Implementing a Cognitive Model in Soar and ACT-R: A Comparison

Tijmen Joppe Muller
tijmen.muller@tno.nl
Department of Training and Instruction
TNO Defence, Security and Safety
P.O. Box 23, 3769 ZG
Soesterberg, The Netherlands

Annerieke Heuvelink
annerieke.heuvelink@tno.nl
Department of Training and Instruction
TNO Defence, Security and Safety
P.O. Box 23, 3769 ZG
Soesterberg, The Netherlands

Fiemke Both
f.both@few.vu.nl
Department of Artificial Intelligence
Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081 HV
Amsterdam, The Netherlands

ABSTRACT

This paper presents an implementation of a cognitive model of a complex real-world task in the cognitive architecture Soar. During the implementation process there were lessons learned on various aspects, such as the retrieval of working memory elements with relative values, alternative approaches to reasoning, and reasoning control. Additionally, the implementation is compared to an earlier implementation of the model in the ACT-R architecture and both implementations are discussed in terms of cognitive theories.

1. INTRODUCTION

People performing tasks in uncertain and dynamic environments require much training in order to gain the necessary expertise. However, the nature of these tasks makes it hard to set up real world training. An appropriate alternative for training decision making in complex environments is scenario-based simulation training [17]. To create a useful training, a simulation needs to represent the aspects of the real world that are vital for achieving the learning objectives. One of these aspects is human interaction; therefore, simulated entities that respond naturally and validly are needed. These entities, known as *agents*, can be used to simulate team members, opponents or bystanders. There is growing conviction and evidence that *cognitive agents* can be developed by capturing human cognitive processes in a cognitive model and implementing it in a cognitive architecture [19, 20, 8].

An architecture poses constraints on the implementation of a model and therefore influences design choices. This paper reports the experiences of implementing the same formal cognitive model in two different cognitive architectures. First, the implementation of the model in the cognitive architecture Soar [13] is presented. This agent performs a real-world task in a complex environment. Implementing the cognitive model provides insights into the use of Soar for agent applications and it may be used to validate the model's behavior in future research. Next, the Soar implementation is compared to an earlier implementation of the same model in the cognitive architecture ACT-R [2]. This allows for the second goal of this paper: the comparison of Soar and ACT-R.

The next section presents the cognitive task and the formal model. Section 3 presents the ACT-R architecture and the implementation, BOA. Section 4 elaborates on the implementation in Soar, which resulted in the agent named Boar. The paper concludes with a comparison of both implementations on various aspects and their connection to the cognitive theories.

Jung, Michel, Ricci & Petta (eds.): *AT2AI-6 Working Notes, From Agent Theory to Agent Implementation, 6th Int. Workshop,* May 13, 2008, AAMAS 2008, Estoril, Portugal, EU.

2. COGNITIVE TASK AND MODEL

The real-world task that has been modeled is the *tactical picture compilation task* (TPCT) from the naval warfare domain. In this task, a navy operator sees a large number of radar *contacts* on his display. Each contact indicates a detected vessel in the vicinity of the own ship. The identities and classifications of these vessels are unknown. The operator can obtain information on these tracks by monitoring the radar screen, such as speed, course, distance to own ship and adherence to shipping lanes. The task of the operator is to use this information to determine both the identity (e.g. hostile, friendly) and the classification (e.g. frigate, fishing boat) of each contact.

A complete cognitive model of the TPCT is constructed using the principles described in this section. The model is based on an extended Belief, Desire and Intention (BDI) framework [7]. BDI facilitates the translation of domain knowledge into a model since domain experts tend to talk about their knowledge in terms similar to beliefs, desires and intentions. The domain knowledge needed to model the TPCT has been elicited from naval experts [6].

