly.

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. Ie., 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 lead to a linear, serialized list achieving the time-priority condition without gaps.

```
add_q(Time,Priority,Evt) <=>
  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`.

```
de_q(X), s --> A, A --> B <=>
  X = A,
  s --> B.

de_q(X), s --> A <=>
  X = A.

de_q(X) <=> X = nil.
```

The first object just is 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`.

Special Operation for ACT-R

In this implementation, another default method is added to the priority queue:

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 such that the time-priority condition does hold:

```
s --> q(Time,P1,E1) \ after_next_event(Evt) <=>
  NP1 is P1 + 2, % increase priority of first event, so it still has highest priority
  P is P1 + 1, % priority of event that is added, ensured that it is higher than of the r
  de_q(_), % remove head of queue
  add_q(Time,NP1,E1), % add head of queue again with new priority. Will be first again, b
  add_q(Time,P,Evt). % add new event. Will be < than Prio of head but it is ensured that
```

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 time point and 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 a official method of the queue.

4.7.2 Scheduler

The scheduler component is a 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` that only 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.

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 production rule before.

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

If no rule is applicable, the next time that a rule could be applicable 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 condition is performed 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 low priority. This simulates that the procedural module is free again, after all in-place actions of a rule have been fired.

   d) There are two possibilities:

      i. *The next rule matches:* It will be performed like the last rule.

      ii. *No rule matches:* The next time, it could be possible that a rule can fire, is when something in the buffers changed. This only can happen, after 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 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` (so 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 computatic

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 reasonable 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}, schedule_fire.
```

where schedule fire is defined as:

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

As last production rule, there has to be a rule:

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

which removes the fire constraint if still present and states that no rule has been fired (since the fire constraint is still present). In this case, a new `fire` event is scheduled after the next event:

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

4.8 Configuration

4.9 Subsymbolic Layer

4.9.1 Activation of Chunks

4.9.2 Production Utility

5 Compiler

6 Example Models

6.1 The Counting Model

6.2 The Addition Model

6.3 The Semantic Model

6.4 The Fan Model

simplified version of fan model

7 Conclusion

7.1 Inventory: What does already work?

7.2 Future Work

A Source Codes

In diesem Anhang sind einige wichtige Quelltexte aufgeführt.

```
:- use_module(library(chr)).

% just a test

a(X) <=> check(X) | b.

check(13).
check(X) :-
  X <10.
```

B Grammar for Production Rules

The complete grammar for production rules as defined in [**actr_reference**]:

```
production-definition ::= p-name {doc-string} condition* ==> action*
p-name ::= a symbol that serves as the name of the production for reference
doc-string ::= a string which can be used to document the production
condition ::= [ buffer-test | query | eval | binding | multiple-value-binding]
action ::= [buffer-modification | request | buffer-clearing | modification-request | buff
binding | multiple-value-binding | output | !stop!]
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> [direct-value | isa chunk-type request-spec*]
request-spec ::= {slot-modifier} [slot-name | request-parameter] slot-value
request-parameter ::= a Lisp keyword naming a request parameter provided by the buffer sp
direct-value ::= a variable or Lisp symbol
buffer-clearing ::= -buffer-name>
modification-request ::= +buffer-name> slot-value-pair*
buffer-overwrite ::= =buffer-name> direct-value
variable ::= a symbol which starts with the character =
eval ::= [!eval! | !safe-eval!] form
binding ::= [!bind! | !safe-bind!] variable form
+
```

```
multiple-value-binding ::= !mv-bind! (variable ) form
output ::= !output! [ output-value | ( format-string format-args*) | (output-v
output-value ::= any Lisp value or a bound-variable
format-string ::= a Lisp string which may contain format specific parameter pr
format-args ::= any Lisp values, including bound-variables, which will be proc
format-string
bound-variable
 ::= a variable which is used in the buffer-test conditions of the production
variable which names the buffer that is tested in a buffer-test or dynamic-buf
an explicit binding in the production
form ::= a valid Lisp form
```

Bibliography

[Acta] *The ACT-R Homepage*. URL: http://act-r.psy.cmu.edu/ (visited on 03/22/2013) (cit. on p. 3).

[Actb] *The ACT-R Tutorial*. 2012. URL: http://act-r.psy.cmu.edu/actr6/units.zip (cit. on pp. 9 sqq., 15–18, 23, 31).

[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) (cit. on pp. 3 sq., 8 sqq., 14 sq., 17).

[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 (cit. on pp. 4, 7, 10, 12, 18 sq.).

[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) (cit. on p. 16).

[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://www.researchgate.net/publication/2605305_Working_Memory_Activation_Limitations_on_Retrieval/file/d912f50d532cd825c1.pdf (visited on 04/24/2013) (cit. on p. 9).

[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 (cit. on pp. 13, 15 sq.).

[Frü09] Thom Frühwirth. *Constraint Handling Rules*. 08/2009. ISBN: 9780521877763. URL: http://www.constraint-handling-rules.org (cit. on pp. 30, 49).

[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 (cit. on p. 18).

[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) (cit. on p. 3).

[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) (cit. on pp. 3 sq., 15).

[Whi] Jacob Whitehill. "Understanding ACT-R – an Outsider's Perspective". In: (). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.184.8589&rep=rep1&type=pdf (visited on 03/22/2013) (cit. on p. 20).

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