Patterns and Conflicts for the Specification and Verification of Cognitive Models

F. Mili, A. MacKlem
Oakland University,
Rochester, MI48309-4478
mili@oakland.edu

C. Adams, S. Dungrani
RDECOM/TARDEC
Warren, MI 48397-5000

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 particular, we propose an approach to capturing specifications for cognitive models in an incremental and modular way. We then discuss ways of proving that that a cognitive model meets its specification.

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 cognitive theories and associated architectures are Soar and ACT-R. They both originated at Carnegie Mellon: SOAR was developed by Newell et al.; ACT-R was developed by Anderson et al. The two architectures have many common aspects and components (cognitive processor, memories, and buffers) and some differences in the processes by which they learn, use, and forget knowledge and in the processes that capture various behavioral moderators such as fatigue, fear, and other emotional factors. Cognitive models are used in the laboratories for experimental research in cognitive science and in industrial applications to play a role traditionally played by a human (e.g. automatic piloting), for simulation and training (e.g. war gaming) and in the entertainment industry to create virtual actors, and credible computer game characters. 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. Because the state of the art in the software engineering of cognitive models is still in its infancy stage, 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; there is little reuse taking place, and when models are reused, the process of adapting and combining models is still tentative, manual, and ad hoc. 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 specification and verification. In section 2, we propose a language and a methodology for writing specifications for cognitive models. In section 3, we define the semantics of cognitive models in terms of trace languages. In section 4 we formulate the problem of model correctness as a comparison between context-free languages. In section 5, we show a heuristic graph-based verification method. We use the towers of Hanoi problem as a running example.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **24 MAY 2004**	2. REPORT TYPE **Journal Article**	3. DATES COVERED **24-04-2004 to 24-05-2004**

4. TITLE AND SUBTITLE **Patterns and Conflicts for the Specification and Verification of Cognitive Models**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **A. MacKlem; F. Mili; C. Adams; S. Dungrani**	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **U.S. Army TARDEC ,6501 E.11 Mile Rd,Warren,MI,48397-5000**	8. PERFORMING ORGANIZATION REPORT NUMBER **#14100**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **U.S. Army TARDEC, 6501 E.11 Mile Rd, Warren, MI, 48397-5000**	10. SPONSOR/MONITOR'S ACRONYM(S) **TARDEC**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) **#14100**

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. 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 particular, we propose an approach to capturing specifications for cognitive models in an incremental and modular way. We then discuss ways of proving that that a cognitive model meets its specification.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Public Release**	**19**	

2. Specification Methodology and Language

2.1 Challenges to cognitive models specifications

Although specification and verification of traditional sequential programs are no longer research issues, they have been almost absent in cognitive models in particular, and artificial intelligence programs in general. Two main inhibitors to formal specification and verification can be identified:

1. Tasks for which cognitive models are developed are ill-structured and hard to specify especially that their requirement is that they perform the task at hand in a human-like manner. As a result, cognitive models have been validated mostly through testing by comparing their traces to results generated from human experimentation [1, 2, 11, 12, 14] to show that they emulate human performance.
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 (precondition, post-condition) [6].

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 an incremental formulation of the specification.
2. The fact that cognitive models' mission is to emulate humans performing a task reflects different components of the requirements: (i) <u>perform the task</u> (ii) <u>perform it in a human-like manner</u>. The "human-like" requirement is typically qualified further by specifying the specific aspects for which the model must be similar to a human. This may include: decisions made and actions taken, rationale used for the decisions, type of errors made, frequency of errors, learning taking place, timing characteristics, task switching, etc. Both components of the requirements: (i) and (ii) need to be captured explicitly before hand.
3. Research in the area of software specifications has generated a wealth of specification languages adapted to capturing requirements of a variety of systems including history-based systems such as concurrent processes and real time systems. Some of these specification formalisms can be adapted to the specification of cognitive models.

In other words, the specification of cognitive models must encompass:

- The specification of functional requirements, i.e. capture the requirement that the task be realized.
- The specification of other aspects of cognitive behavior such as timing, errors, task switching, etc.

The specification language must be able to:

- Capture state-based behavior (pre-condition, post-condition).
- Capture history-based behavior.

The specification language and methodology must support:

- The modular specification of tasks in an incremental way.

2.2 Running Example

We use the example of the Tower of Hanoi throughout this paper. In this problem, there are three pegs A, B, and C and three disks of three different sizes: 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.

