ASP as a Cognitive Modeling Tool: Short-Term Memory and Long-Term Memory

Marcello Balduccini[1] and Sara Girotto[2]

[1] Intelligent Systems, KRL
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com
[2] Department of Psychology
Texas Tech University
Lubbock, TX 79409 USA
sara.girotto@ttu.edu

Abstract In this paper we continue our investigation on the viability of Answer Set Programming (ASP) as a tool for formalizing, and reasoning about, psychological models. In the field of psychology, a considerable amount of knowledge is still expressed using only natural language. This lack of a formalization complicates accurate studies, comparisons, and verification of theories. We believe that ASP, a knowledge representation formalism allowing for concise and simple representation of defaults, uncertainty, and evolving domains, can be used successfully for the formalization of psychological knowledge. In previous papers we have shown how ASP can be used to formalize a rather well-established model of Short-Term Memory, and how the resulting encoding can be applied to practical tasks, such as those from the area of human-computer interaction. In this paper we extend the model of Short-Term Memory and introduce the model of a substantial portion of Long-Term Memory, whose formalization is made particularly challenging by the ability to learn proper of this part of the brain. Furthermore, we compare our approach with various established techniques from the area of cognitive modeling.

1 Introduction

In this paper we continue our investigation on the viability of Answer Set Programming (ASP) [1,2,3] as a tool to formalize psychological knowledge and to reason about it.

ASP is a knowledge representation formalism allowing for concise and simple representation of defaults, uncertainty, and evolving domains, and has been demonstrated to be a useful paradigm for the formalization of knowledge of various kinds (e.g. intended actions [4] and negotiation [5] to name a few examples).

The importance of a precise formalization of scientific knowledge has been known for a long time (see e.g. Hilbert's philosophy of physics). Most notably, formalizing a body of knowledge in an area improves one's ability to (1) accurately study the properties and consequences of sets of statements, (2) compare competing sets of statements, and

(3) design experiments aimed at confirming or refuting sets of statements. In the field of psychology, some of the theories about the mechanisms that govern the brain have been formalized using artificial neural networks and similar tools (e.g. [6]). That approach works well for theories that can be expressed in quantitative terms. However, theories of a more qualitative or logical nature, which by their own nature do not provide precise quantitative predictions, are not easy to formalize in this way.

In [7], we have presented an ASP-based formalization of the mechanism of Short-Term Memory (STM). Taking advantage of ASP's direct executability, i.e. the fact that the consequences of collections of ASP statements can be directly – and often efficiently – computed using computer programs, we have also applied our encoding of STM to practical tasks from the area of human-computer interaction.

However, as argued by various sources (e.g. [8]), to assess the viability of ASP for knowledge representation and reasoning, researchers need to build a number of theories, thus testing ASP's expressive power on a variety of domains. Therefore, in this paper we continue our investigation by using ASP to formalize a substantial part of the mechanism of Long-Term Memory (LTM), focusing in particular on a more complete account of the phenomenon of *chunking* (e.g. [9,10]), where by chunking we mean the brain's ability to recognize familiar patterns and store them efficiently (for example, it has been observed that it is normally difficult for people to remember the sequence CN NIB MMT VU SA, while most people have no problems remembering the sequence CNN IBM MTV USA, because each triplet refers to a familiar concept). Whereas chunking has been partially covered in [7], here we go far beyond the simplified representation of chunks adopted in our previous paper, and tackle the formalization of the LTM's ability to learn chunks upon repeated exposure, which is technically rather challenging. In this paper, we also compare our approach with various established techniques from the area of cognitive modeling.

As we already did in previous papers, it is worth stressing that we do not intend to claim that the psychological models we selected are the "correct" ones. On the contrary, any objections to the models that may come as a result of the analysis of our formalizations are a further demonstration of the benefits of formalizing psychological knowledge.

This paper is organized as follows. We start by providing a background on ASP and on the representation of dynamic domains. Next, we give an account of the mechanics of STM, LTM, and chunking, as it is commonly found in psychology literature. Then, we recap our ASP-based formalization of the mechanics of STM. The following section presents our formalization of the model of LTM and chunking. Finally, we compare with other approaches and conclude with a discussion on what we have achieved and on possible extensions.

2 Answer Set Programming and Dynamic Domains

Let us begin by giving some background on ASP. We define the syntax of the language precisely, but only give the informal semantics of the language in order to save space. We refer the reader to [1,11] for a specification of the formal semantics. Let Σ be a

propositional signature containing constant, function and predicate symbols. Terms and atoms are formed as usual in first-order logic. A (basic) literal is either an atom a or its strong (also called classical or epistemic) negation $\neg a$. A *rule* is a statement of the form:

$$h_1 \ \vee \ \ldots \ \vee \ h_k \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$$

where h_i's and l_i's are ground literals and *not* is the so-called *default negation*. The intuitive meaning of the rule is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, must believe one of h_i's. Symbol $\leftarrow$ can be omitted if no l_i's are specified.

Often, rules of the form $h \leftarrow \text{not } h, l_1, \ldots, \text{not } l_n$, where h is a fresh literal, are abbreviated into $\leftarrow l_1, \ldots, \text{not } l_n$, and called *constraints*. The intuitive meaning of a constraint is that $\{l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n\}$ must not be satisfied. A rule containing variables is interpreted as the shorthand for the set of rules obtained by replacing the variables with all the possible ground terms.

A *program* is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of rules over Σ. We often denote programs just by the second element of the pair, and let the signature be defined implicitly. Finally, an *answer set* (or *model*) of a program Π is one of the collections of its credulous consequences under the answer set semantics. Notice that the semantics of ASP is defined in such a way that programs may have multiple answer sets, intuitively corresponding to alternative views of the specification given by the program. In that respect, the semantics of default negation provides a simple way of encoding choices. For example, the set of rules $\{p \leftarrow \text{not } q. \ q \leftarrow \text{not } p.\}$ intuitively states that either p or q hold, and the corresponding program has two answer sets, $\{p\}$, $\{q\}$.

Because a convenient representation of alternatives is often important in the formalization of knowledge, the language of ASP has been extended with *constraint literals* [11], which are expressions of the form $m\{l_1, l_2, \ldots, l_k\}n$, where m, n are arithmetic expressions and l_i's are basic literals as defined above. A constraint literal is satisfied by a set X of atoms whenever the number of literals from $\{l_1, \ldots, l_k\}$ that are satisfied by X is between m and n, inclusive. Using constraint literals, the choice between p and q, under some set Γ of conditions, can be compactly encoded by the rule $1\{p, q\}1 \leftarrow \Gamma$. A rule of this form is called *choice rule*. To further increase flexibility, a constraint literal $m\{l(a_1, b_1, \ldots), l(c_2, d_2, \ldots), \ldots, l(c_k, d_k, \ldots)\}n$ can also be specified using intentional notation, with an expression $m\{l(X, Y, \ldots) : dom_a(X) : dom_b(Y) : \ldots\}n$, where X and Y are variables, and relations dom_a and dom_b define the domain of the corresponding variables. We refer the reader to [11] for a more detailed definition of the syntax of constraint literals and of the corresponding extended rules. Readers who are familiar with ASP may have noticed that we do not impose restrictions on the dom_i relations above. This is done to keep the presentation short. From the perspective of the implementation, when using ASP parsers that expect each dom_i to be a domain predicate, one would have to use a slightly more elaborate encoding of knowledge.

