Cognitive Modelling with Term Rewriting

Ivica Milovanović

Johan Jeuring

Technical Report UU-CS-2017-011
June 2017

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Cognitive Modelling with Term Rewriting

Ivica Milovanović (i.milovanovic@uu.nl)
Utrecht University, Princetonplein 5
3584 CC Utrecht, Netherlands

Johan Jeuring (J.T.Jeuring@uu.nl)
Utrecht University, Princetonplein 5
3584 CC Utrecht, Netherlands

Abstract

Term rewriting is a well established formal method used for defining semantics of programming languages, program transformations, automatic theorem proving, symbolic programming, intelligent tutoring system development etc. In this paper, we present a language based on term rewriting as an alternative formalism for modelling cognitive skills. We show how the language overcomes some deficiencies of production systems (compositionality, readability, control-flow etc.) and how, as a consequence, it can help with addressing practical problems raised by the cognitive modelling community.

Keywords: ACT-R; Language for Cognitive Modelling; Production Systems; Term Rewriting

Introduction

Since the pioneering work of Simon and Newell, production systems have been a dominant formalism for cognitive modelling of human behaviour. A production system is a set of condition-action pairs, called production rules or simply productions. Production conditions test against a collection of facts, commonly called working memory. A typical production system interpreter is a loop that continuously selects and fires a production whose condition matches the current working memory content. Firing a production executes its action which can add, remove or modify working memory facts. If more than one production matches at a time, a conflict resolution strategy is used to select and fire a single one. In addition to cognitive modelling, production systems have been used for implementing Knowledge-Based expert Systems, Intelligent Tutoring Systems (ITSs), Business Rule Engines etc.

ACT-R Cognitive Architecture

The ACT-R cognitive architecture (Anderson, 2007; Anderson, Byrne, Douglass, Lebiere, & Qin, 2004; Taatgen & Lee, 2003) is similar to general production systems. It distinguishes procedural knowledge represented as production rules from declarative knowledge represented as a set of untyped key-value pairs called chunks. Keys are commonly called slots. A distinctive characteristic of ACT-R is that the main goal of its development has been to faithfully reproduce human behaviour. For that reason the architecture introduces a number of constraints not present in general-purpose production rule engines. The architecture is divided into a number of modules, each one specialised for performing specific brain functions. The central procedural module stores and interprets production rules and coordinates the function of all the other modules. The visual, aural, motor and speech modules communicate with the outside world. The goal module (also called control state module), the imaginal module (also called problem state module) and the declarative module are specialised for different aspects of internal cognitive processing. The goal module keeps track of the current state in a task, the imaginal module keeps track of the current internal problem representation and the declarative module is responsible for storing long term declarative knowledge and making it available when needed. The modular nature of the architecture reflects the modular nature of the brain itself, and each module can be mapped to a particular brain region (Anderson, 2007). A module is capable of fast internal parallel processing, but can only communicate with other modules through a buffer that can contain a single chunk at a time. The procedural module does not have its own buffer. It can read buffer content of all the other modules, test the content against LHSs of many productions in parallel, but it can only fire a single production at a time. A cognitive skill is a sequence of productions each of which can test against the buffers and perform actions that directly or indirectly modify the buffers.

Cognition is not a deterministic but rather a stochastic process. Knowledge or lack of a skill is not a binary presence or absence of a given production. It is rather a continuous quantity. For example, a skill may be present, but not trained. All the new skills get better with time. To model that, the ACT-R symbolic system is augmented with a neural-like subsymbolic layer. Each chunk has an activation value which determines whether or not it will be retrieved from declarative memory. If its activation is lower than a certain threshold value, a chunk is not retrieved even if it matches the declarative request. Each production has a utility value that is used in conflict resolution. When multiple productions match, the one with highest utility is selected. Both chunk activations and production utilities are evolving quantities. Activations and utilities start low for newly formed chunks and productions respectively, and get higher with practice.