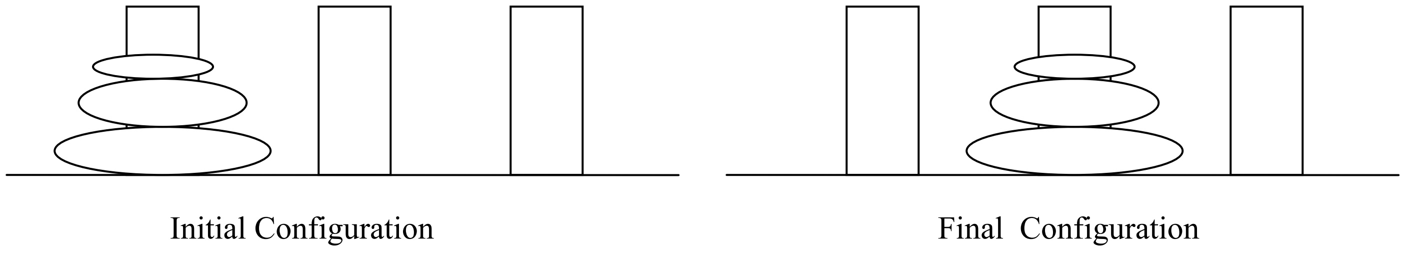


Figure 1: Towers of Hanoi

A state-based specification of this task would consist of a transcription of the information in Figure 1. Obviously, this information is insufficient. A complete specification of the task would include the definition of legal moves as well as the definition of the specific aspects that need to be human like and how.

2.3 Multi-layered specification

For the sake of separation of concerns and reusability, we recommend distinguishing between at least two separate layers in the specification:

- *Functional layer:* in which we capture what constitutes a valid execution of the task at hand. For example, the functional layer specification of the Tower of Hanoi specifies that disks are moved one at a time, that no disk is placed on top of a smaller one, and that eventually the disks are all placed on peg B in their final configuration.
- *Cognitive*-layers: these layers capture specific aspects of the trace that make it human-like. The aspects of interest vary depending on the underlying architecture and depending on the application at hand. The underlying architectures enable the mimicking of some aspects of human behavior; for example, ACT-R is tuned to duplicating the exact timing with which humans make decisions at the 50ms level and tuned to predicting some type of errors. On the other hand, EPIC focuses on emulating the process by which humans interleave the processing of data from different sensors (vision, hearing, touching) and motor actions. The domain of application dictates the characteristics that are of particular interest. Emotional accuracy maybe more critical than intelligent behavior to make a computer game character compelling. For the Tower of Hanoi problem, the cognitive layers would capture such things as: the occurrence of trial and error; the occurrence of learning by not repeating the same mistake; the average time with which the task is executed.

There are a number of advantages to separating the different layers of the specification. On the one hand, this allows us a separation of concerns and facilitates the elicitation of specifications. On the other hand, by capturing the functional layer separately, the same functional specification can be used for a model regardless of the architecture under which it is being implemented and of the application in which it is used. Similarly, some aspects of the cognitive specification can be captured in a generic way and reused for different tasks. Another benefit of the separation is modularity. Even if some aspects of the cognitive layer can be hard to formalize, we can at least capture the functional level and verify the model's functional correctness. The other aspects can, then, be left for testing.

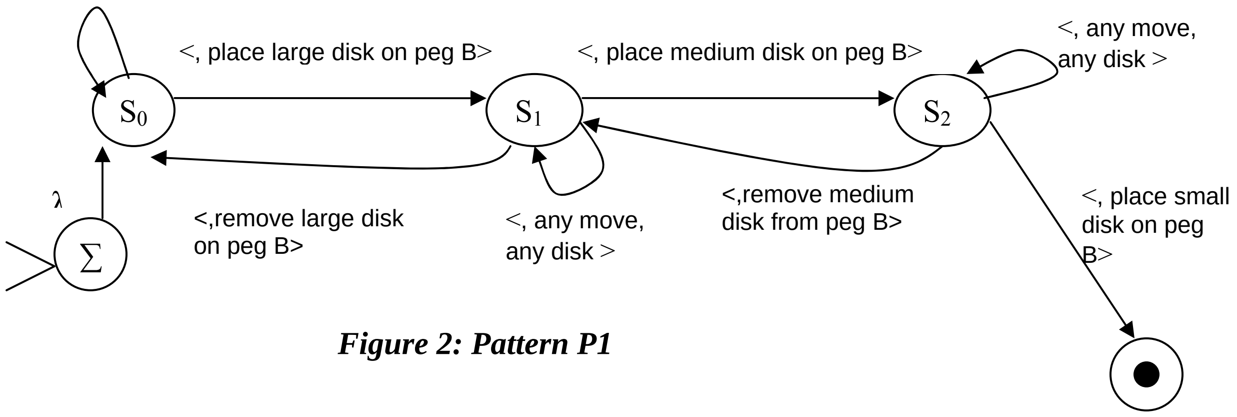
2.5 Specification Language

The specification of what constitutes a valid execution of a task can be though of 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 are best defined positively, such as "every pick-up-disk must be followed by a put-down-disk". Some constraints are best defined negatively such as "Must not place disk x on top of disk y where y<x". Positive constraints are labeled patterns; negative constraints are conflicts.

Before defining the specification language formally, we illustrate it by showing patterns and conflicts from the specification of the Tower of Honoi:

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 initial state labeled with an arrow head represents the beginning of the pattern. The final state labeled with a dot inside the circle represents the (successful) completion of the pattern. The labels of the arcs (state transitions) represent the event that triggers the transition.