Because of the dynamic nature of STM and LTM, for their formalization we use techniques from the area of reasoning about actions and change. The key elements of the

representation techniques are presented next; we refer the readers to e.g. [12,13] for more details. *Fluents* are first-order ground terms, and intuitively denote the properties of interest of the domain (whose truth value typically depends on time). For example, an expression of the form $on(block_1, block_2)$ is a fluent, and may mean that $block_1$ is on top of $block_2$. A fluent literal is either a fluent f or its negation ($\neg f$). Actions are also first-order ground terms. For example, $move(block_3, block_2)$ may mean that $block_3$ is moved on top of $block_2$. A set of fluent literals is *consistent* if, for every fluent f, f and $\neg f$ do not both belong to the set. A set of fluent literals is *complete* if, for every fluent f, either f or $\neg f$ belong to the set. The set of all the possible evolutions of a dynamic domain is represented by a *transition diagram*, i.e. a directed graph whose nodes – each labeled by a consistent set of fluent literals – correspond to the states of the domain in which the properties specified by the fluents are respectively true or false, and whose arcs – each labeled by a set of actions – correspond to the occurrence of state transitions due to the occurrence of the actions specified. When complete knowledge is available about a state, the corresponding set of fluent literals is also complete. When instead a set of fluent literals is not complete, that means that the knowledge about the corresponding state is incomplete (e.g. it is unknown whether f or $\neg f$ holds). Incomplete or partial states are typically used to represent uncertainty about domains.

The size of transition diagrams grows exponentially with the increase of the number of fluents and actions; a direct representation is thus usually impractical. Instead, transition diagrams are encoded using an indirect representation, in the form of a domain description in an action language [12], or as a program in ASP (like in [14,15,16]). Here, we adopt the latter approach.

The encoding is based on the notion of a path in the transition diagram from a given initial state, corresponding to a particular possible evolution of the domain from that initial state. The steps in a path are identified by integers (with 0 denoting the initial state), and logical statements (often called *laws*) are used to encode, in general terms, the transitions from one step to the next. The fact that a fluent f holds at a step i in the evolution of the domain is represented by the expression $h(f, i)$, where relation h stands for *holds*. If $\neg f$ is true, we write $\neg h(f, i)$. Occurrences of actions are represented by expressions of the form $o(a, i)$, saying that action a occurs at step i (o stands for *occurs*). An *action description* is a collection of laws describing the evolution of the domain, and in particular the changes of state caused by the execution of actions. For example, the effect of flipping a switch on a connected bulb could be represented by the action description:

$$\begin{aligned} h(on(bulb), S+1) \leftarrow\ & \neg h(on(bulb), S), \\ & h(connected(switch, bulb), S), \\ & o(flip(switch), S). \\ \neg h(on(bulb), S+1) \leftarrow\ & h(on(bulb), S), \\ & h(connected(switch, bulb), S), \\ & o(flip(switch), S). \end{aligned}$$

Given an action description AD, a description of the initial state σ_0 (e.g. $\sigma_0 = \{h(f_1, 0), \neg h(f_2, 0), \ldots\}$), and a sequence of occurrences of actions α (e.g. $\alpha =$

$\{o(a_1, 0), o(a_3, 0), o(a_4, 1), \ldots\}$), the corresponding path(s) in the transition diagram can be computed by finding the answer set(s) of $AD \cup \sigma_0 \cup \alpha$.

3 Short-Term Memory, Long-Term Memory, and Chunking

STM is "the memory storage system that allows for short-term retention of information before it is either transferred to long-term memory or forgotten" [10]. This view is based on the so called *three-stage model of memory* [17]: sensory inputs are first stored in Sensory Memory, which is very volatile and has large capacity; then, a portion of the inputs is processed – and possibly transformed into more rich representations – and moved to STM, which is less volatile than Sensory Memory, but of limited capacity. STM is also often viewed as a working memory, i.e. as a location where information is processed [18]. Finally, selected information is moved to LTM, which has larger capacity and longer retention periods.[3]

Beginning in the 1950s, several studies have been conducted to determine the capacity of STM. Miller [19] reported evidence showing that the capacity of STM in humans is of 7 pieces of information. Later studies have lowered the capacity limit of STM to about 4 pieces of information (e.g. [20]). Interestingly, the limit on the number of pieces of information that STM can hold does not affect directly the *amount* of information (in an information-theoretic sense) that STM can hold. In fact, STM appears to be capable to storing *references* to concepts that are stored in LTM. Although one such reference counts as a single piece of information toward the capacity limit of STM, the amount of information it conveys can be large. For example, it has been observed that it is normally difficult for people to remember the 12-letter sequence CN NIB MMT VU SA, while most people have no problems remembering the sequence CNN IBM MTV USA, because each triplet refers to a concept stored in LTM, and can thus be represented in STM by just 4 symbols [9]. The phenomenon of the detection and use of known patterns in STM is referred to as *chunking*. The patterns stored in LTM – such as CNN or USA – are called *chunks*.

Another limit of STM is that the information it contains is retained only for a short period of time, often set to about 30 seconds by researchers [10]. [4] This limit can be extended by performing *maintenance rehearsal*, which consists in consciously repeating over and over the information that needs to be preserved. To increase the flexibility of our formalization, in our encoding we abstract from specific values for the limits of capacity and retention over time, and rather write our model in a parametric way. This makes it possible, among other things, to use our formalization to analyze the effects

[3] In fact, according to some researchers, LTM has unlimited capacity and retention periods. When information is forgotten, that is because of an indexing problem, and not because the information is actually removed from LTM.

[4] Notice however that the issue of a time limit on the information stored in STM is somewhat controversial – see e.g. [20,18]. For example, according to [18], decay is affected by variables such as the number of chunks that the user is trying to remember, and retrieval interference with similar chunks.

of different choices for these parameters, effectively allowing us to compare variants of the theory of STM.

As we mentioned earlier, LTM serves as a long-term storage of information. The literature describes several types of information that LTM has been found to store and arrange differently, ranging from information about statements which are known to be true (or false), to information about episodes of our life (see e.g. [10]). Researchers claim that, in spite of their being treated differently, the various types of information interact with each other. For example, information about episodes of our life may be used by the brain to derive knowledge about statements which are known to be true or false.

The information stored in LTM appears to serve two main purposes: (1) it can be recalled in order to perform inference by integrating it with information present in STM, and (2) it can be used to perform chunking. The conditions under which the memorization of information in LTM is triggered appear to be quite complex. Researchers have observed that repeated exposure to the same pieces of information causes them to be eventually moved from STM to LTM. Storage is more likely to occur when the exposures are spread over time. Maintenance rehearsal of information in STM can also be used to promote the transfer to LTM. Because of the amount of information stored in LTM, proper indexing plays an important role in being able to access the information at a later time. In order to improve the quality of the indexing for information that is being stored in LTM, one can replace maintenance rehearsal by *elaborative rehearsal*, which consists in consciously thinking about the information of interest and establishing links with knowledge that is already available.

4 A Formalization of Short-Term Memory

This section provides a brief overview of our formalization of STM. For a complete description, the reader is invited to refer to [7].