The Programming Model of ACT-R

An ACT-R program consists of a set of production rules and chunks. As a programming language, ACT-R inherits all the characteristics of the production system paradigm. In isolation, productions can be seen as similar to imperative procedures which modify some global state or to functions that take the state as an argument and return a new modified state as a

result. Different from procedures and functions, a production is never called directly in code. A programmer writes rules, and the interpreter decides which rule to fire when. Control flow in production systems is implicit. The production system paradigm is well suited for programming solutions to problems with a large number of independent states and a set of relatively independent actions that can be performed on those states, and not so well suited for programming solutions to problems in well-structured domains, where problem states are related by some formal laws and actions are organised into complex control flows (Buchanan & Shortliffe, 1985). A typical example of the former class are business rule engines, and of the latter class are typical problems that both students and experts solve in STEM fields. Production systems are less suitable for such a domain because of the implicit control flow. As productions can communicate only indirectly through working memory, the only way to establish a desired control flow is to write and read control information to and from that (global) memory. Doing so results in a goto-like coding style which is error-prone and produces code that is difficult to follow, reason about, compose, reuse and maintain. Nevertheless, production systems have often been used for modelling problem solving procedures in STEM fields, particularly in ITSs (Anderson, Corbett, Koedinger, & Pelletier, 1995). The reason for that choice may be partly historical - ITSs emerged from the ACT-R research.

One might argue that implicit control flow in the context of cognitive modelling reflects the flexibility of the brain itself. While that is true, we are investigating the programming aspect of cognitive modelling here. Many published cognitive models[1] are based on complex control flow. This holds in particular for models of learning from instructions or models of metacognitive processes. Both explicit learning and metacognition require complex control flow and planning. The production compilation mechanism (Taatgen & Lee, 2003) eventually produces efficient productions with implicit control flow, but the modeller has to write complex productions with complex control flow. Using ACT-R terminology, production systems become less suitable as a programming model with increased usage of the goal (control state) module. Interestingly, Taatgen (2005) formulated a minimal control principle, which states that humans tend to use a strategy with minimal number of control states while learning a new task. What is more difficult for humans to express as a production system is also more difficult for the brain to execute. Many tasks do require at least some amount of control information and some tasks require complex control. Hence, cognitive modelling can benefit from a programming formalism in which both flexible and complex control can be expressed naturally.

One more difficulty with ACT-R as a programming language, is that modellers have to think explicitly about which modules to use for given actions. Using an appropriate module is important as different combinations of modules for performing the same task can give different predictions of timing, fMRI features etc. Ideally, we would like a modeller to think about task models at a high-level of abstraction and let an interpreter map different actions from a given model to the appropriate modules. An interpreter should know as much as possible about the ACT-R theory and the modeller should focus on the domain-specific features of a particular task.

Heeren and Jeuring have developed a language based on strategic term rewriting (Heeren, Jeuring, & Gerdes, 2010) and used the language to create a number of ITSs and serious games. Besides addressing the software engineering issues of production systems mentioned above, the approach uses well established and rigorous formal methods from formal language theory and practice to analyse and automatically calculate feautures important for intelligent tutoring, such as instructions and feedback. Strategic term rewriting as a computational paradigm allows for writing high-level declarative task models using domain-specific notations. For example, reducing a logic expression to disjunctive normal form is modelled as a set of rewrite rules corresponding to laws of propositional logic (e.g. `not(not(p)) -> p`). A flexible strategy applies all rules extensively until no rule can be applied any more. Such a strategy is similar to strategies used for interpreting production systems. Contrary to production rules, more complex strategies can be expressed naturally. Rewrite rules can be composed to build complex first-class procedures which themselves can be reused in other procedures and composed to build complex hierarchical domains.

Most of the current cognitive architectures do not offer an easy way to reuse knowledge from a model (Taatgen, van Vugt, Borst, & Mehlhorn, 2016). As a result, most published models are isolated theories sharing only common architectural features, and „few cognitive modellers ever use, or even look at, models built by other modellers". Taatgen et al claim that using tools for interactive programming, similar to iPython notebooks, can make cognitive models more accessible and understandable. Term rewriting as a computational paradigm can help addressing all of these problems. Firstly, we can easily compose and reuse terms and rewrite rules, both automatically and by programmers. Secondly, the high-level nature of term rewriting allows for writing cognitive models that are easier to understand (compared to low-level ACT-R models) even by modellers who are not familiar with classical AI programming. Finally, term rewriting is an excellent and natural technique for interactive development.