<, place large disk on peg B>
<, place medium disk on peg B>
<, any move, any disk >
S0
S1
S2
λ
Σ
<,remove large disk on peg B>
<, any move, any disk >
<,remove medium disk from peg B>
<, place small disk on peg B>

Figure 2: Pattern P1

We will ignore for now the first transition labeled with the empty action, λ and the loop that follows it. The pattern starts when the large disk is placed on peg B taking us to state S_1. From S_1, three events can take place:

- The medium disk is placed on peg B, which takes us to the next state, S_2.
- The large disk is removed from peg B, which takes us back to state S_0.
- Any **other** move which leaves us looping on the same state S_1.

The final state is reached as soon as the small disk is placed on peg B (on top of the medium and the large).

Pattern P2. Every <remove disk x from peg _> must **eventually** be followed by a <place disk x on peg _>. This pattern ensures that no disks are removed; they can only be moved. If we want to insist that the removal of a disk must be **immediately** followed by a placement of the same disk, we need to disallow all other actions until it is placed. This is the subject of conflict C2 below.

Conflict C1. Pattern P1 ensures that eventually disks are placed on peg B. Yet, pattern P1 does not guarantee that when the model stops all disks are still on peg B. We add the requirement that states that any disk movement that follows that point is a conflict.

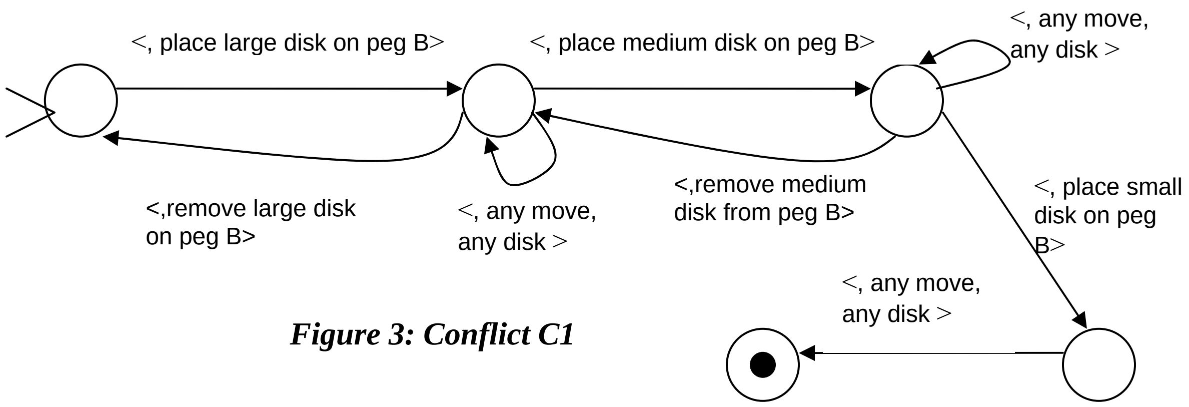

<, place large disk on peg B>
<, place medium disk on peg B>
<, any move, any disk >
<,remove large disk on peg B>
<, any move, any disk >
<,remove medium disk from peg B>
<, place small disk on peg B>
<, any move, any disk >

Figure 3: Conflict C1

As can be seen above, patterns and conflicts are represented in the same way. Whereas patterns state what should happen, i.e. we want to reach the final state; conflicts state what should not happen, i.e. their final states represent conflicts that should not be reached.

Conflict C2. After <remove disk x from peg _>, any operation other than <place disk x on peg _> represents a conflict. This conflicts disallows picking up 2 or more disks before placing any of them back which violates the rules of the game.

The above two patterns and conflicts are a representative sample of the functional requirements of the task Tower of Hanoi. They also illustrate the following characteristics of this specification approach:

- *Modularity:* Writing a specification consists of adding patterns and conflicts until we are satisfied that we have captured all the aspects of interest.
- *Separation of Concerns:* Using patterns and conflicts, the specifier can capture functional requirements and cognitive requirements independently.
- *Encompasses State-based:* Pattern P1 and Conflict C1 together capture the requirement that when the task is done, the post-condition of having the three disks in the correct order placed on peg B. This illustrates the fact that this

approach allows us to capture state-based requirements, in addition to history-based requirements (sequence of events).

- *Methodological support:* In the methodology subsection we discuss methodological support for this activity.

We are ready now to define the specification language.

Definition, Specification.
A task specification S is defined by the definition of:

- an alphabet SA,
- a set of languages (called pattern languages) PL_1, PL_2 , …PL_k on SA, and
- a set of languages (called conflict languages) CL_1, CL_2 , …CL_l on SA,

This specification defines the language SL on alphabet SA where

$$SL= PL_1 \cap PL_2 \cap \ldots PL_k \cap CL_1 \cap CL_2 \ldots \cap CL_l.$$

■

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

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 $\alpha \rightarrow \beta$.
When the productions are all of one of the following forms
$A \rightarrow w$
where A is a non-terminal symbol and w is a non-empty string from $\mathbf{N} \cup \mathbf{T}$.
$\sum \rightarrow w$
$\sum \rightarrow \lambda$
where λ is the empty string,
the language is qualified as context-free.