Using a common methodology in ASP-based knowledge representation, we start our formalization by condensing its natural-language description, as it is found in the literature, into a number of statements – still written in natural language, but precisely formulated. Next, the statements are encoded using ASP.

The statements describing STM are:

1. STM is a collection of symbols;
2. The size of STM is limited to ω elements;
3. Each symbol has an expiration counter associated with it, saying when the piece of information will be "forgotten";
4. New symbols can be added to STM. If a symbol is added to STM when ω elements are already in STM, the symbol that is closest to expiring is removed from STM ("forgotten"). In the case of multiple symbols equally close to expiring, one is selected arbitrarily;
5. When a symbol is added to STM its expiration counter is reset to a fixed value ε;

6. When maintenance rehearsal is performed, the expiration counters of all the symbols in STM are reset to a constant value ε;
7. At each step in the evolution of the domain, the expiration counter is decreased according to the duration of the actions performed;
8. When the expiration counter of a symbol in STM reaches zero, the symbol is removed from STM.
9. *Simplifying assumption:* only a single operation (where by operation we mean either addition of one symbol or maintenance rehearsal) can occur on STM at any given time.

We now show how the statements above are formalized in ASP. Fluent $in\_stm(s)$ says that symbol s (where s is a possibly compound term) is in STM; $expiration(s, e)$ says that symbol s will expire (i.e. will be "forgotten") in e units of time, unless the expiration counter is otherwise altered. Action $store(s)$ says that symbol s is stored in STM; action $main\_rehearsal$ says that maintenance rehearsal is performed on the current contents of STM. Relation $symbol(s)$ says that s is a symbol. Relation $stm\_max\_size(\omega)$ says that the size of STM is limited to ω elements; $stm\_expiration(\varepsilon)$ states that the symbols in STM expire after ε units of time. Finally, in order to update the expiration counters based on the duration of the actions executed at each step, relation $dur(i, d)$ says that the overall duration of step i, based on the actions that took place, is d units of time.

The direct effect of storing a symbol s in STM is described by the following axioms.[5]

$$h(in\_stm(S), I+1) \leftarrow\ symbol(S),\ step(I),\ o(store(S), I).$$
$$h(expiration(S, E), I+1) \leftarrow symbol(S),\ stm\_expiration(E),\ o(store(S), I).$$

The above axioms say that the effect of the action is that s becomes part of STM, and that its expiration counter is set to ε. The next axioms ensure that the size of STM does not exceed its limit when a new symbol is added:

$$\begin{aligned}\neg h(in\_stm(S2), I+1) \leftarrow\ & \\ & o(store(S1), I), \\ & stm\_max\_size(MX),\ curr\_stm\_size(MX, I), \\ & \text{not } some\_expiring(I),\ picked\_for\_deletion(S2, I).\end{aligned}$$
$$\begin{aligned}1\{\ picked\_for\_deletion(S2, I)\ :\ oldest\_in\_stm(S2, I)\ \}1 \leftarrow\ & \\ & o(store(S1), I), \\ & stm\_max\_size(MX),\ curr\_stm\_size(MX, I), \\ & \text{not } some\_expiring(I).\end{aligned}$$

The first axiom states that, if adding a symbol to STM would cause the STM size limit to be exceeded, then one symbol that is the closest to expiring will be removed from

[5] To save space, we drop atoms formed by domain predicates after their first use and, in a few rules, use default negation directly instead of writing a separate rule for closed-world assumption. For example, if p holds whenever q is false and q is assumed to be false unless it is known to be true, we might write $p \leftarrow$ not q instead of the more methodologically correct $\{p \leftarrow \neg q.\ \ \neg q \leftarrow \text{not } q.\}$.

STM. Notice that the simplifying assumption listed among the natural language statements above guarantees that it is sufficient to remove one symbol from STM (lifting the assumption is not difficult, but would lengthen the presentation). The second axiom states that, if multiple symbols are equally close to expiring, then one is selected arbitrarily. From a practical perspective, this encoding allows to consider the evolutions to STM corresponding to all the possible selections of which symbol should be forgotten. The definition of auxiliary relation $oldest\_in\_stm$ can be found in [7]. The reader should however note that the two axioms above extend the encoding of [7] by allowing to deal with cases with symbols equally close to expiring (this situation may occur, for example, as the result of performing a maintenance rehearsal action, whose formalization is discussed later). It is worth pointing out that the second axiom can be easily modified to mimic the behavior formalized in [21], in which an arbitrary symbol (rather one closest to expiring) is forgotten.

The next axiom (together with auxiliary definitions) says that it is impossible for two STM-related actions to be executed at the same time.

$$\leftarrow o(A1, I), o(A2, I),\ A1 \neq A2,\ stm\_related(A1),\ stm\_related(A2).$$
$$stm\_related(store(S)) \leftarrow symbol(S).$$
$$stm\_related(maint\_rehearsal).$$

The direct effect of performing maintenance rehearsal is formalized by an axiom stating that, whenever maintenance rehearsal occurs, the expiration counters of all the symbols in STM are reset:

$$\begin{aligned} h(expiration(S, E), I+1) \leftarrow\ & \\ & stm\_expiration(E), \\ & h(in\_stm(S), I), \\ & o(maint\_rehearsal, I). \end{aligned}$$

The next group of axioms deals with the evolution of the contents of STM over time. Simple action theories often assume that fluents maintain their truth value *by inertia* unless they are forced to change by the occurrence of actions. In the case of fluent $expiration(s, e)$, however, the evolution over time is more complex (and such fluents are then called *non-inertial*). In fact, for every symbol s in STM, $expiration(s, \varepsilon)$ holds at first, but then $expiration(s, \varepsilon)$ becomes false and $expiration(s, \varepsilon - \delta)$ becomes true, where δ is the duration of the latest step, and so on, as formalize by the axioms:

$$noninertial(expiration(S, E)) \leftarrow symbol(S),\ expiration\_value(E).$$
$$\begin{aligned} h(expiration(S, E-D), I+1) \leftarrow\ & \\ & expiration\_value(E), \\ & h(expiration(S, E), I), \\ & dur(I, D), E > D, \\ & \text{not } different\_expiration(S, E-D, I+1), \\ & \text{not } \neg h(in\_stm(S), I+1). \end{aligned}$$
$$\begin{aligned} different\_expiration(S, E1, I) \leftarrow\ & \\ & expiration\_value(E1), \\ & expiration\_value(E2), \\ & E2 \neq E1, \\ & h(expiration(S, E2), I). \end{aligned}$$

The inertia axiom, as well as axioms defining the evolution of fluent $in\_stm(s)$ depending on $expiration(s, e)$ and the auxiliary relations, can be found in [7].

5 A Formalization of Long-Term Memory and Chunking

In this section we describe our formalization of LTM. As mentioned earlier, we focus in particular on the learning of new chunks and on the phenomenon of chunking of the contents of STM by replacing them with references to suitable chunks from LTM.

In the discussion that follows, we distinguish between chunks, described in Section 3, and *proto-chunks*, which are used in the learning of new chunks. Like a chunk, a proto-chunk provides information about a pattern that has been observed, such as the symbols it consists of. However, differently from a chunk, a proto-chunk is also associated with information used to establish when it should be transferred to LTM and transformed into a chunk, such as the number of times it has been observed and the time elapsed from its first detection. Furthermore, proto-chunks cannot be used to perform chunking of the contents of STM.