In order to develop cognitive agents for training purposes, cognitive behavior that can vary in level of rationality needs to be modeled: agents that can perform a task on different levels of expertise are needed. To make this possible, a belief framework was developed [9]. Three arguments are added to beliefs: a timestamp, the source of the belief and a certainty level. BDI models usually throw away beliefs as soon as a new belief is created that causes an inconsistency. However, to enable reasoning over time, every belief is kept and labeled with the time of creation. The source and cer-

tainty labels make it possible to reason about information from multiple sources and with uncertainty, and reasoning can be done in both a rational and a biased way. A belief `belief(P(A,V),T,S,C)` has a predicate `P` with attribute `A` and value `V`, a timestamp `T`, source `S` and certainty `C`. Below is an example belief – there was an `Identification` of `contact1` as `friendly` with certainty `0.7`, done by reasoning rule `determine-id` on timestamp `12`:

```
belief(Identification(contact1, friendly),
          12, determine-id, 0.7)
```

A cognitive model typically consists of declarative knowledge, denoting facts, as well as procedural knowledge, denoting reasoning rules. Besides modeling how to reason, it is also necessary to model the control on when to reason about what. The next subsection presents the format of the declarative and procedural knowledge and subsection 2.2 explains the control structure of the model.

2.1 Reasoning over Beliefs

The goal in the tactical picture compilation task is to correctly classify and identify all contacts. In order to fulfill this goal, the agent needs information about the contacts' behavior. There are two ways to gather such information. The first is from the external world, e.g. the agent can watch the screen that displays information from sensors such as the radar system. Additionally, the agent can decide to perform actions that lead to more knowledge about the situation, such as activating its radar or sending a helicopter to investigate a contact. The second method to gain information is through the internal process of reasoning about beliefs to deduce new beliefs. In the reasoning process, often multiple beliefs form the evidence for the formation of a new belief. Any uncertainty in the source beliefs will be transferred to the new belief.

An example of this type of deduction is reasoning about formations. If a number of vessels have the same course and are close to one another (source beliefs), they might move in formation (new belief). Moving in formation is an indication that these vessels are frigates. Figure 1 contains an example rule that is part of reasoning about formations. The position of the contact that the agent is currently reasoning about is compared to the position of every other contact that is detected by the radar system. A new belief is created for every pair that indicates how certain the agent is that they are within a distance that can indicate a formation.

The function **Position-Difference** calculates the distance between two positions, **Certainty-Handling-Positions-Difference** calculates the certainty of the distance given the position certainties, **Possible-Distances** returns all possible distances given the calculated distance and certainty, and **Reason-Belief-Parameter** adds the timestamp and stores the belief in long-term memory. In this example, the latest beliefs about the positions are used. In other rules, beliefs are used that ever had a specific value, or those beliefs the agent is most certain about.

2.2 Control of Reasoning

Control is an important aspect of a cognitive agent; it determines when the agent does what. In the TPCT there is one main goal, which is considered the navy operator's desire in the BDI model: to identify all contacts correctly.

Determine-Within-Formation-Distance-Contact(X)
FOR ALL Y
  IF (
    belief(`PositionContact`$(X, P_1), T_1, S, C_1)$
    belief(`PositionContact`$(Y, P_2), R_1, S, C_2)$
    Position-Difference(P_1, P_2, D)
    Certainty-Handling-Positions-Difference
      (C_1, C_2, D, C_3)
    Possible-Distances$(D, C_3, [R])$
    M = *maximum-distance-relevant-for-formation*
    C_4 = (number-of-$[R \leq M]$)/(number-of-$[R]$)
) THEN (
    Reason-Belief-Parameter
      (`WithinFormationDistanceContact`(X, Y),
       `DetermineWithinFormationDistanceContact`$, C_4)$
)

Figure 1: Rule for determining if two contacts are within formation distance

The three subtasks that the agent can perform in order to fulfill this desire are:

1. processing information about contacts on the screen;
2. changing the activity of the radar system; and
3. sending the helicopter on observation missions to gain more information about a specific contact.