The relationship between the grammar and the associated language is defined as follows:

Definition: derivation
Given a grammar **G,** a production $\alpha \rightarrow \beta$, and two strings $\omega = \varphi\ \alpha\ \psi$ and $\omega' = \varphi\ \beta\ \psi$, we say that ω' is *immediately derived* from ω in **G.** This is denoted by $\omega \Rightarrow \omega'$. When $\omega_1, \omega_2, \ldots \omega_n$ is a sequence of strings such that each is immediately derived from the predecessor, we say that ω_n is derivable from ω_1. This is denoted by $\omega_1 \Rightarrow^* \omega_n$.

■

Definition: Language L(G)
The language L(**G**) generated by a formal grammar **G** is the set of terminal strings derivable from ∑:

$$L(\mathbf{G}) = \{ \omega | \sum \Rightarrow^* \omega \}$$

■

Pattern Grammars

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, P,** ∑ > where **T**={read, write}, **N=**{A}, and **P** consists of the following two productions:

∑ → read A

A → write.

The traditional interpretation of this grammar defines the language consisting of a single sentence: "read write", i.e. L(G)={"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. read read read write
5. open read write read write close

In each of the sentences above, every read is eventually followed by a write. Sentence 1 illustrates the fact that a pattern may occur any number of times in a sentence therefore the beginning (end) of a pattern is not necessarily the beginning (end) of the sentence. Sentences 2 and 3 illustrate the fact that a pattern can be vacuously met; the sentences do not include read (prefix of the pattern), thus vacuously satisfy the pattern. Sentence 4 illustrates the fact that different occurrences of a pattern can overlap within the same sentence and share some of their symbols; the pattern here appears three times (3 reads) all of which are terminated with a common occurrence of the symbol write. Sentences 2 and 5 illustrate the fact that the sentences' alphabet is not restricted to that of the pattern, but is generally a superset of it.

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. For example, given the pattern grammar "read write" above, and the alphabet {"read", Write"}, it defines the pattern read-write which defines the set of all sentences that meet the pattern, i.e. $(\text{read}^{+}\text{write}^{+} \cup \text{write}^{*})^{*}$.

Definition, Pattern Grammar:

A Pattern Grammar, PG is defined by a quadruplet PG =<**T, N, P,** ∑ > as defined for languages in general.

■

Definition, Pattern:

The language P generated by a Pattern Grammar PG is called a Pattern.

■

Definition, Pattern prefix occurrence, complete pattern occurrence:

Given a pattern sentence ps= $s_1, s_2, \dots s_p$ and given a sentence $\omega = \omega_1, \omega_2, \dots, \omega_n$, sentence ω 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: $\omega_{f(k)} = s_k$ mapped positions contain the same symbols.
- $\forall$ k,l:1..n, if k>l, f(k)>f(l). The symbols of ps must appear in w 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 ω after the occurrence of s_{k-1}.

When ω contains a pattern prefix occurrence of size p, we say that ω contains a *complete pattern occurrence.*

■

Example of pattern prefix occurrence:
Given the pattern sentence: ps= 369 and the sentence ω =12**3**45**6**7867, ω contains a prefix of ps at position 3. The mapping is shown in the bolded, red symbols in ω. There is at most one pattern prefix occurrence starting from one given position; the second 6 in ω cannot be part of a pattern prefix because it is not the first occurrence of 6 that follows 3.

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 S_0 (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 λ, what it states instead is "once *nothing* (λ), 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 λ transition.

Definition, Pattern Language:
Given a Pattern Grammar PG =<**T, N, P,** $\sum$ >, and an alphabet **A** such that **T** $\subseteq$ **A,** we define the pattern language PL(PG,A)={ω| $\omega \in$ $\mathbf{A}^*$: every prefix occurrence of a pattern string in ω is a complete pattern occurrence of a pattern string (not necessarily the same) in ω.}

■

Definition, Compliance with a pattern:
A language L with alphabet A is said to be compliant with a pattern defined by grammar PG iff L $\subseteq$ PL (PG, A).

■

Conflict Grammars

In the same way that patterns define what must happen, conflicts are used to define what must not happen. For example, the grammar G =<**T, N, P,** ∑ > Where **T**={close, read}, **N=**{A}, and **P** consisting of the following two productions:

$\sum \rightarrow$ remove (f) A

$A \rightarrow$ open (f)

The language defined by this grammar is L(G)={"remove (f) , open (f)"}.

The conflict defined here is that once an object (f) is removed, it cannot be opened.

Definition, Conflict Grammar:
A Conflict Grammar, CG is defined by a quadruplet CG =<**T, N, P,** ∑ > as defined for languages in general.