In order to simplify the presentation, the ASP encoding shown here is focused on dealing with sequences of items, called *tokens*. (Extending the formalization to arbitrary data is not difficult.) A token is either an *atomic token*, such as a digit, or a *compound token*, which denotes a sequence of tokens. Given a sequence of tokens, the expression $seq(n, k)$, denotes the sub-sequence starting at the n^{th} item and consisting of token k. If k is atomic, then the sub-sequence is the token itself. If k is compound, then the sub-sequence consists of the atomic elements of k. In the terminology introduced for the formalization of STM, $seq(n, k)$ is a symbol (see item (1) of Section 4).

As in the previous section, we begin our formalization by providing a natural-language description consisting of precisely formulated statements. To improve clarity, we organize the statements in various categories. Category *LTM model* contains the statements:

1. LTM includes a collection of chunks and proto-chunks;
2. A chunk is a token;[6]
3. Each chunk and proto-chunk is associated with a set of symbols, called elements;
4. A proto-chunk is associated with counters for the times it was detected and the amount of time since its first detection.

Category *chunking* consists of the statements:

5. A chunk is detected in STM if all the symbols that are associated with it are in STM;
6. When a chunk is detected in STM, the symbols associated with it are removed from STM and the corresponding chunk symbol is added to STM;
7. A symbol can be extracted from STM if it belongs to STM, or if it is an element of a chunk that can be extracted from STM; [7]

[6] Note that a proto-chunk is not considered a token.

[7] This statement can of course be applied recursively.

8. *Simplifying assumption:* chunks can only be detected when STM is not in use (i.e. no addition of maintenance rehearsal operations are being performed);
9. *Simplifying assumption:* at every step, at most one chunk can be detected.

Category *proto-chunk detection* contains the statements:

10. A proto-chunk is present in STM if all the symbols that are associated with it are in STM; a proto-chunk is detected in STM if it is present in STM, and the symbols associated with it have not been previously used for proto-chunk formation or detection;
11. When a proto-chunk is detected in STM, the counter for the number of times is incremented;
12. After a proto-chunk is detected, those particular occurrences of its elements in STM cannot be used for detection again except after performing maintenance rehearsal.
13. *Simplifying assumption:* presence of proto-chunks can only be verified when STM is not in use (see statement (8)), and no chunks are detected.

Category *proto-chunk formation* consists of the statements:

14. A new proto-chunk is formed by associating with it symbols from STM that have not yet been used for proto-chunk formation or detection;
15. A proto-chunk can only be formed if there are at least 2 suitable symbols in STM;
16. *Simplifying assumption:* when a proto-chunk is formed, it is associated with *all* the symbols from STM, which have not been previously used for proto-chunk formation or detection;
17. *Simplifying assumption:* proto-chunks can only be formed when STM is not in use (see statement (8)), no chunks are detected, and no proto-chunks are present; [8]

Category *chunk learning* contains the statements:

18. A proto-chunk is replaced by a chunk after being detected at least τ times over a period of π units of time;
19. When a proto-chunk is replaced by a chunk, the elements associated with the proto-chunk become associated with the chunk, and the proto-chunk is removed.

5.1 LTM Model

Category *LTM model* is formalized in ASP by defining suitable fluents and relations (the actual code is omitted to save space). In particular, fluent $in\_ltm(c)$ says that chunk c is in LTM; $chunk\_element(c, s)$ says that s is an element of c; $chunk\_len(c, l)$ says that the number of elements associated with c is l; $is\_pchunk(c)$ says that c is a proto-chunk (and is in LTM); $pchunk\_element(c, s)$ says that s is an element of proto-chunk c; $pchunk\_len(c, l)$ says that the number of elements associated with c is l; $times\_seen(c, n)$ says that proto-chunk c was detected n times; $age(c, a)$ says that the age of proto-chunk c (i.e. the time since its first detection) is a; finally, fluent $considered(s)$ says that symbol s in STM has already been used for proto-chunk formation or detection. Relation $min\_times\_promotion(\tau)$ says that a proto-chunk must be detected τ or more times before it can be transformed into a chunk; $min\_age\_promotion(\pi)$ says that π units of time must have elapsed from a proto-chunk's first detection, before it can be transformed into a chunk.

[8] As a consequence, at every step, at most one proto-chunk can be formed.

5.2 Detection of Chunks

Category *chunking* of statements is formalized as follows. The detection of a chunk is encoded by auxiliary relation $detected(seq(p, c), i)$, intuitively stating that the symbols corresponding to chunk c were detected, at step i, starting at position p in the sequence of tokens stored in STM. The detection occurs, when STM is not in use as per simplifying assumption (8), by checking if there is any chunk whose components are all in STM. If symbols corresponding to multiple chunks are available in STM, only one chunk is detected at every step, as per assumption (9). The choice of which chunk is detected is non-deterministic, and encoded using a choice rule, as follows:

$$
\begin{aligned}
&1\{\ detected(seq(P, C), I) \\
&\qquad\qquad : chunk(C)\ :\ h(in\_ltm(C), I) \\
&\qquad\qquad : \neg chunk\_element\_missing(seq(P, C), I)\ \}1 \leftarrow \\
&\qquad stm\_idle(I), \\
&\qquad chunk\_detectable(I).
\end{aligned}
$$

$$
\begin{aligned}
&chunk\_detectable(I) \leftarrow \\
&\qquad stm\_idle(I), \\
&\qquad chunk(C),\ h(in\_ltm(C), I), \\
&\qquad position(P), \\
&\qquad \text{not}\ chunk\_element\_missing(seq(P, C), I).
\end{aligned}
$$

$$
\begin{aligned}
&\neg chunk\_element\_missing(seq(P, C), I) \leftarrow \\
&\qquad \text{not}\ chunk\_element\_missing(seq(P, C), I).
\end{aligned}
$$

$$
\begin{aligned}
&chunk\_element\_missing(seq(P, C), I) \leftarrow \\
&\qquad chunk\_element\_instance(P, C, I, S), \\
&\qquad \neg h(in\_stm(S), I).
\end{aligned}
$$

$$
\begin{aligned}
&chunk\_element\_instance(P1, C, I, seq(P1 + P2 - 1, T)) \leftarrow \\
&\qquad h(chunk\_element(C, seq(P2, T)), I).
\end{aligned}
$$

$$
\begin{aligned}
&\neg stm\_idle(I) \leftarrow \\
&\qquad o(A, I),\ memory\_related(A).
\end{aligned}
$$

$$
\begin{aligned}
&stm\_idle(I) \leftarrow \\
&\qquad \text{not}\ \neg stm\_idle(I).
\end{aligned}
$$

Relation $detected(seq(p, c), i)$, where c is a chunk, says that sub-sequence $seq(p, c)$ was detected in STM at step i. When such a sub-sequence is detected in STM, we say that *chunk c was detected in STM*. Relation $chunk\_element\_missing(seq(p, c), i)$ says that, at step i, the sub-sequence $seq(p, c)$ is not contained in STM, because one or more elements of c is missing. Relation $chunk\_detectable(i)$ says that there exists at least one chunk c such that the sub-sequence $seq(p, c)$ is in STM for some position p. Relation $chunk\_element\_instance(p, c, i, s)$ says that, at step i, symbol s is an item of the sub-sequence $seq(p, c)$.