The subtasks above are the intentions of the BDI model that the agent can commit to. A valid manner in cognition to determine when which intention becomes a commitment is to have events in the world trigger an intention. For example, when a contact suddenly changes its behavior, the attention of the agent should be drawn to this contact, regardless of the current intention. However, this type of control requires a parallel processing of all events in the world and a parallel checking of relevancy for all subtasks, which is hard to implement. This is why currently a simpler, linear control system is modeled. The agent alternately commits to one of the three intentions to simulate parallel processing. Within the subtasks, the control is also kept simple, e.g. in the first subtask all contacts on the screen are monitored consecutively.

The rule in Fig. 2 illustrates a part of the simple control structure. It determines when the agent starts committing to a new intention. The input parameter I is the current intention, the rule **Start-New-Intention-Selection** determines which intention is selected next depending on beliefs about contacts, and the rule **Select-Next-Contact-To-Monitor** selects the next contact from the list.

3. BOA

In order to execute the model that was presented in the previous section, a cognitive agent needs to be implemented; cognitive architectures are a suitable platform for this purpose. Such an architecture specifies a fixed set of processes, memories and control structures [15] that define the underlying theory about human cognition. The architecture limits implemented cognitive models by this set and consequently

Determine-New-Intention(I)
IF (
  I = Monitor-Contacts
  belief(`NumberOfContactsMonitored`(X), T_1, S, C)
  X = *maximum-number-of-contacts-to-monitor*
) THEN (
  Start-New-Intention-Selection(I)
) ELSE IF (
  I = Monitor-Contacts
  Number-of-Contacts(X)
  X < *maximum-number-of-contacts-to-monitor*
) THEN (
  Select-Next-Contact-To-Monitor()
  Reason-Belief-Parameter
    (`NumberOfContactsMonitored`($X + 1$),
    `DetermineNewIntention`, 1)
)

Figure 2: Rule for determining a new intention

imposes its cognitive theory on these models: it should make correct models easier and incorrect models harder to build. Moreover, the actual behavior of the agent is influenced by the architecture [12].

The presented model has already been implemented in the cognitive architecture ACT-R [4] – this implementation was named BOA. Since this research has been done earlier, several new developments in ACT-R are not taken into account [1]. However, the insights reported here are nevertheless of interest from an agent-application perspective: several of the issues mentioned in this paper have been changed in the latest version of ACT-R. These developments in ACT-R seem to support our experiences that the architecture was too restrictive on some aspects.

3.1 ACT-R

The theory of ACT-R incorporates two types of memory modules: declarative memory and procedural memory. Declarative memory is the part of human memory that can store items; procedural memory is the long-term memory of skills and procedures. ACT-R consists of a central processing system, where *production rules*, representing procedural memory, are stored and executed. The central processing system can communicate with several modules through buffers. One of those modules is the *declarative memory module* where memory items are stored. These memory items, called *chunks*, are of a specific chunk-type, which can be defined by the modeler. In a chunk-type definition, the modeler defines a number of *slot*s that chunks of this type can assign values to. Chunks from the declarative memory module can be placed in the *retrieval buffer* if they match a retrieval request made by a production rule. A retrieval request must contain the requested chunk-type, and may contain one or more slot-value pairs that the chunk must match. The matching chunk is then placed in the retrieval buffer, so it can be read by a production rule. All buffers in ACT-R, including the retrieval buffer, can only store one chunk at a time, even when more chunks match the conditions of the request. If more chunks are available, an activation function defining the accessibility of chunks is used to select a single candidate.

3.2 Implementation Issues

The implementation of the cognitive model in ACT-R resulted in three main observations. The first focuses on the limit of one chunk in the retrieval buffer. The model prescribes access to multiple beliefs in the working memory at the same time in order to reason over them. For example, different positions in time are compared in order to determine a contact's speed. The ACT-R implementation supported this by using the goal buffer for temporary storage and LISP functions to retrieve beliefs.