Definition, Conflict:
The language C generated by a Conflict Grammar CG is called a Conflict.

■

Definition, Conflict Language:
Given a Conflict Grammar CG =<**T, N, P,** ∑ >, and an alphabet **A**, **T** ⊆ **A,** we call the conflict language CL(CG,A)={ω| ω∈ $\mathbf{A}^*$: there is no complete occurrence of any conflict sentence in ω.}

■

Definition, Compliance with a conflict:
A language L with alphabet A is said to be compliant with a conflict iff L ⊆ CL (CG, A).

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

Definition, Specification by pattern and conflict grammars.
A task specification S is defined by the definition of:

- an alphabet A,
- a set of pattern grammars PG_1, PG_2 , …PG_k on A, and
- a set of conflict grammars CG_1, CG_2 , …CG_l on A,

This specification defines the language SL on alphabet A:

$$SL= PL(PG_1 , A) \cap \ldots PL(PG_k ,A) \cap CL(CG_1 , A) \ldots \cap CL(CG_l, A).$$

■

2.6 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 organize the specification lifecycle along two orthogonal axes: *phases,* which define a chronological structuring of the process (what gets done when?); and *activities,* which define an organizational structuring of the process (who does what?). We identify two phases, two activities. The partners who participate in the lifecycle are the *user group*, the *specifier group*, and the *verification and validation group*. The 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.

	Specification Generation Activity	Specification validation activity
Specification Generation phase	Generating specification	Generating redundant requirements information
Specification Validation phase	Updating the specification in light of V&V feedback	Matching the specification against Validation information.

Figure 4. 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:

Generating Specifications.

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. The relative independence between the patterns and the conflicts allows their generation to be performed independently. The process of generating patterns and conflicts can be supported by tools. 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 5.

Specifier: enters pattern P1 starting it at state S_0
Tool: generates string
<place small disk on peg B>
Explanation: P1 met because not triggered.
Specifier: edits pattern P1 by adding $\sum$ and associated transitions.
Tool: generates string
<place large disk on B, place medium disk on B, place small disk on B, place small disk on C>
Explanation: P1 met complete occurrence at position 1.
Specifier: adds constraint C1.
Tool: generates string …

Figure 5: Specification Generation support

Generating Redundant Requirements Information.
The role of the Verification and Validation group is to capture properties that can be used to check the completeness and minimality of the specification generated by the specifier group.
The verification and validation group focuses on generating two types of properties:

1. Completeness properties: These are properties that the verification and validation group suspects the specifier group might have missed, making the specification not complete. The verification and validation group can capture such information with patterns and conflicts. Because these properties need to be matched with the specifications, it is best to keep them as simple as possible. Completeness properties can be captured through negative examples: set of traces that should not be allowed by the specification generated. If the specification generated allows these traces, then it is incomplete because too permissive. Examples of completeness properties for the Tower of Hanoi are shown in Figure 6.

NE1: <place large disk on peg B, place medium disk on peg B, place small disk on peg B>
Explanation: No disk can be placed before it is removed.

NE2: <remove large disk, remove medium disk, place large disk on peg B, …>
Explanation: Once a disk is removed, it must be placed immediately.

Figure 6: Sample Completeness properties (Negative Examples)

2. Minimality properties: These are properties that the verification and validation group suspects that the specifier group might have included (when they should not have) making the specification not minimal. The minimality properties can also best be captured using examples, positive in this case. These positive examples must be allowed by the specification generated; if they are not, this would indicate an over-specification. Examples of minimality properties are shown in Figure 7.

PE1: <remove small disk, place small disk on peg B, remove small disk, …>
Explanation: Trial and error should not be disallowed.

Figure 7: Sample of minimality properties (positive examples)

Matching Specifications against redundant information.
During the validation phase, the specification is matched against the completeness and minimality properties. If the specification is such that it "rejects" all negative examples and "accepts" all positive examples, then it is certified. Otherwise, it is reviewed and revised accordingly.

3. Semantics of cognitive models
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 (received from the environment) and actions.

We call MA, for model alphabet, the set of events and actions that figure in the trace.

Definition, Trace Language:
Given a model M, we define the Trace Language TL(M) as the set of traces generated by the model. The set of symbols (events and actions) occurring in the traces constitutes the Model's Alphabet (MA).

■

4. Model Correctness: theoretical formulation
Definition, Correctness:

Given a model M, with model alphabet MA, and 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 $\subseteq$ SL.

■

The above definition holds when the two alphabets are identical. This condition is too restrictive for two reasons:

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 (open, close, read, write), 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 higher level of abstraction than the model's operations. For example, in the specification of the Tower of Hanoi we think of move-disk as an atomic operation; in fact at the model level, this action may be represented by the sequence "select disk; select destination, pick up disk, place disk". Therefore the single symbol "move-disk" in the specification alphabet is represented by the set {select disk, remove disk, select peg, place disk on peg} in the model's alphabet.

