ABSTRACT
Cognitive modeling is the creation of computer-based processes that mimic human problem-solving and task execution using existing cognitive theories. Cognitive modeling remains a labor-intensive and error-prone activity with little theoretical and tool support. In this paper, we propose an approach to capturing specifications for cognitive models in an incremental and modular way. We use patterns and conflicts to capture specifications and discuss methodological and tool support for specification activities.

KEYWORDS

1. INTRODUCTION
Cognitive modeling is the creation of computer-based processes that mimic human problem-solving. Knowledge-based cognitive models capture task-specific knowledge. They are built to run on cognitive architectures, which are virtual machines capturing general-purpose regularities in human cognition, such as knowledge acquisition (learning), knowledge use (problem solving), and knowledge decay (forgetfulness). Cognitive architectures are an embodiment of cognitive theories. The most notable are Soar [Rosenbloom et al. 1993] and ACT-R [Anderson, Lebiere 1998]. Cognitive models are used in the laboratories for experimental research in cognitive science [Miyake et al. 1995] and in industrial applications to play a role traditionally played by a human (e.g. automatic piloting) [Jones et al. 1995], for simulation and training (e.g. war gaming) [Pew and Mavor, 1998] and in the entertainment industry to create virtual actors, and credible computer game characters [Funge 2000]. Because architectures are by design low-level virtual machines, cognitive models for non-trivial tasks are lengthy and complex. Models created to perform a single task easily exceed thousands of rules [Jones et al. 1995]. Because the state of the art in the software engineering of cognitive models is still in its infancy stage [Ritter et al. 2003], models are typically created from scratch; a model created for one architecture cannot be used with another; the validation of models is exclusively done through extensive testing; and there is little systematic reuse taking place [Kieras and Meyer, 1995, Meyer and Kieras 1999]. One of the major impediments to progress on the above aspects is the absence of formal specifications and formal definition of model correctness. In this paper, we focus on the specification of cognitive models. In section 2, we propose a language and a methodology for writing specifications. In section 3, we define the semantics of cognitive models in terms of trace languages, formulate the problem of model correctness and briefly discuss its verification. In section 4, we summarize and conclude.
2. SPECIFICATION METHODOLOGY AND LANGUAGE

2.1 Challenges to cognitive models specifications

Cognitive models are being developed without the benefits of specification and verification. The two main inhibitors are:

1. Tasks for which cognitive models are developed are generally ill-structured; a correct model is required not only to perform the task, but also to perform it in a human-like manner. As a result, cognitive modelers have shied away from specifying. To validate the models, they relied on testing by comparing the model traces to results generated from human experimentation (e.g. [Anderson and Lebiere1998, Jones et al. 1999, Salvucci 2001]).

2. Because cognitive models emulate the processes of human problem solving, they cannot be adequately captured with the most widely used specification languages and formalisms, which are state-based (pre-condition, post-condition). They are more adequately captured by specification languages used for reactive and real-time systems [Bellini et al. 2000].

These two above inhibitors, while real, can be overcome as follows:

1. The ill-structured nature of the tasks can be addressed by using a specification formalism that allows for partial specifications and by using a specification methodology that supports the incremental formulation of the specification.

2. The fact that cognitive models’ mission is to emulate humans performing a task reflects the dual role of the specifications: (i) Capture the requirement that model perform the task (ii) Capture the requirement that the model perform it in a human-like manner. Specifications must be able to capture both aspects.

In this paper, we propose a specification language and formalism that overcome these difficulties, specifically: The specification language captures both state-based behavior and history-based behavior; the specification language captures functional (competency) requirements as well as cognitive requirements; and the specification language and methodology support the modular specification of tasks in an incremental way.

We illustrate our approach using the task of solving the Towers of Hanoi problem. There are three pegs A, B, and C and three disks: small, medium, and large placed on the pegs. The object of the task is to transfer all three disks from peg A to peg B with the constraint that the disks must be moved one at a time and that no disk can be placed on top of a smaller one.

For this problem, the requirements can be stated informally as follows:

**Functional requirements:** The disks must be moved according to the rules of the game until the final configuration is reached.