This paper is organised as follows: we first briefly describe a symbolic language based on term-rewriting. Then we show how the language can be used to develop a cognitive model of the pyramid task (Tenison, Fincham, & Anderson, 2016), and we discuss which characteristics an interpreter for the language should have so that it can give cognitive predictions. We conclude by comparing the introduced approach with related work and giving future work. Although we base our work on ACT-R, the language itself can be used for any cognitive theory by augmenting the interpreter with knowledge of that particular theory.

[1] http://act-r.psy.cmu.edu

Language

The design of our language is inspired by Mathematica (Wolfram Research, Inc., 2017) and Pure[2]. The accompanying example is implemented in Mathematica[3]. Fundamental constructs of the language are expressions (terms) and rewrite rules. For rules of the form `f(e1,e2...)->rhs` we use the terms rule and function interchangeably. Table 1 gives examples of these and other language constructs used in this paper. The interpreter of the language is a higher-order conditional term rewriter with a programmable evaluation strategy. A cognitive model is a set of rewrite rules in the language. A cognitive task is a (potentially complex) term. The interpreter traverses the term following an evaluation strategy and applies rewrite rules to sub-terms. A single traversal step, followed by the application of one or more rewrite rules to a visited sub-term, is roughly equivalent to the three stages in a cognitive task, encoding, solving and responding, as defined in (Tenison et al., 2016). We leave a detailed discussion of the interpreter and evaluation strategies for future work. Fundamental aspects of the interpreter important for cognitive modelling are described in the next section. The model used in this paper is encoded as a simple small term, hence no traversal strategy is necessary.

Model

We use the cognitive model of the pyramid task (Tenison et al., 2016) as our main example in this paper. It requires relatively complex control flow, is expressed in a relatively recent version of ACT-R (6.0), and uses most of the modern ACT-R features. The pyramid task is a simple arithmetic task of the form `base$height` where `base` and `height` are numbers. To evaluate a pyramid, we start from `base` and recursively add `base-1` to it until we reach the total of `height` addends. For example, `8$3` is evaluated as `8+7+6=21`. The (Tenison et al., 2016) model predicts three learning phases, which were corroborated by neural imaging experiments. A learning phase consists of three learning stages - encoding, solving and responding. In the first learning phase there is a long solving stage. With enough practice with a given pyramid, people learn to retrieve the answer for that pyramid without doing the arithmetics, shortening the solving stage significantly. Finally, with even more practice with the same pyramid, people transition to the third learning phase in which they automatically respond without even retrieving the answer. Here we describe the published ACT-R pyramid model informally, using a self-explanatory pseudo-syntax for chunks only occasionally. Rather than trying to follow every technical detail of the model from (Tenison et al., 2016), we use the general theoretical ideas from the model, namely the ACT-R theory of learning by following instructions (Salvucci, 2013; Taatgen & Lee, 2003). We sometimes use slightly more generic productions than those in the published model. For example, instead of having a special production retrieving the sum of two numbers, we assume the existence of more general productions that can retrieve results of any unary, binary or ternary operation respectively. In addition, we use simplified declarative knowledge of arithmetics by treating two-digit numbers as atomic in the same way single-digit numbers are. This is a simplification of a real model, but we want to keep the example model simple. As the entire approach we describe is highly compositional, adding more complicated representation for multi-digit numbers is just a matter of more work. Table 2 gives an informal description of representative parts of the ACT-R pyramid model.