The second observation was the fact that retrieving a belief with specific features (for example, the belief created last, i.e. with the highest value for the *time* slot) is not guaranteed by using ACT-R's activation function. For example, the agent often uses the latest position of a contact, so he needs the latest belief with predicate `position-contact` for a specific contact. It may however be that an older chunk has been retrieved more often than the latest chunk, resulting in a higher activation score and subsequently the older chunk being retrieved. As a solution, LISP functions were created as substitute to the activation function.

The third issue is about the many calculations the cognitive model requires: these can only be modeled in a low-level manner, making it inefficient to implement them in the architecture. For example, calculating the speed of a contact from its positions over time would require many production rules, while it would not represent the actual cognitive processes of a warfare officer. Here too LISP functions were used for these type of calculations. As a result of this problem and the previous problem, about half of the programming code consists of ACT-R production rules and the other half of supporting LISP functions.

3.3 Control

Control in the context of BDI agents aims at specifying the commitment of the agent at a certain time. The intentions to which the agent can commit and the type of control in the cognitive model were described in section 2.2. The BOA agent implements a simple, linear control system. The agent commits alternately to each intention and within the intention of processing screen information, the contacts are monitored sequentially. This is illustrated by the rule in Fig. 3.

```
(p select-next-contact-goal1
   =goal>
      ISA         commitment1
      goal        monitor-contacts
      state       next-contact
      contact     =contact1
==>
   !bind! =contact2 (determine-next-contact)
   !eval! (determine-rate-other-desires)
   !bind! =eop (= (mod *counter* *rate-other*) 0)
   =goal>
      plan        read-basal-info
      state       start-step
      contact     =contact2
      eop-marker  =eop
)
```

Figure 3: ACT-R code for intention selection

The rule requires the agent to be committed (`commitment1`) to monitoring contacts and be ready to select a new contact to monitor. This new contact is determined by the user-defined LISP function `determine-next-contact`, which loops through the list of contacts. The `*rate-other*` variable defines the number of contacts after which the agent switches to another intention: if this number is reached, the end-of-process marker (`eop`) is set to true. The agent will then consider committing to sending the helicopter, followed by considering to commit to changing the radar. After these considerations and, possibly, reasoning and actions, the agent continues monitoring contacts. Reacting to events in the environment is limited to altering the order of the list of contacts in the 'monitor contacts' intention: if a contact has been identified by the helicopter, that contact is moved to the top of the list to force the agent to monitor it next.

4. BOAR

This section presents the Boar agent, which is the implementation of the model from section 2 in Soar. The next subsection will explain this architecture in more detail and subsection 4.2 describes several implementation issues.

4.1 Soar

Soar, like ACT-R, is a well-known cognitive architecture. Soar defines the world as a large problem space with states and goals. It considers behavior as movement in the problem state by performing actions, either internal (mental activity) or external (observable in the environment). In Soar, this is done by *operators*; in a single cycle, more operators can be proposed, one of these is selected and eventually applied, changing the state of the environment. Goal-directed behavior states that the agent will choose those operators that lead to a goal state. [14]

The memory structure of Soar is somewhat similar to that of ACT-R. It specifies two types of memory: the long-term memory, consisting of procedural, semantic and episodic knowledge, and the *working memory*, corresponding to ACT-R's declarative memory module. The working memory consists entirely of working memory elements (WMEs), which are attribute-value pairs. The attributes of a WME need not be defined beforehand, as is the case with the slots of a chunk. Additionally, the number of WMEs that can be accessed at one moment is not limited – there is no such thing as a retrieval buffer in Soar.

In long-term memory, the procedural knowledge is primarily responsible for the behavior of an implemented model and is defined in terms of *productions*. When conditions apply, a production either proposes the execution of an operator or it executes some reasoning independent from an operator – both may result in changes to the working memory. The difference lies within the persistence of the changes: a WME that was created by an operator will stay in working memory until an explicit change is made. A production without operator reference, also called *elaboration*, creates WMEs that only exist as long as the conditional part of the elaboration matches. The first is said to have operator support or *o-support*, while the latter has instantiation support or *i-support*.