**Cognitive requirements:** The model must exhibit some trial and error; it must exhibit some level of learning by not repeating the same mistake; the number of steps and the time taken to solve the problem must be commensurate to that of a human solving the problem.

2.2 Specification Language

We adapt the concept of patterns and conflicts used by Nodine et al. [1995] to specify correct schedules of transactions by specifying compulsory interleaving and forbidden interleaving of operations. For cognitive models, we formulate the specification of what constitutes a valid execution of a task as a set of constraints on the traces generated by the execution. The trace of an execution is the sequence of observable events perceived by or initiated by the agent (model) executing the task. Some constraints, patterns, specify interleaving between events that must take place such as “every pick-up-disk must be followed by a put-down-disk”. Some constraints, conflicts, specify interleaving that must never take place such as “(must never) place disk x on top of disk y where y<x”. The two are illustrated below.

Functional Specifications of the Tower of Hanoi:

**Pattern P1.** Eventually, all disks, the Large, then the Medium, then the Small must be placed on peg B in that order. This pattern is represented by the finite state automaton below. The labels of the arcs (state transitions) represent the event that triggers the transition.
Pattern P2. Every <remove disk x from peg p> must eventually be followed by a <place disk x on peg p>. This pattern ensures that no disks are removed; they can only be moved.

Conflict C1. Once all three disks reach their final configuration, no disk should be moved.

Conflict C2. After <remove disk x from peg p>, any operation other than <place disk x on peg p> represents a conflict. This disallows picking up a second disk before placing back the first one.

We are ready now to define the specification language. We start with an abstract definition of the specification; we then discuss how we formulate its components.

Definition 1 A task specification S is defined by the definition of: an alphabet SA, a set of languages (called pattern languages) PL1, PL2, ..., PLk on SA, and a set of languages (called conflict languages) CL1, CL2, ..., CLl on SA. This specification defines the language SL on alphabet SA where SL = PL1 ∩ PL2 ∩ ... PLk ∩ CL1 ∩ CL2 ... ∩ CLl.

The specification alphabet SA is the set of “observable events” of interest to the specifier. For example, the alphabet of the tower of Hanoi is SA = {<REMOVE, disk, peg>, <PLACE, disk, peg> | disk in {small, medium, large} and peg in {A, B, C}}.

The specification is conceived as a language (set of allowable traces). This language is defined as the intersection of partial specifications provided by the patterns and conflicts.

The traditional approach to defining a language is by defining its underlying grammar. A grammar G is formally defined as a quadruplet G = <T, N, ∑, P> [Denning et al. 1978] where: T is the alphabet (finite set of terminal symbols); N is a finite set of non-terminals; ∑, a sentence symbol not in N or T and P is a finite set of productions of the form α → β.

Given a grammar G, a production α → β, and two strings ω = φ · α · ψ and ω' = φ · β · ψ, we say that ω' is immediately derived from ω in G. This is denoted by ω ⇒ ω'. When o1, o2, ..., on is a sequence of strings such that each is immediately derived from the predecessor, we say that on is derivable from o1. This is denoted by o1 ⇒* on. The language L(G) generated by a formal grammar G is the set of terminal strings derivable from ∑: L(G) = { o | ∑ ⇒* o }.

The interpretation of grammars provided above is the traditional one. Grammars can also be interpreted as defining patterns. For example, consider the grammar G = <T, N, ∑> where T = {read, write}, N = {A}, and P consists of the following two productions:

Σ → read A
A → write.
The traditional interpretation of this grammar defines the language consisting of a single sentence: “read write”, i.e. \( L(G) = \{ \text{“read write”} \} \). By contrast, the pattern interpretation of this grammar defines the constraint that every read must eventually be followed by a write. The following sentences satisfy the above pattern:

1. read write read write read write
2. display
3. write write write
4. open read write read write close

In each of the sentences above, every read is eventually followed by a write. The pattern may occur any number of times (including 0) and different occurrences of a pattern can overlap within the same sentence and share some of their symbols. The patterns’ alphabet is generally only a subset of the language’s alphabet.

We use a two-step process to define this interpretation of grammars. 1. The grammar defines a pattern (which is a language as specified in previous section). 2. The pattern in turn defines a language.