We treat the pyramid task as a formal language with a grammar and semantics. A cognitive model of the pyramid task is an interpreter of that language. To interpret an expression from the language (i.e. to solve a particular pyramid problem), the expression is parsed into an abstract syntax term. After parsing, we apply a series of transformations to the term to obtain a desired solution. Finally, the solution is used somehow, e.g. it is written or pronounced or used as an input to some other sub-problem. The three stages of the interpreting process roughly correspond to the encoding, solving and reporting stages. A typical interpreter of a programming language parses the entire source code into a large abstract syntax tree and performs a series of in-memory traversals and transformations on that tree. Doing the same in interpreters that serve as cognitive models would generally be psychologically unrealistic due to limited capacity of the working memory. Cognitive models are more like sets of small interpreters each of which interprets a small portion of, say, a visual scene (the equivalent of source code). To perform an entire task, small interpreters are composed using a traversal strategy. In case of the pyramid task, however, the entire problem can be parsed and solved in the head. We leave implementing the parsing stage for future work. Here, we assume that the parsing (encoding) stage is already performed, resulting in an abstract syntax term. For example, parsing `8$3` produces the following term:

```
solvePyramid(base->8,height->3)
```
(T1)

The above term has a straight-forward interpretation in the context of ACT-R theory - a goal to solve a pyramid is stored in the goal buffer, and base and height are stored in the imaginal buffer. More generally, the contents of the goal and imaginal buffers are represented as a function call, and a list of arguments of that function, respectively. Different ways of achieving a goal are modelled with different definitions of the function representing that goal. The simplest way of achieving a goal is the automatic response to a stimulus which occurs in the third phase of learning. For example, if there is a goal to solve the pyramid with base 8 and height 3, a production (P1) can fire to immediately store the solution of that pyramid in the imaginal buffer. Solving a particular pyramid is represented by a single procedural skill, i.e. a single production specific for that pyramid. Such productions are formed by the production compilation mechanism and strengthened by subsequent practice with the same pyramid.

[2]https://purelang.bitbucket.io

[3]https://github.com/IvicaM/CogEx

Table 1: Examples of Language Constructs Used in the Paper

expression	rewrite rule	anonymous function	subscript notation
`add,subtract,f`	`add(2,3)->2`	`|add($1,$2)|`	get:`pyramid[base]`
`add(2,3),p(b->8,h->3)`	`b->8`	`|add($,3)|`	set:`pyramid[sum->8]`

Table 2: Representative Portion of the ACT-R Pyramid Model

Productions	Facts	Operators
P1: automatically respond to a goal	(add 8 7 15)	O1/O3:(set base/2 sum/count)
P2: retrieve a fact	(decrement 3 2)	O2:(subtract base 1 term)
P3: harvest retrieved fact	(subtract 7 1 6)	O4:(add term sum sum)
P4: retrieve an operator	(increment 2 3)	O7/O8:(decrement/increment term/count)
P5: interpret retrieved operator		O5:(say sum); O6:iterate or respond

In phase 2 of learning, when there is no specialised production for a given pyramid, a solution for that pyramid is retrieved from declarative memory. In that case, two productions, (P2) and (P3) fire, one which sends a declarative request and one which harvests it. Different from specific productions from phase 3, productions from phase 2 are general and can retrieve and harvest any declarative fact. In phase 1 of learning, retrieval fails as the given pyramid fact does not yet have sufficient activation, and the pyramid task is solved step by step by following instructions. We now show how we model all three phases using functions defined as rewrite rules. As automatic procedural skills and declarative knowledge are fundamental and the simplest building blocks in ACT-R theory necessary to model learning from instructions, we start from phase 3 and gradually move towards phase 1.

Rewrite rules that transform a pyramid to its solution represent procedural knowledge used in phase 3. For example:

```
solvePyramid(base->8,height->3)->21   (R1)
```

Analogously to ACT-R production rules, rewrite rules such as (R1) are stored in procedural memory and have utility values. Given a term, such as (T1), the interpreter matches it against LHSs of all the rules in procedural memory. A matching rule transforms the term. If multiple rules match, the one with highest utility value is selected. As with production firing, it takes 50ms to apply a rewrite rule stored in procedural memory. Up to this point, the main difference of the approach relative to ACT-R is syntactic - rewrite rules are much like overloaded functions from ordinary programming languages. Contrary to ordinary interpreters, which are deterministic and usually select the most specific overload of a function, our interpreter selects an appropriate overload according to the assumptions of the ACT-R theory. We show that using functions instead of productions that operate on buffers with chunks allows us to write cognitive models in a compositional way.