It is interesting to note that the components of a detected chunk are allowed to be located anywhere in STM. However, *now that the model is formalized at this level of detail, one cannot help but wonder whether in reality the focus of the mechanism of chunking is on symbols that have been added more recently.* We were unable to find published studies regarding this issue.

The next three axioms state that, when a chunk is detected in STM, the symbols associated with it are replaced by the chunk symbol in STM[9], whose expiration counter is set to ε.

$$\neg h(in\_stm(S), I+1) \leftarrow$$
$$detected(seq(P,C), I),$$
$$chunk\_element\_instance(P, C, I, S).$$

$$h(in\_stm(seq(P,C)), I+1) \leftarrow$$
$$detected(seq(P,C), I).$$

$$h(expiration(seq(P,C), E), I+1) \leftarrow$$
$$stm\_expiration(E),$$
$$detected(seq(P,C), I).$$

Finally, the fact that a symbol can be extracted from STM if it belongs to it, or if it is an element of a chunk that can be extracted from STM, is encoded by axioms:

$$from\_stm(S, I) \leftarrow$$
$$h(in\_stm(S), I).$$

$$from\_stm(S, I) \leftarrow$$
$$from\_stm(seq(P,C), I),$$
$$chunk\_element\_instance(P, C, I, S).$$

It is worth stressing that it is thanks to ASP's ability to encode recursive definitions that this notion can be formalized in such a compact and elegant way.

5.3 Detection of Proto-Chunks

Next, we discuss the formalization of category *proto-chunk detection* of statements. The presence of a proto-chunk in STM is determined by the axioms:

$$pchunk\_present(seq(P,C), I) \leftarrow$$
$$stm\_idle(I),$$
$$\text{not } chunk\_detectable(I),$$
$$h(is\_pchunk(C), I),$$
$$\neg pchunk\_element\_missing(seq(P,C), I).$$

$$some\_pchunk\_present(I) \leftarrow$$
$$pchunk\_present(seq(P,C), I).$$

The first axiom informally states that, if proto-chunk c is in LTM and the sub-sequence of all of its elements is found in STM starting from position p, then proto-chunk is present in STM, and starts at that position. Following simplifying assumption (13), the axiom is only applicable when STM is not in use, and no chunks are detected. The definition of relation $pchunk\_element\_missing(seq(p,c), i)$ is similar to that of relation $chunk\_element\_missing(seq(p,c), i)$, shown above. The axioms that follow are a rather straightforward encoding of statement (13), describing the conditions under

[9] Our simplifying assumption that at most one chunk can be detected at every step ensures that the number of items in STM does not increase as a result of the chunking process.

which a proto-chunk is detected:

$$
\begin{aligned}
&pchunk\_detected(seq(P,C),I) \leftarrow \\
&\qquad pchunk\_present(seq(P,C),I), \\
&\qquad \text{not } \neg pchunk\_detected(seq(P,C),I).
\end{aligned}
$$

$$
\begin{aligned}
&\neg pchunk\_detected(seq(P,C),I) \leftarrow \\
&\qquad pchunk\_present(seq(P,C),I), \\
&\qquad pchunk\_element\_instance(P,C,I,S), \\
&\qquad h(considered(S),I).
\end{aligned}
$$

Whenever a proto-chunk is detected, the counter for the number of times it was observed is incremented. Furthermore, the occurrences, in STM, of the symbols that it consists of are flagged so that they can no longer be used for proto-chunk detection or formation. The flag is removed when maintenance rehearsal is performed:

$$
\begin{aligned}
&h(times\_seen(C,N+1),I+1) \leftarrow \\
&\qquad h(is\_pchunk(C),I), \\
&\qquad pchunk\_detected(seq(P,C),I), \\
&\qquad h(times\_seen(C,N),I).
\end{aligned}
$$

$$
\begin{aligned}
&h(considered(S),I+1) \leftarrow \\
&\qquad pchunk\_detected(seq(P,C),I), \\
&\qquad pchunk\_element\_instance(P,C,I,S), \\
&\qquad h(in\_stm(S),I).\neg h(considered(S),I+1) \leftarrow \\
&\qquad h(in\_stm(S),I), \\
&\qquad o(maint\_rehearsal,I).
\end{aligned}
$$

5.4 Proto-Chunk Formation

The statements in category *proto-chunk formation* are formalized as follows. First, we encode the conditions for the formation of a new proto-chunk listed in statements (14) and (15).

$$
\begin{aligned}
&new\_pchunk(I) \leftarrow \\
&\qquad stm\_idle(I), \\
&\qquad \text{not } chunk\_detectable(I), \\
&\qquad \text{not } some\_pchunk\_present(I), \\
&\qquad num\_avail\_sym\_in\_stm(N,I), \\
&\qquad N > 1.
\end{aligned}
$$

$$
\begin{aligned}
&num\_avail\_sym\_in\_stm(N,I) \leftarrow \\
&\qquad N\{\ avail\_sym(S,I) : symbol(S)\ \}N.
\end{aligned}
$$

$$
\begin{aligned}
&avail\_sym(S,I) \leftarrow \\
&\qquad h(in\_stm(S),I), \\
&\qquad \neg h(considered(S),I).
\end{aligned}
$$

In the axioms above, relation $new\_pchunk(i)$ intuitively states that a new proto-chunk can be formed at step i. Relation $num\_avail\_sym\_in\_stm(n,i)$ says that STM contains n symbols that are available for chunk formation (that is, that have not been previously used for proto-chunk formation or detection). Whenever the conditions for chunk for-

mation are met, the following axioms are used to create the new proto-chunk.

$$
\begin{aligned}
&h(is\_pchunk(p(I)), I+1) \leftarrow \\
&\qquad new\_pchunk(I).\\
&h(pchunk\_element(p(I), seq(P2-P1+1, T)), I+1) \leftarrow \\
&\qquad new\_pchunk(I),\\
&\qquad lowest\_seq\_index(P1, I),\\
&\qquad h(in\_stm(seq(P2, T)), I),\\
&\qquad \neg h(considered(seq(P2, T)), I).\\
&lowest\_seq\_index(P, I) \leftarrow \\
&\qquad h(in\_stm(seq(P, T)), I),\\
&\qquad \text{not } \neg lowest\_seq\_index(P, I).\\
&\neg lowest\_seq\_index(P2, I) \leftarrow \\
&\qquad P2 > P1,\\
&\qquad h(in\_stm(seq(P1, T)), I).
\end{aligned}
$$

The first axiom causes fluent $is\_pchunk(c, i)$ to become true. Function term $p(\cdot)$ is used to assign a name to the new proto-chunk. The name is a function of the current step. Note that naming convention relies on the simplifying assumption that at most one proto-chunk is formed at each step. The second statement determines the elements of the new proto-chunk. The sequence of elements of a proto-chunk is indented start from position 1. However, because the contents of STM expire over time, all the symbols in STM may be associated with a position greater than 1. For this reason, in the axiom we offset the position of each symbol accordingly. The offset is determined by finding the smallest position of a symbol in STM. More precisely, the thirst axiom above says that p is the smallest position that occurs in STM unless it is known that it is not. The last axiom states that p is not the smallest position that occurs in STM if there is a symbol in STM whose position is smaller than p.