**Definition 2** A Pattern Grammar, PG is defined by a quadruplet \( PG = \langle T, N, P, \sum \rangle \) as defined for languages in general. The language \( P \) generated by a Pattern Grammar PG is called a Pattern. Given a pattern sentence \( ps = s_1, s_2, \ldots, s_p \) and given a sentence \( \omega = \omega_1, \omega_2, \ldots, \omega_n \), sentence \( \omega \) is said to contain a pattern prefix occurrence (of size \( j \)) of pattern sentence \( ps \) at position \( i \) if there exists an injective mapping \( f \) from \([1..j]\) where \( j \leq p \) into \([i...N]\) such that

- \( f(1) = i \) the pattern occurrence starts at position \( i \),
- \( \forall k:1..n: \omega(f(k)) = \omega(k) \) mapped positions contain the same symbols.
- \( \forall k,l:1..n, \text{if} \ k>l, f(k)>f(l) \). The symbols of \( ps \) must appear in \( \omega \) in the same order as they do in \( ps \), i.e. the mapping \( f \) must be monotonous.
- \( \forall k:2..n, \forall j:f(k-1)..f(k)-1: \omega_j \neq s_k \) the mapped position must be the first occurrence of \( s_k \) in \( \omega \) after the occurrence of \( s_{k-1} \).

When \( \omega \) contains a pattern prefix occurrence of size \( p \), we say that \( \omega \) contains a complete pattern occurrence.

In the definition of pattern prefix occurrence, prefixes have a size of at least one. There are cases where we need to allow a prefix of size zero. Consider again pattern \( P1 \) for the tower of Hanoi. If the pattern were to start at state \( S0 \) (instead of \( \sum \)), the pattern would state that “once the large disk is placed on B, eventually, the medium then the small, must also be placed on peg B”. In other words, if the large disk is never placed on peg B, the requirement is irrelevant. Because the pattern starts at \( \sum \) with initial transition \( \lambda \), what it states instead is “once nothing (\( \lambda \)), eventually, the large, then the medium, then the small disks must be placed on peg B”. In other words, the pattern \( P1 \) must always be met. To allow these unconditional patterns, we amend the definition of prefix by allowing empty prefixes for patterns whose first transition is a \( \lambda \) transition.

**Definition 3** Given a Pattern Grammar \( PG = \langle T, N, P, \sum \rangle \), and an alphabet \( A \) such that \( T \subseteq A \), we define the pattern language \( PL(PG,A) = \{ \omega | \omega \in A^*: \text{every prefix occurrence of a pattern string in} \ \omega \ \text{is a complete pattern occurrence of a pattern string (not necessarily the same) in} \ \omega. \} \) A language \( L \) with alphabet \( A \) is said to be compliant with a pattern iff \( L \subseteq PL(PG, A) \).

In the same way that patterns define what must happen, conflicts are used to define what must not happen. For example, the grammar \( G = \langle T, N, P, \sum \rangle \) where \( T = \{ \text{open, close, remove, read} \} \), \( N = \{ A \} \), and \( P \) consisting of the following two productions:

\[ \sum \rightarrow \text{remove} \ (f) \ A \]
\[ A \rightarrow \text{open} \ (f) \]

The language defined by this grammar is \( L(G) = \{ \text{“remove (f), open (f)”} \} \). The conflict defined here is that once an object (\( f \)) is removed, it cannot be opened.

**Definition 4** A Conflict Grammar, CG is defined by a quadruplet \( CG = \langle T, N, P, \sum \rangle \) as defined for languages in general. The language \( C \) generated by a Conflict Grammar CG is called a Conflict. Given a Conflict Grammar \( CG = \langle T, N, P, \sum \rangle \), and an alphabet \( A \), \( T \subseteq A \), we call the conflict language \( CL(CG,A) = \{ \omega | \omega \in \)
A* : there is no complete occurrence of any conflict sentence in \( \omega \). A language \( L \) with alphabet \( A \) is said to be *compliant with a conflict* if and only if \( L \subseteq CL(CG, A) \). □

We revisit the definition of specification given in the beginning of this section by stating how the pattern and conflict languages are defined.