To model phase 2, we represent declarative knowledge. In ACT-R, chunks represent declarative facts, e.g. `(solvePyramid 8 3 21)` (F1). Fact (F1) and its procedural counterpart production (P1) contain exactly the same knowledge. Conceptually, both are functions that take some arguments and return a result. Procedural knowledge is stored in procedural memory and interpreted directly by the ACT-R interpreter while declarative knowledge is stored in declarative memory and interpreted indirectly, through productions similar to (P2) and (P3). Productions (P2) and (P3) act as embedded (in ACT-R) interpreters of declarative knowledge. We use the same representation for both declarative and procedural knowledge. For example, the declarative fact equivalent to (F1), which can be retrieved in phase 2, is represented by the same function (R1) as its procedural counterpart. Contrary to procedural knowledge, declarative knowledge is stored in declarative memory and rules in that memory have activations, analogously to ACT-R chunks. When there is a goal, such as the one represented by the term (T1), the interpreter first matches it against all the rules in procedural memory. If no rule matches or if the utility of any matched rule is low (because the model is not yet in phase 3), the interpreter matches the term against the rules in declarative memory. If no rule matches or if the activation of any matched rule is low (because the model is still in phase 1), a model starts interpreting instructions. Thus we free a modeller from writing generic rewrite rules, equivalent to productions (P2) and (P3), whose only purpose is to interpret declarative knowledge. We do not imply that the skills represented by (P2) and (P3) are innate, i.e. a part of the architecture (term rewriting interpreter in this case). We simply consider the equivalent rewrite rules boilerplate code from the programming perspective. As the ability to retrieve and harvest declarative knowledge is essential for almost any cognitive skill, most modellers would probably want those rules. In a future implementation, we will expose that part of the interpreter as rules that can be removed or customised, if desired.

To summarise, we represent both declarative and the corresponding procedural knowledge as functions stored in the two distinct memories. In learning phase 2, the solution of a problem is stored in declarative memory as a function that

can be fetched and applied. In phase 3, the same function is present in procedural memory and can be applied directly, reflecting automatisation of the given skill. In ACT-R, transition between phases 2 and 3 is modelled as a production compilation that collapses rules (P2) and (P3) into (P1). Note that no such compilation is necessary in our approach as both kinds of knowledge share exactly the same representation. The given declarative knowledge is converted into procedural by copying a corresponding function to the procedural memory. After copying, its utility can increase with subsequent practice, as described in ACT-R theory.

In learning phase 1, a given problem is solved by following the instructions encoded as declarative knowledge. In addition to instructions, declarative knowledge of other facts is usually necessary, as shown in Table 2. We represent those facts with functions stored in the declarative memory. For example:

```
add(8,7)->15 (R2)
decrement(3)->2  (R4)
subtract(7,1)->6 (R4)
```

Operators encode instructions. A typical operator consists of an operation to be done, names of the slots from which arguments are obtained and name of a slot that stores the result. An argument may be a concrete value instead of a slot name. Table 2 shows the operators encoding the instructions for solving a pyramid task. Production (P4) requests an operator of the declarative module that can achieve the current goal. The retrieved operator is executed. As there are different possible kinds of actions, multiple productions are necessary for executing each of those actions. We call this set of productions (P5). Even though productions (P5) are specific to particular actions, they are still fairly generic as they can be used in different domains. Dynamic pattern matching (Anderson, 2007) is necessary to follow the instructions as slot names to be matched against are not known in advance and have to be read from the operators. In the same way (P2) and (P3) can be seen as embedded interpreters of declarative facts, (P4) and (P5) can be seen as embedded interpreters of declarative operators. As with declarative facts, we implement declarative operators as functions stored in declarative memory and implicitly include rules that fetch and interpret operators. Operators are higher order functions that return a function. Primitive generic operators perform basic actions corresponding to reading, comparing and writing slot values. These operators abstract over dynamic pattern matching. Non-primitive operators are composed from primitive operators and fact functions. A cognitive model is programmed as a set of operator functions composed in different ways to achieve given goals. The acquisition of a skill can be seen as „the composition of already-known component skills in novel ways to enable the performance of new skills and tasks" (Salvucci, 2013). The approach we introduce here can thus be seen as an implementation of that view.