Soar's productions fire in parallel: all productions that have one or more matches for their conditional part in the current state will execute. Consequently, many operators may be proposed at a single moment. Which operator is selected is resolved by means of preferences, which can be added to an operator.

The fact that Soar allows more production rules to fire simultaneously is in contrast to ACT-R's procedure: here, only one production rule can fire at a single moment. If more chunks are available for retrieval by this production, the activation function determines beforehand which chunk is picked.

If a task is too complex to solve by simply adding some beliefs to the working memory, it can be decomposed in subtasks. An example is reasoning about the usage of the helicopter. In order to decide which contact the helicopter is sent to, all contacts are scored. The rule for proposing the operator that scores a single contact is shown in Fig. 4. If this operator is chosen, there is no immediate score available to be added as belief: it needs to be calculated. As a result, there is an impasse and a new substate is created which has the goal to calculate this score. Various operators are available to calculate a part of the score; after each operator calculated its part, the score is available and the attribute `heli-score` will have a value. Consequently, the operator shown in Fig. 4 will be retracted, having achieved its goal.

```
# Propose to score a contact
sp {consider-heli*propose*score-contact
   (state <s> ^name consider-heli
              ^contacts.contact <contact>
              ^top-state.constants <const>)
   (<constants ^max-distance <max>)
              # No score, no visual id
      # and in range of self
   (<contact> -^heli-score
              -^visual-id-belief
               ^distance-to-self <= <max>)
-->
   (<s> ^operator <op> + =)
   (<op> ^name score-contact
         ^current-contact <contact>)
}
```

Figure 4: Soar code for proposing to score a contact

4.2 Implementation

4.2.1 Retrieving Beliefs

In section 2 we argued that the model needs to be able to reason over time. For example, in Fig. 5 the latest position of the own ship (`self`) is retrieved, i.e. the belief with the maximum value for the `time` attribute. This is an example where a belief with a *relative* value is needed for a certain attribute and the absolute value is of no importance. However, it is not easy to match such a belief if no absolute value is available. We tackled this problem by ordering the beliefs with greater-than and smaller-than relations. In order to retrieve the last-but-one belief a new production needs to be added, another for the last-but-two, and so on.

4.2.2 Reasoning Efficiency

Two alternatives arise when generating new knowledge by means of reasoning (i.e. internal action). As explained in subsection 4.1, there are two ways of adding new elements to the working memory: either with o-support or with i-support. The advantages of using i-supported WMEs are:

```
# Calculate distance of contact to self.
sp {boar*elaborate*contacts*distance-to-self
   (state <s> ^name boar
      ^beliefs <beliefs>
      ^contacts.contact <contact>)
   # Retrieve latest position of self
   (<beliefs> ^belief (^predicate position-contact
                       ^attribute self
                       ^time <time>
                       ^value <self-pos>))
  -(<beliefs> ^belief (^predicate position-contact
                       ^attribute self
                       ^time > <time>))
   # Retrieve position of contact
   (<contact> ^position-belief.value <pos>)
-->
   (<contact> ^distance-to-self (float (exec
                      calcPositionDifference
                      <self-pos> |;| <pos>)))
}
```

Figure 5: Soar code for retrieving the distance between contact and own ship

1. they are created automatically if the conditions or the creating production apply in the current state;

2. they are removed if this not the case anymore and thus are not valid in the current state; and

3. they are updated automatically if new information is available.

The advantage of o-supported WMEs is that they are only created when the operators are proposed explicitly, so only at these times some reasoning is done.

If the beliefs that are used for reasoning stay the same for some time, it is more efficient to use elaborations and thus create i-supported WMEs, because if operators are used for this reasoning, they may perform the same reasoning steps multiple times. If beliefs change continuously, the use of elaborations may become computationally expensive, because they perform the reasoning at every change, even if the results are not used. In this case using operators and o-supported WMEs is more efficient. There is no clear procedure for choosing i-support or o-support: one should think about the trade-offs for every situation in order to pick the most efficient option.