**Definition 5**

A task specification \( S \) is defined by the definition of:
- an alphabet \( A \),
- a set of pattern grammars \( PG_1, PG_2, \ldots PG_k \) on \( A \), and
- a set of conflict grammars \( CG_1, CG_2, \ldots CG_l \) on \( A \),

This specification defines the language \( SL \) on alphabet \( A \):
\[
SL = PL(PG_1, A) \cap \ldots \cap PL(PG_k, A) \cap CL(CG_1, A) \ldots \cap CL(CG_l, A). □
\]

### 2.3 Specification Methodology.

Capturing the right specifications is at the same time critical and challenging. The literature in software specifications is rich with lists of qualities that specification processes must possess and that software specifications must have. In [Mili et al, 1994], we capture the process qualities of a specification by two properties: completeness and minimality. A specification is complete if it captures all of user’s requirements. A specification is minimal if it captures nothing but the user’s requirements. Completeness and minimality cannot be formally proven. They can only be established through redundancy. We define a software specification lifecycle that generates redundancy and uses it to establish the completeness and minimality of a specification. We identify two phases, two activities in the specification lifecycle are:

**Specification generation.** This activity is carried out by the specifier group with input from the user group. It consists of generating the specification from the user concept, possibly adjusting in light of feedback from the verification and validation group.

**Specification validation.** This activity is carried out by the verification and validation group. It consists of generating redundant requirements information from the user concept, then using it to certify the generated specification or to correct it.

The two phases in the specification lifecycle are:

**Specification generation:** During this phase, both the specifier group and the verification and validation group are interacting with the user group to elicit requirements from it.

**Specification validation:** During this phase the verification and validation group checks whether the specification derived by the specifier group satisfies the properties generated by the verification and validation group. Corrective actions are taken accordingly.

The overall process is summarized in Figure 4 below.

<table>
<thead>
<tr>
<th>Specification Generation phase</th>
<th>Specification Validation phase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activity</strong></td>
<td><strong>Activity</strong></td>
</tr>
<tr>
<td>Generating specification</td>
<td>Generating redundant requirements information</td>
</tr>
<tr>
<td>Updating the specification in light of V&amp;V feedback</td>
<td>Matching the specification against Validation information</td>
</tr>
</tbody>
</table>

Figure 2. The Specification Process.

The generic specification process above applies for the specification of cognitive models as well, with each of the activities tailored. We discuss them in turn:

As stated earlier, the specification of a task can be divided into function-level specification and cognitive-level specifications. The process described here can be applied to each of the levels individually or to all of them combined.
The generation of the specification consists of generating individual patterns and conflicts. Tool support for this activity can be greatly beneficial. We are currently developing a tool that generates traces consistent with patterns and conflicts provided by the specifier. Traces generated allow the specifier to tighten the specification by incrementally adding patterns and conflicts or refining the ones provided. We illustrate this with the scenario shown in Figure 3.

In the above scenario, the tool is generating positive examples, i.e., examples of traces that meet all patterns and conflicts. The tool being developed generates traces of negative traces as well, i.e., traces that violate a pattern or a conflict. While positive traces support in the generation of complete specifications by guarding against omissions, negative traces support in the generation of minimal specifications by guarding against over-specification.

3. MODEL CORRECTNESS

We consider a model (program) simulating the actions of an agent executing a task. When the model is run, it generates a trace of events consisting of actions initiated possibly as a reaction to events received from the environment. We call MA, for model alphabet, the set of events and actions that figure in the trace. Given a model M, we define the Trace Language TL (M) as the set of all traces that can be generated by the model. The set of symbols (events and actions) occurring in the traces constitutes the Model’s Alphabet (MA).

Definition 6 Given a model M, with model alphabet MA, and given a specification S, with specification alphabet SA, if SA=MA, the model M is said to be correct with respect to specification S if and only if ML ⊆ SL.

The above definition holds when the two alphabets are identical. The two alphabets are often not identical, though:

1. Specifications are typically concerned with only one subset of the events and actions of the model. For example, if the task is a file manipulation task, the specification may be exclusively concerned with operations affecting the file integrity and the correctness of the results, namely: open, close, read, write. On the other hand, the actual trace of the model is likely to include other events and actions as well such as manipulation and use of the data read and written.