We abstract over dynamic pattern matching by using `get`, `set` and `equal` functions. Function `get` takes a list of subscripts and returns a function (getter) that takes an expression and applies the subscripts to the expression. For example, `get(base)` returns `|$[base]|`. Function `set` also takes a list of subscripts, but it returns a function (setter) that takes a list of values and an expression and returns an expression with values stored at the subscripts. For example, `set(term)` returns `|$2[term->$1]|`. Function `equal` takes a list of subscripts and returns a function that tests whether the values at given subscripts are equal or not. Function `not` is a standard logic negation operator. Operator `>>` is a left-associative generalised function composition operator overloaded to work with different kinds of functions and constants. Finally, we define function `update(s,f)` as `get(s)>>f>>set(s)`. Although the implementation of the described constructs may seem involved, writing cognitive models using these constructs is relatively easy. We express almost all the operators from Table 2 as follows:

```
get(base)>>set(sum) (O1)
get(base)>>|subtract($,1)|>>set(term) (O2)
2>>set(count) (O3)
get(term,sum)>>add>>set(sum) (O4)
get(sum)>>say (O5)
update(term,decrement) (O7)
update(count,increment) (O8)
```

`add`, `subtract` etc. are exactly the same functions used to encode declarative knowledge. The main advantage of using the unified function representation for all kinds of knowledge is that we can build a cognitive model incrementally by hierarchically composing functions starting from the most primitive ones. Each intermediate function can be easily understood and tested in isolation and reused in other cognitive models. The only operator missing from the previous list is (O6), which does not encode any action. Instead, it encodes an instruction to test whether `height` and `count` are equal and decides to respond (fetch and execute (O5)) if they are or repeat the sequence (O7-O8-O4) if they are not. We express that kind of iteration by the `while(condition,f)` function which repeatedly applies `f` as long as the `condition` is `true`. With `while`, we express the remaining operator (O6) as `while(not(equal(height,count)),O7>>O8>>O4)`.

Finally, we compose the operators to express the entire pyramid task: `solvePyramid->O1>>...>>O4>>O6>>O5`. The resulting model is a function that can be further reused and composed too. The `solvePyramid` function is not stored in declarative memory as that would not be psychologically plausible. Instead, it reflects the order in which instructions are received, and we use the order to distribute the activation to the individual operators. Fetching the operators based on their activation is similar to the PRIMS approach (Taatgen, 2013). As we use rewrite rules to implement functions, we can trace the execution steps of the model. By applying cognitive interpretation to each step, using the principles described above, we effectively get the same simulation as by running the ACT-R model (Tenison et al., 2016), but with all the benefits of functions. The model starts with rules similar to (R2-R4) and operators (O1-O8) stored in declar-

ative memory, much like the ACT-R model. When there is a goal encoded by the term (T1) and there is no function (R1) in the declarative memory, the model solves the problem by fetching and interpreting the appropriate operators. Some operators, such as (O1), correspond to single productions and can be interpreted in a single step. Other operators require sub-goalling and hence correspond to multiple productions that create sub-goals, fetch additional declarative facts, and return to the main goal by trying to fetch it from declarative memory after the subgoal is achieved. For example, after execution of (O1-O3), the working memory contains `solvePyramid(base->8,height->3,term->7)` (T2) and the operator (O4) is fetched. The operator reads the arguments supplied by `get` and passes them to `add` resulting in the term `add(8,7)`. This interprets as a subgoal and results in fetching (R2) from declarative memory and storing (T2) in declarative memory. If (R2) is fetched and executed, the interpreter retrieves and restores (T2) and stores the result of addition as encoded by `set`. To completely reproduce the results of (Tenison et al., 2016), we interpret setting the count and saying the result as motor actions performed by the corresponding modules. We leave the details of implementing motor actions as compositional functions for future work. When a pyramid is solved, its solution is stored in declarative memory as a function similar to (R1). Subsequent solving of the same pyramid increases the activation of its declarative function. When the activation is high enough, the model transitions to phase 2 and subsequently to phase 3. These transitions are similar to the equivalent transitions in the ACT-R model. However, as we start from fine-grained primitive functions rather than coarse-grained production rules, the composition process also produces many intermediate functions that can be reused in different models, as in PRIMs (Taatgen, 2013). Behaviour similar to PRIMs arises naturally as a consequence of using compositional functions and not as a consequence of our intentional design.