The next set of axioms resets the age and the counter for the times the proto-chunk was detected, and describes how the age of a proto-chunk increases from one step to the next according to the duration of the current step (see Section 6 regarding limitations of this representation, and ways to improve it). Because these fluents are *functional* (that is, each describes a function, with the value of the function encoded as a parameter of the function itself), we represent them as non-inertial fluents.[10]

$$
\begin{aligned}
&noninertial(times\_seen(C, N)).\\
&h(times\_seen(p(I), 1), I+1) \leftarrow new\_pchunk(I).\\
&noninertial(age(C, A)).\\
&h(age(p(I), 0), I+1) \leftarrow new\_pchunk(I).\\
&h(age(C, A+D), I+1) \leftarrow \\
&\qquad h(is\_pchunk(C), I),\ h(age(C, A), I),\\
&\qquad dur(I, D),\ \text{not } \neg h(age(C, A+D), I+1).
\end{aligned}
$$

[10] The non-inertial encoding appears to be more compact than the inertial encoding, in that it avoids having to write a rule explicitly stating that $f(n)$ becomes false whenever $f(n+1)$ becomes true.

5.5 Chunk Learning

Finally, we discuss the formalization of the statements in category *chunk learning*. The following axiom determines when the conditions from statement (18) are met:

$$\begin{aligned} can\_be\_promoted(C, I) \leftarrow \\ & min\_times\_promotion(Nmin), \\ & min\_age\_promotion(Amin), \\ & h(is\_pchunk(C), I), \\ & h(times\_seen(C, N), I), \\ & N \geq Nmin, \\ & h(age(C, A), I), \\ & A \geq Amin. \end{aligned}$$

When the conditions are met, a suitable chunk is added to LTM:

$$\begin{aligned} h(in\_ltm(C), I+1) \leftarrow \\ & can\_be\_promoted(C, I). \\ h(chunk\_element(C, S), I+1) \leftarrow \\ & h(is\_pchunk(C), I), \\ & can\_be\_promoted(C, I), \\ & h(pchunk\_element(C, S), I). \end{aligned}$$

The proto-chunk is removed from LTM with axioms such as the following (the other axioms are similar and omitted to save space):

$$\begin{aligned} \neg h(is\_pchunk(C), I+1) \leftarrow \\ & can\_be\_promoted(C, I). \end{aligned}$$

5.6 Experiments

To demonstrate that our formalization captures the key features of the mechanisms of STM, LTM, and chunking, we subjected it to a series of psychological tests. Here we use variations of the *memory-span test*. In this type of test, a subject is presented with a sequence of digits, and is asked to reproduce the sequence [9]. By increasing the length of the sequence and by allowing or avoiding the occurrence of familiar sub-sequences of digits, one can verify the capacity limit of STM and the mechanics of LTM and chunking. Experiments using the basic type of memory-span test were described and used in [7]. Here we expand the tests by introducing a phase in which the subject learns chunks, which can be later used to memorize sequences the subject is presented with. This allows the testing of the detection and formation of proto-chunks, of the learning of chunks, and of the mechanism of chunking.

Because we are only concerned with correctly modeling the mechanisms of interest, we abstract from the way digits are actually read, and rather represent the acquisition of the sequence of digits directly as the occurrence of suitable $store(s)$ actions. Similarly, the final reproduction of the sequence is replaced by checking the contents of STM and LTM at the end of the experiment. As common in ASP, all computations are reduced to finding answer sets of suitable programs, and the results of the experiments are determined by observing the values of the relevant fluents in such answer sets.

From now on, we refer to the above formalization of STM, LTM, and chunking by Π_{MEM}. Boundary conditions that are shared by all the instances of the memory-span test are encoded by the set Π_P of rules, shown below. The first two rules of Π_P set the value of ω to a capacity of 4 symbols (in line with [20]) and the value of ε to 30 time units. The next rule states that each step has a duration of 1 time unit. This set-up intuitively corresponds to a scenario in which STM has a 30 second time limit on the retention of information and the digits are presented at a rate of one per second. The next two rules set the value of τ to 2 detections and the value of π to 3 units of time. The last set of rules define the atomic tokens for the experiments.

$$stm\_max\_size(4).$$
$$stm\_expiration(30).$$
$$dur(I, 1).$$
$$min\_times\_promotion(2).$$
$$min\_age\_promotion(3).$$
$$token(0).\ token(1).\ \ldots token(9).$$

The initial state of STM is such that no symbols are initially in STM. This is encoded by σ_{STM}:

$$\neg h(in\_stm(S), 0) \leftarrow symbol(S).$$

In the first instance, the subject is presented with the sequence $2, 4$. Human subjects are normally able to reproduce this sequence. Moreover, based on our model of LTM and chunking, we expect our formalization to form a proto-chunk for the sequence (intuitively, this simply means that LTM sets up the data structures needed to keep track of future occurrences of the sequence). The sequence of digits is encoded by set $SPAN_1$ of rules:

$$o(store(seq(1, 2)), 0).\ o(store(seq(2, 4)), 1).$$

To predict the behavior of the brain and determine which symbols will be in STM at the end of the experiment, we need to examine the path(s) in the transition diagram from the initial state, described by σ_{STM}, and under the occurrence of the actions in $SPAN_1$. As explained earlier in this paper, this can be accomplished by finding the answer set(s) of $\Pi_1 = \Pi_{MEM} \cup \Pi_P \cup \sigma_{STM} \cup SPAN_1$. It is not difficult to check that, at step 2, the state of STM is encoded by an answer set containing:

$$h(in\_stm(seq(1, 2)), 2),\ h(in\_stm(seq(2, 4)), 2),$$
$$h(expiration(seq(1, 2), 29), 2),\ h(expiration(seq(2, 4), 30), 2),$$

which shows that the sequence is remembered correctly (and far from being forgotten, as the expiration counters show). We also need to check that the proto-chunk has been correctly formed. For that, one can observe that the answer set contains:

$$h(is\_pchunk(p(2)), 3),$$
$$h(pchunk\_element(p(2), seq(1, 2)), 3),$$
$$h(pchunk\_element(p(2), seq(2, 4)), 3),$$
$$h(times\_seen(p(2), 1), 3),$$
$$h(age(p(2), 0), 3).$$

This shows that a proto-chunk named $p(2)$ has indeed been formed for the sequence $2, 4$. The counter for the number of times it has been detected is initially set to 1, and the age of the proto-chunk is set to 0.