2. Specifications are often captured at a level of abstraction that is higher than that of the model. For example, the specification of the Tower of Hanoi may have the symbol move-disk in its alphabet; for the same application, in the model’s world, move-disk may consist of the sequence: “select disk, select destination, pick up disk, place disk”. This results in a discrepancy between the two alphabets.

(i) The first difference can be addressed easily by adjusting the alphabet parameter in the definition of the specification language SL. Recall that a specification language SL is defined by the definition of Pattern Grammars PGi, conflict grammars CGj, and an alphabet A superset of each of the grammars’ T set. The
alphabet \( A \), can be selected to meet all relevant conditions, namely: 1. \( A \) is a superset of each of the \( T \) set. 2. \( A \) includes all symbols in \( ML \).

(ii) The second difference requires a transformation of one of the languages, i.e. either abstracting \( ML \) or detailing \( SL \). Both cases require a mapping between the abstract language symbols (e.g. move-disk) and the corresponding phrases in the detailed language (e.g. select-disk, remove disk, select-destination, place disk). There may be more than one phrase in the detailed language to capture the symbol in the abstract language. Detailing \( SL \), defined on alphabet \( SA \), to generate \( SL' \) defined on alphabet \( MA \) consists of replacing every occurrence of every \( SA \) symbol by each one of its associated phrases. Abstracting \( ML \), defined on alphabet \( MA \), to generate \( ML' \) defined on alphabet \( SL \) consists of replacing each occurrence of each of the \( ML \) phrases with its abstract mapping. Symbols that remain non-mapped are left as is and added to the alphabet as discussed in (i) above.

We conclude this section by refining the definition of correctness.

**Definition 7** Given a model \( M \), a specification \( S \), given the languages \( SL \) and \( ML \) defined on a common alphabet \( A \), \( M \) is correct with respect to \( S \) if and only if:

- \( ML \) is a superset of \( PL \) (PGi, A) for each pattern grammar PGi.
- \( ML \) is a superset of \( PL \) (CGi, A) for each conflict grammar CGi.

For all of its interest, the above proposition is of limited practical use when it comes to proving a model’s correctness. Comparing two context-free languages is an undecidable problem [Dennings et al. 1978]. In [MacKlem, 2004], we present a graph-based heuristic approach to verifying model correctness with respect to patterns and conflicts.

4. SUMMARY, CONCLUSION

Cognitive models are used increasingly in industrial settings ranging from the entertainment industry to safety-critical and mission-critical applications. Yet, these models are developed outside most norms of software engineering: specification, validation, reuse, and verification. One of the roots to this problem is the difficulty of specifying cognitive models. In this paper, we discuss the characteristics of cognitive models related to specifications and propose a language and a methodology for generating and validating specifications.

The key characteristics of the proposed approach are:

- We distinguish between two layers of the specification. The functional layer and the cognitive layer. This distinction promotes modularity and separation of concerns, and allows the reuse of specifications across cognitive architectures.
- We use patterns and conflicts to characterize valid traces. A specification is constructed as a set of independent patterns and conflicts. By construct, patterns and conflicts are abstract. Furthermore, because they are independent, patterns and conflicts can be constructed in an incremental way.
- We propose a methodology for generating and validating specifications. It is based on establishing a specification’s completeness and minimality through redundancy.

We are currently extending this work in three different ways:

- Establishing a library of reusable specifications. The main cognitive architectures have a set of published models. We are developing specifications for these models to be made available for reuse.
- Creating tools to support specification generation and specification validation. These analytical tools automatically generate positive and negative examples from a specification to help the specifier refine and correct their patterns and conflicts.
- Creating tools to support the verification of cognitive models. We have developed a graph-based approach for the verification process. Many of the steps of this approach can be automated.
ACKNOWLEDGEMENT

This research with partially funded by RDECOM/TARDEC under contract number. Undergraduate students Gillian Gilmer and Caroline Ziemkiewicz contributed to this research with funding from NSF under grant number 0139580.

REFERENCES


Salvucci, D. 2001 Predicting the Effects of In-Car Interfaces on Driver Behavior using a Cognitive Architecture. SIGCHI. Vol. 3, Issue 1, pg 120-127.