The first difference can be addressed easily in light of the way the language SL is defined. The languages PL_i and CL_j are defined function of the given Patterns and Conflicts, and the alphabet A, superset of T. Therefore, it suffices to add the missing symbols to A in order to get PL_i and CL_j and thus SL defined on an alphabet that contains all symbols needed.

The second difference requires a transformation of one of the languages. We discuss both transformations in turn:

- Transform SL defined on SA into SL' defined on MA. The transformation consists of:
  - Mapping each SA symbol into (a) sequence of MA symbols (e.g. move-disk-to-peg mapped to pick-up-disk; select-destination-peg; drop-disk-on-peg.
  - Substitute every occurrence of every symbol of SA in SL (or its grammar) by each one of the corresponding sequence(s) from MA.

- Transform ML defined on MA into ML' defined on SA'. The transformation consists of
  - Mapping each SA symbol into (a) sequence of MA symbols (e.g. move-disk-to-peg mapped to pick-up-disk; select-destination-peg; drop-disk-on-peg.
  - Substitute every sequence of MA symbols in ML that has a mapping into an SL symbol into its corresponding symbol.
  - Leave any non mapped MA symbols as is.

  This transformation is more difficult, but presents the advantage of raising the level of abstraction of ML.

From this point we will assume that A is the common alphabet to the specification and the model.

Definition, Verification:
Given a model M, a specification S, the verification of correctness of M with respect to S is the proof that the language T defined by the traces of M is a subset of the language L defined by S.

■

Proposition:
To prove that a model M is correct with respect to a specification S, it is sufficient to prove that the language T generated by M is

- Is compliant with each of the patterns P_i and
- Is compliant with each of the conflicts C_j .

■

Proof:
This proposition is a direct consequence of the definitions of correctness, specification language, and compliance.

■

Proposition:
The verification of a model with respect to a specification as specified in this chapter is an undecidable problem.

■

Proof:
Theorem about unsolvable problems for context-free languages (Denning et al. 1978).

The fact that the correctness problem is undecidable means that there is no general algorithmic solution for it. In the next section, we restrict the scope and seek an approximate semi automatic approach.

5. Model Correctness: heuristic manual approach

5.1 Triggering graphs

For models written as production systems, which is the case of interest here, we can think of the trace of a model as the sequence of productions fired during one execution. While the set of possible sequences of productions is not easily accessible, an approximation (superset) of it can be easily generated from the triggering graph defined below.

Given a set of productions P_1, P_2, …P_n where each production is a pair <condition, action>, we say that production P_i triggers production P_j if the action of P_i makes the condition of P_j true. For example, consider the pair

```
P1= if goal is A
     then (display A and set goal =to B) and
P2= if goal is B
     then (display B and set goal to C)
```

Because of the presence of parameters and variables in the productions conditions and actions, the triggering of one production by another is history and context dependent. Consider for example the following three productions:

Production P1:

If goal is place-disk
And step is start
Then
Set goal to select-move

Production P2:

If goal is place-disk
And step is select-move
and disksize is 2
Then
Set goal to check-move
Set destination to peg a

Production P3:

If goal is place-disk
And step is select-move
And disksize is 2
Then
Set goal to check-move
Set destination to peg b

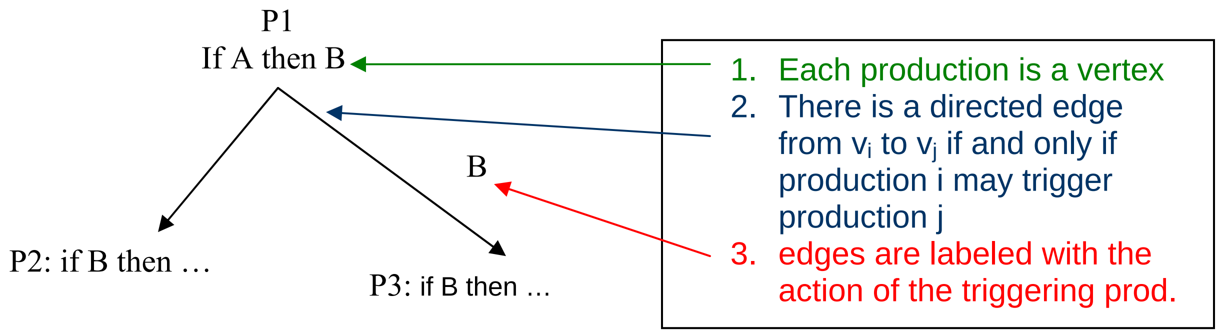
This example illustrates the fact that, in general, we cannot state with certainty whether the execution (firing) of a production will necessarily make the condition of another true. We are generally satisfied with the possibility that this might be the case. Furthermore, triggering graphs are useful to the extent that they are constructed easily and provide us with useful information. In practice, the determination that a production may trigger another is based on only one or a few variables that are easily accessible and take discrete values. In the example above, we would, for instance base the decision on the variable goal alone, thus concluding that P1 triggers P1 and P2.