Let us now consider another instance, in which the sequence of digits $2, 4$ is presented, and then, after a brief pause, maintenance rehearsal occurs. The corresponding actions are encoded by $SPAN_2 = SPAN_1 \cup \{o(maint\_rehearsal, 4)\}$. The pause is intended to provide enough time for the formation of the proto-chunk before maintenance rehearsal occurs. As explained earlier, maintenance rehearsal helps human subjects learn information. Because of the limit we have chosen for τ and π, we expect $SPAN_2$ to cause a chunk to be formed in LTM. It is not difficult to verify that our formalization exhibits the expected behavior. In fact, according to the answer set of program $\Pi_2 = \Pi_{MEM} \cup \Pi_P \cup \sigma_{STM} \cup SPAN_2$, the state of LTM at the end of the experiment is:

$$h(in\_ltm(p(2)), 7),$$
$$h(chunk\_element(p(2), seq(1, 2)), 7),$$
$$h(chunk\_element(p(2), seq(2, 4)), 7),$$

which shows that a new chunk $p(2)$ describing the sequence $2, 4$ has been added to LTM. As a further confirmation of the correctness of the formalization, it is not difficult to check that the answer set also contains:

$$h(in\_stm(seq(1, p(2))), 8),$$

showing that chunking of the contents of STM occurred as soon as the new chunk became available.

In the next instance, we perform a thorough test of chunk learning and chunking by presenting the subject with a sequence that is, in itself, too long to fit in STM. We assume the subject's familiarity with the area code 806, encoded by the set Γ_1 of axioms:

$$h(in\_ltm(ac(lbb)), 0).$$
$$h(chunk\_element(ac(lbb), seq(1, 8)), 0).$$
$$h(chunk\_element(ac(lbb), seq(2, 0)), 0).$$
$$h(chunk\_element(ac(lbb), seq(3, 6)), 0).$$

Initially, the sequence $5, 8, 5$ is presented, and maintenance rehearsal is performed as before, in order to allow the subject to learn a new chunk. Then, the sequence $8, 0, 6 - 5, 8, 5$ is presented, where "$-$" represents a 1-second pause in the presentation of the digits. This sequence is, in principle, beyond the capacity of STM. However, if chunk learning and chunking are formalized correctly, then the sequence can be chunked using just two symbols, and thus easily fits in STM.[11] The sequence of digits is encoded by $SPAN_3$:

$$o(store(seq(1, 5)), 0).\ o(store(seq(2, 8)), 1).\ o(store(seq(3, 5)), 2).$$
$$o(maint\_rehearsal, 5).$$
$$o(store(seq(1, 8)), 7).\ o(store(seq(2, 0)), 8).\ o(store(seq(3, 6)), 9).$$
$$o(store(seq(4, 5)), 11).\ o(store(seq(5, 8)), 12).\ o(store(seq(6, 5)), 13).$$

[11] The 1-second pause is used to allow sufficient time for the detection of the first chunk. Subject studies have shown that chunk detection does not occur or occurs with difficulty when the stimuli are presented at too high a frequency.

From a technical perspective, the experiment is made more challenging by the fact that $5, 8, 5$ is initially presented so that it starts at position 1, but later it occurs as a subsequence of $8, 0, 6 - 5, 8, 5$, starting from position 4. This allows to verify that chunk learning and chunking occur in a way that is position-independent. It is not difficult to check that at step 8, the state of LTM predicted by our formalization includes:

$$h(in\_ltm(p(3)), 8),$$
$$h(chunk\_element(p(3), seq(1, 5)), 8),$$
$$h(chunk\_element(p(3), seq(2, 8)), 8),$$
$$h(chunk\_element(p(3), seq(3, 5)), 8),$$

which shows that the chunk for $5, 8, 5$ was learned correctly. At the end of the experiment (we select step 15 to allow sufficient time for the chunking of the final triplet to occur), the state of STM predicted by our formalization is:

$$h(in\_stm(seq(1, ac(lbb))), 15),$$
$$h(in\_stm(seq(4, p(3))), 15),$$
$$h(expiration(seq(1, ac(lbb)), 26), 15),$$
$$h(expiration(seq(4, p(3)), 30), 15).$$

As can be seen, the sub-sequence $8, 0, 6$ was chunked using the information encoded by Γ_1, while the sub-sequence $5, 8, 5$ was chunked using the learned chunk for $5, 8, 5$. Summing up, (1) a chunk for $5, 8, 5$ has been learned, and (2) the chunking of the two sub-sequences has occurred, allowing STM to effectively store a sequence of digits that is longer than ω symbols.

Although space restrictions prevent us from formalizing alternative theories of STM, LTM, and chunking, and from performing an analytical comparison, it is worth noting that even the single formalization presented here allows comparing (similar) variants of the theories corresponding to different values of parameters ω, ε, τ, and π. One could for example repeat the above experiments with different parameter values and compare the predicted behavior with actual subject behavior, thus confirming or refuting some of those variants.

6 Discussion and Related Work

In this paper we have continued our investigation on the viability of ASP as a tool for formalizing, and reasoning about, psychological models. Expanding the formalization from [7], we have axiomatized a substantial portion of LTM, with particular focus on the mechanisms of chunk learning and of chunking. The resulting formalization has been subjected to psychological experiments, in order to confirm its correctness and to show its power.

The formalization allows analysis and comparison of theories, and it allows one to predict the outcome of experiments, thus making it possible to design better experiments. Because of the direct executability of ASP, and the availability of ASP-based reasoning techniques, it is also possible to use the formalization in experiments involving e.g. planning and diagnosis (see e.g. [22,23,24]). Various reasons make the formalization

of knowledge of this kind challenging. As we hope to have demonstrated, ASP allows one to tackle the challenges, thanks to its ability to deal with common-sense, defaults, uncertainty, non-deterministic choice, recursive definitions, and evolving domains.

We believe it is difficult to find other languages that allow writing a formalization at the level of abstraction of the one shown here, and that are also directly executable.

An interesting attempt in this direction was made by Cooper et al. in [21]. Their motivation was similar to ours, in that they wanted to define a language suitable for specifying cognitive models, and satisfying four key requirement: "(1) being syntactically clear and succinct; (2) being operationally well defined; (3) being executable; and (4) explicitly supporting the division between theory and implementation detail." The language proposed by Cooper et al., called Sceptic, is an extension of Prolog that introduces a forward chaining control structure consisting of rewrite rules and triggers. The rewrite rules are used to express the procedural control aspects of the program, thus keeping them separate from the formalization of the theory. An example of a Sceptic statement, used to describe memory decay in the Soar 4 architecture (see e.g. [25]), is:

$$
\begin{aligned}
& memory\_decay: \\
& \qquad parameter(working\_memory\_decay, D), \\
& \qquad wm\_match(WME), \\
& \qquad wme\_is\_terminal(WME), \\
& \qquad random(Z), \\
& \qquad Z < D \\
& \Rightarrow wm\_remove(WME).
\end{aligned}
$$

The intuition is that memory decay occurs with the probability specified by parameter $working\_memory\_decay$ (implemented using random number generation); when memory decay occurs, one terminal memory element is deleted by generating a wm_remove event. As can be seen from the structure of the Sceptic statement, the modeling approach adopted by [21] does not explicitly represent the evolution of the domain over time. Using the terminology adopted here, we would say that a particular path in the transition diagram is traversed during the execution of the program, but the path itself is not explicitly represented in the conclusions of the program. Because of that, it is difficult to analyze such path, and to compare alternative paths whenever multiple evolutions of the domain are possible (see e.g. the last memory-span test described in [7]). The approach is also likely to suffer from the known limitations inherited from Prolog – e.g. regarding the handling of defaults and uncertainty, especially when yielding multiple alternative conclusions – as well as the aspects of procedural flavor such as the importance of the order of rules within the program and of the elements of the body within a rule. One rather interesting claim made in [21] is that Sceptic allows to clearly distinguish between a psychological theory and the specification of the "implementation details", such as the definition of when certain conditions occur. To a large extent, this discussion seems to be motivated by the general lower level of abstraction of Prolog, compared to ASP, and thus does not apply to our approach. However, the discussion seems to also apply, at least in part, to the specification of boundary conditions, of which we gave a simple example earlier with the definition of the values of parameters such as τ and π. The intuition is that the specification of the theory and that

of the boundary conditions should be kept clearly separated. We believe that this can be achieved by adopting one of the extensions of ASP that provide modules (see e.g. [26,27]).