Concluding Remarks

A number of alternatives to using low-level production systems in cognitive modelling have been suggested. (Paik, Kim, & Ritter, 2010) divides those approaches in two groups: reimplementing existing languages in higher-level general purpose languages (1), and creating completely new higher-level languages specialised to writing cognitive models (2). A more recent approach is to program cognitive models directly in a general purpose high-level language by using a library implemented in that language (Salvucci, 2016) (3). Our approach is somewhere between 2) and 3). It is based on a particular higher-level language and in that sense similar to 2). Rather than defining special constructs for abstracting over the low level features of a cognitive architecture (ACT-R in this case), we model all aspects of cognition by first-order and higher-order functions, implemented as rewrite rules. As a result, programming feels much like programming in a general-purpose functional programming language, hence the similarity with 3). The work reported here is preliminary and its purpose is to introduce the approach to the cognitive modelling community. We plan to further develop the language and explore its strengths and weaknesses by using it to develop real world cognitive models. We also want to examine mentioned similarity with PRIMS (Taatgen, 2013) in more detail. Cognitive models implemented as term rewriting systems are actually algebraic structures. An interesting research question we want to explore is if we can utilise those structures and a number of formal methods based on term rewriting, to reason about cognitive models on a more fundamental mathematical level. Term rewriting is closely related to formal languages. We want to further explore theoretical and practical aspects of that relation in the context of cognitive modelling, what can be of special interest to computational and cognitive linguists.

References

Anderson, J. R. (2007). *How can the human mind occur in the physical universe?* (1st ed.). Oxford University Press.

Anderson, J. R., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of the mind. *Psychological Review*, *111*(4), 1036-1050.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences*, *4*(2), 167–207.

Buchanan, B. G., & Shortliffe, E. H. (Eds.). (1985). *Rule-based expert systems: The MYCIN experiments of the Stanford heuristic programming project*. Addison-Wesley.

Heeren, B., Jeuring, J., & Gerdes, A. (2010). Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, *3*(3), 349–370.

Paik, J., Kim, J., & Ritter, F. (2010). Building large learning models with Herbal. In *ICCM'10*.

Salvucci, D. D. (2013). Integration and reuse in cognitive skill acquisition. *Cognitive Science*, *37*(5), 829–860.

Salvucci, D. D. (2016). Cognitive code : An embedded approach to cognitive modeling. In *ICCM'16* (pp. 15–20).

Taatgen, N. A. (2005). Modeling parallelization and flexibility improvements in skill acquisition: from dual tasks to complex dynamic skills. *Cognitive Science*, *29*(3), 421–455.

Taatgen, N. A. (2013). The nature and transfer of cognitive skills. *Psychological review*, *120*(3), 439–71.

Taatgen, N. A., & Lee, F. J. (2003). Production compilation: A simple mechanism to model complex skill acquisition. *Human Factors*, *45*(1), 61-76.

Taatgen, N. A., van Vugt, M. K., Borst, J. P., & Mehlhorn, K. (2016). Cognitive modeling at ICCM: state of the art and future directions. *Topics in Cognitive Science*, *8*(1), 259–263.

Tenison, C., Fincham, J. M., & Anderson, J. R. (2016). Phases of learning: How skill acquisition impacts cognitive processing. *Cognitive Psychology*, *87*, 1–28.

Wolfram Research, Inc. (2017). *Mathematica 11.1*. Champaign, Illinois.