The triggering graph of the set of productions is defined as follows:

Definition, Triggering Graph:

Given a set of productions P= { P_1, P_2, …P_n }, the triggering graph of P is the directed labeled graph <V, E →A > where V, the set of vertices is P the set of productions (i.e. there is one vertex for each production), and E, the set of directed edges consists of the pair (v_i, v_j) for which production P_i triggers P_j. The label associated with edge (v_i, v_j) is the action of production P_i.

■


P1
If A then B
B
P2: if B then …
P3: if B then …
1. Each production is a vertex
2. There is a directed edge from v_i to v_j if and only if production i may trigger production j
3. edges are labeled with the action of the triggering prod.

We will assume that every cognitive model admits a root production, i.e. the production that will always be executed first, not triggered by any other. This assumption is not restrictive because if it is not met, we can always add a production that triggers all others with a label λ.

With this assumption, the triggered graph of a model is a rooted directed graph.

<u>*Definition, Triggering trace:*</u>
Given a triggering graph with labeling alphabet A, each path on the graph defines a string of A symbols: the sequence of labels on the edges traversed. *The triggering trace language* of a model M TTL(M, G) is the set of strings defined by all paths on the triggering graph G of M that start in the root of the graph.

■

<u>*Proposition, TL(M) subset TTL(M, G):*</u>
Given a model M, a triggering graph G of M, every trace of M is also in TTL(M,G). In other words, TL(M) $\subseteq$ TTL(M, G).

■

5.2 Finite State automata
We restrict this study to regular grammars. A pattern is a rooted directed labeled graph.

5.3 Correction heuristics
Formulate axioms concerning patterns and conflicts rel. triggering graph.
If pattern graph subset of triggering graph, then pattern met.

Need to look for all prefixes.

If conflict graph is not in triggering graph then conflict is met.

5.4 Graph matching

6. Summary, Conclusion

Acknowledgements
This research with partially funded by RDECOM/TARDEC. Undergraduate students Gillian Gilmer and Caroline Ziemkiewicz contributed to this research with funding from NSF.

References

Peter J. Denning, Jack B. Dennis, Joseph E. Qualitz *Machines, Languages, and Computation*, Prentice Hall, 1978.

1. Salvucci, D. Predicting the Effects of In-Car Interfaces on Driver Behavior using a Cognitive Architecture. *SIGCHI*. Vol. 3, Issue 1, 2001, pg 120-127.
2. Chong, R. S., & Laird, J. E. (1997). *Identifying dual-task executive process knowledge using EPIC-Soar*. In M. Shafto & P. Langley (Eds.), Proceedings of the Nineteenth Annual Conference of the Cognitive Science Society (pg. 107-112). Hillsdale, NJ: Erlbaum.
3. Krueger, C. Software Reuse. *ACM Competing Surveys (CSUR).* Vol. 24, Issue 2, 1992, pg. 131-183.

4. Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*. Vol. 12, Issue 10, 1969, pg. 576-580.
5. Hoare, C. A. R. et al. Laws of programming. *Communications of the ACM*. Vol. 30, Issue 8, 1987, pg. 672-686.
6. Apt, K. R. Ten years of Hoare's logic: A survey – Part 1. *ACM Trans. Prog. Lang. Syst*. Vol. 3, Issue 4, 1981, pg. 431 - 483.
7. Bergstra, J. A., and Tucker, J. Expressiveness and completeness of Hoare's logic. *J. Comput. Syst. Sci*. 25 (1982), pg. 267-284.
8. Anderson, J. R., Bothell, D., Byrne M. D. & Lebiere, C.. An Integrated Theory of the Mind. *Psychological Review*. (2002). pg. 105-172
9. Byrne, M. D., (in press). ACT-R/PM and menu selection: Applying a cognitive architecture to HCI. *International Journal of Human-Computer Studies*.
10. Ritter, F. E., Shadbolt, N. R., Elliman, D., Young, R., Gobet, F., & Baxter, G. D. Techniques for modeling human and organizational behaviour in synthetic environments: A supplementary review. Wright-Patterson Air Force Base, OH: Human Systems Information Analysis Center. 2003
11. Ritter, F. E., Baxter, G. D., Jones, G., & Young, R. M. Supporting cognitive models as users. *ACM Transactions on Computer-Human Interaction*, 7(2), pg.141-173. 2000.
12. Bass, E.J, Baxter, G.D., & Ritter, F.E. Creating models to control simulations: A generic approach. *AI and Simulation of Behaviour Quarterly*, 93, pg.18-25. 1995.
13. Nerb, J., Ritter, F. E., & Krems, J. Knowledge level learning and the power law: A Soar model of skill acquisition in scheduling. *Kognitionswissenschaft [Journal of the German Cognitive Science Society]* Special issue on cognitive modelling and cognitive architectures, D. Wallach & H. A. Simon (eds.). pg. 20-29. 1999 .
14. Ritter, F. E., & Major, N. P. Useful mechanisms for developing simulations for cognitive models. *AI and Simulation of Behaviour Quarterly*, 91(Spring), pg. 7-18. 1995.
15. Young, R. M., & Ritter, F. E. Report on the Second European Conference on Cognitive Modelling. *AI and Simulation of Behaviour Quarterly*, 101, pg. 10-11. 1999.
16. Salvucci, D. D., & Siedlecki, T. "Toward a unified framework for tracking cognitive processes". To appear in *Proceedings of the 25th Annual Conference of the Cognitive Science Society*. Mahwah, NJ: Lawrence Erlbaum Associates.
17. Salvucci, D. D., & Lee, F. J. (2003). Simple cognitive modeling in a complex cognitive architecture. In *Human Factors in Computing Systems: CHI 2003 Conference Proceedings* New York: ACM Press. 2003. pg. 265-272.
18. Kieras, D.E., Meyer, D.E., Mueller, S., & Seymour, T. "Insights into working memory from the perspective of the EPIC architecture for modeling skilled perceptual-motor and cognitive human performance". In A. Miyake and P. Shah (Eds.), *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control*. New York: Cambridge University Press. 1999. pg.183-223.
19. Meyer, D. E., & Kieras, D. E. "Precis to a practical unified theory of cognition and action: Some lessons from computational modeling of human multiple-task performance". In D. Gopher & A. Koriat (Eds.), *Attention and Performance*