In the area of cognitive modeling, quite popular is also the use of cognitive architectures, especially Soar [25] and ACT-R [28]. These are quite sophisticated architectures, not dissimilar from agent architectures, which often cover all the aspects of information processing in the brain, from input acquisition, to output generation. Their parametric and modular structure makes it possible to test alternative theories by replacing suitable modules or adjusting parameter values. In both Soar and ACT-R, the stress seems to be on the definition of a specific architecture, with assumptions being made on the way certain processes occur. Thus, formalization of theories that are not compatible with those assumptions seems difficult. On the other hand, our approach does not enforce any particular assumption, and allows greater freedom of formalization. Furthermore, in the area of cognitive architectures there appears to be no particular interest in, nor means for, the direct theoretical analysis of the properties of the architecture and their components. Whereas we have stressed that ASP allows for inspection of the formalization and comparison of alternative axiomatizations, in these architectures the main tool for analysis appears to be the simulation of the architecture itself, and the analysis of the experimental results (see e.g. [29]; [21] constitutes a remarkable exception).

As a concluding remark, let us stress that the formalization presented here could be made richer in various ways. Our formalization of fluents involving actual time, such as age, is rather simplified. A more sophisticated axiomatization can be obtained by adopting the techniques from e.g. [30]. This would also allow to model continuous decay. Several simplifying assumptions made earlier could be lifted by introducing additive fluents [31]. Furthermore, there may be benefits in describing the interaction between STM and LTM by using an agent architecture framework such as the one in [32].

Acknowledgments
The authors would like to thank Esra Erdem for the useful comments and suggestions.

References

1. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
2. Marek, V.W., Truszczynski, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag, Berlin (1999) 375–398
3. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (Jan 2003)
4. Baral, C., Gelfond, M.: Reasoning about Intended Actions. In: Proceedings of the 20th National Conference on Artificial Intelligence. (2005) 689–694
5. Son, T.C., Sakama, C.: Negotiation Using Logic Programming with Consistency Restoring Rules. In: 2009 International Joint Conferences on Artificial Intelligence (IJCAI). (2009)
6. McCarley, J.S., Wickens, C.D., Gob, J., Horrey, W.J.: A Computational Model of Attention/Situation Awareness. In: Proceedings of the 46th Annual Meeting of the Human Factors and Ergonomics Society. (2002)

7. Balduccini, M., Girotto, S.: Formalization of Psychological Knowledge in Answer Set Programming and its Application. Journal of Theory and Practice of Logic Programming (TPLP) **10**(4–6) (Jul 2010) 725–740
8. Formalizing and Compiling Background Knowledge and its Applications to Knowledge Representation and Question Answering. AAAI 2006 Spring Symposium Series (2006)
9. Kassin, S.: Psychology in Modules. Prentice Hall (2006)
10. Nevid, J.S.: Psychology: Concepts and Applications. second edn. Houghton Mifflin Company (2007)
11. Niemela, I., Simons, P.: Extending the Smodels System with Cardinality and Weight Constraints. In: Logic-Based Artificial Intelligence. Kluwer Academic Publishers (2000) 491–521
12. Gelfond, M., Lifschitz, V.: Action Languages. Electronic Transactions on AI **3**(16) (1998)
13. Gelfond, M.: Representing Knowledge in A-Prolog. In Kakas, A.C., Sadri, F., eds.: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II. Volume 2408., Springer Verlag, Berlin (2002) 413–451
14. Balduccini, M., Gelfond, M., Nogueira, M.: A-Prolog as a tool for declarative programming. In: Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000). (2000) 63–72
15. Delgrande, J.P., Grote, T., Hunter, A.: A General Approach to the Verification of Cryptographic Protocols Using Answer Set Programming. In: 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09). (Sep 2009) 355–367
16. Thielscher, M.: Answer Set Programming for Single-Player Games in General Game Playing. In: 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09). (Sep 2009) 327–341
17. Atkinson, R.C., Shiffrin, R.M.: The Control of Short-Term Memory. Scientific American **225** (1971) 82–90
18. Card, S.K., Moran, T.P., Newell, A.: The Psychology of Human-Computer Interaction. L. Erlbaum Associates Inc. (1983)
19. Miller, G.A.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review **63** (1956) 81–97
20. Cowan, N.: The Magical Number 4 in Short-Term Memory: A Reconsideration of Mental Storage Capacity. Behavioral and Brain Sciences **24** (2000) 87–185
21. Cooper, R.P., Farringdon, J., Fox, J., Shallice, T.: A Systematic Methodology for Cognitive Modelling. Artificial Intelligence **85** (1996) 3–44
22. Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence **138** (2002) 39–54
23. Balduccini, M., Gelfond, M.: Diagnostic reasoning with A-Prolog. Journal of Theory and Practice of Logic Programming (TPLP) **3**(4–5) (Jul 2003) 425–461
24. Balduccini, M., Gelfond, M., Nogueira, M.: Answer Set Based Design of Knowledge Systems. Annals of Mathematics and Artificial Intelligence **47**(1–2) (Jun 2006) 183–219
25. Laird, J.E., Newell, A., Rosenbloom, P.S.: SOAR: An Architecture for General Intelligence. Artificial Intelligence **33** (1987) 1–64
26. Baral, C., Dzifcak, J., Takahashi, H.: Macros, Macro Calls and Use of Ensembles in Modular Answer Set Programming. In: Proceedings of ICLP-06. (2006) 376–390
27. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity Aspects of Disjunctive Stable Models. Journal of Artificial Intelligence Research **35** (Aug 2009) 813–857
28. Anderson, J.R., Bothell, D., Byrne, M.D., Douglass, S., Lebiere, C., Qin, Y.: An Integrated Theory of the Mind. Psychological Review **111**(4) (2004) 1036–1060
29. Halverson, T., Gunzelmann, G., Jr., L.R.M., Dongen, H.V.: Modeling the Effects of Work Shift on Learning in Mental Orientation and Rotation Task. In: 10th International Conference on Cognitive Modeling (ICCM 2010). (Aug 2010)

30. Chintabathina, S., Gelfond, M., Watson, R.: Modeling Hybrid Domains Using Process Description Language. In: Proceedings of ASP '05 Answer Set Programming: Advances in Theory and Implementation. (2005) 303–317
31. Lee, J., Lifschitz, V.: Additive Fluents. In Provetti, A., Son, T.C., eds.: Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning. AAAI 2001 Spring Symposium Series (Mar 2001)
32. Balduccini, M., Gelfond, M.: The AAA Architecture: An Overview. In: AAAI Spring Symposium 2008 on Architectures for Intelligent Theory-Based Agents (AITA08). (Mar 2008)