XVII. 1999.Cognitive regulation of performance: Interation of theory and application (pg. 17 -88). Cambridge, MA: M.I.T. Press. .

20. Kieras, D.E., & Meyer, D.E.. “Predicting performance in dual-task tracking and decision making with EPIC computational models”. Proceedings of the First International Symposium on Command and Control Research and Technology, National Defense University, Washington, D.C., June 19-22. 1995. pg. 314-325.
21. R.G. Babb and A. Mili. Workshop notes, second international workshop on software specification and design. Technical report, Laval University, Quebec City, Canada, March 1984.
22. R.C. Backhouse. *Program Construction and Verification*. Englewood Cliffs, NJ. Prentice Hall, 1986.
23. H.K. Berg et al. *Formal Methods of Program Verification and Specification*. Englewood Cliffs, NJ. Prentice Hall, 1982.
24. A. Mili. *An Introduction to Formal Program Verification*. New York, NY. Van Nostrand Reinhold, 1985.
25. C. A. R. Hoare and J. F. He. The weakest prespecification, part i. *Fundamenta Informaticae*, Vol. 9. 1986. pg. 51-84.
26. C. A. R. Hoare and J. F. He. The weakest prespecification, part ii. *Fundamenta Informaticae*, Vol. 9 1986. pg. 217-252.
27. C. A. R. Hoare and J. F. He. The weakest prespecification, *Information Processing Letters*, Vol. 24. 1987. pg 127-132.
28. Floyd, R. Assigning meaning to programs. *Mathematical Aspects of Computer Science,* Vol. 19. 1967. pg. 19-32.
29. Gries, D. An illustration of current ideas on the derivation of correctness proofs and correct programs. *IEEE Trans. Software Eng.* Vol. 2. 1976. pg. 238 -244.
30. Klaus-Dieter Althoff, Andreas Birk, Susanne Hartkopf, et al. “Managing Software Engineering Experience for Comprehensive Reuse”, *Proceedings of the Eleventh International Conference on Software Engineering and Knowledge Engineering,* Kaiserslautern, Germany June 1999.
31. Musen, M.A. Dimensions of knowledge sharing and reuse. *Computers and Biomedical Research.* Vol. 25. 1992. pg. 435-467.
32. Eriksson H., Shahar Y., Tu S.W., Puerta A.R., and Musen M.A. Task modeling with reusable problem-solving methods. *Artificial Intelligence* Vol. 79. Issue 2. 1995. pg. 293-326.
33. Mili A., Desharnais J., Mili F. *Computer Program Construction*. Oxford University Press. 1994.
34. Gries D. *The Science of Programming.* Springer-Verlag. 1981.
35. Lisse, S. The Phase System: Plan Representation in Soar. Presentation at the 23rd North American Soar Workshop June 25, 2003.
36. Ahmed K. Elmagarmid . *Database Transaction Models for Advanced Applications*. Morgan Kaufmann; 1992.
37. Lehman, Jill Fain, John Laird, Paul Rosenbloom. *A Gentle Introduction to Soar*. 1993.
38. Akyurek, Aladin, and Sayan Bhattacharyya. "Keys and Boxes." 1994. Soar Website. 17 Jul. 2003. <http://www.eecs.umich.edu/~soar/soar73.html>

39. "Demo4." ACT-R Tutorials, ACT-R Website. 17 Jul. 2003. <http://act-r.psy.cmu.edu/tutorials>