To draw the differences between creating i-supported and o-supported WMEs more clearly, an implemented example of both types is given. First consider the production in Fig. 5. It continuously creates an i-supported WME with the distance of contact `<contact>` to the own ship. Every time a new position is observed, either from the own ship (`<self-pos>`) or the contact (`<pos>`), the conditional part of the production for the old WME does not match the current state anymore and the WME is discarded. At the same time, the newly observed information is used to create a new WME for the contact with the `distance-to-self` attribute, and thus the knowledge has been automatically updated. For this reasoning step the i-supported option has been chosen, since the distance is needed continuously for other reasoning. Using an operator would mean that this operator needs to calculate this distance every time the information is needed.

Now consider the production in Fig. 4. This production sets in the creation of knowledge with o-support: it proposes an operator that, when applied, will assign a certain score to a contact. This score is used for considering to send a helicopter to the contact for identification. This scoring is only done incidentally, which makes the use of an operator a better choice. An elaboration would update this score continuously, even while it is not needed most of the time.

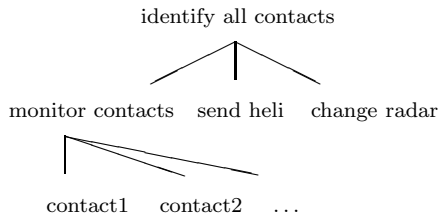

Figure 6: Overview of intentions

4.2.3 Control

The Soar architecture does not provide the means to easily keep a list of contacts, which makes it hard to implement a sequential control for committing to the three intentions described in subsection 2.2. Alternatively, it easily allows the creation of subgoals, as explained in subsection 4.1. By defining the monitoring of every contact as a subgoal of the 'monitoring contacts' intention, the structure of goals and subgoals becomes as shown in Fig. 6.

The commitment to one of the three intentions is decided as follows: if an event triggers the intention to send a helicopter or change radar activity, the agent commits to this intention. Otherwise it will first monitor all contacts, then consider sending the helicopter and finally consider changing radar activity. The linearity of this cycle is forced by explicitly remembering the control status in WMEs. Figure 7 shows the commitment to monitoring contacts: when a new cycle starts, the time is saved in the `start-process-passive` attribute. Then every contact not checked after this time (`-^checked > <starttime>`) is monitored and gets tagged with a new time, until all contacts have been checked this way. After completing the helicopter and radar intentions, a new cycle is started. The order of checking contacts is random and may be different each cycle.

```
sp {boar*propose*process-passive-information
   (state <s> ^name boar
              ^start-process-passive <starttime>
              ^contacts.contact <contact>)
  (<contact> -^checked > <starttime>
              ^id <contact-id>)
-->
   (<s> ^operator <op> + =)
   (<op> ^name process-passive-information
         ^current-contact <contact>)
}
```

Figure 7: Soar code for proposing to monitor a contact

An example of an event triggering the 'send heli' intention

is shown in Fig. 8. It simply states that if the helicopter is airborne and does not have a mission, for example just after identifying a specific contact, the agent should commit to reasoning about what it should do. This commitment can break into the aforementioned cycle at any time the conditions apply.

```
# A heli is considered when it has
# no mission and is airborne.
sp {boar*propose*consider-heli*airborne
   (state <s> ^name boar
              ^helis.heli <heli>)
   (<heli> ^id <heli-id>
           # Heli has no mission and is airborne
           ^mission free
           ^status airborne)
-->
   (<s> ^operator <op> + =)
   (<op> ^name consider-heli
         ^heli <heli>)
}
```

Figure 8: Soar code for the event-driven selection of the 'send heli' intention

4.2.4 External Functions

The simulation environment for performing the tactical picture compilation task is created in Game Maker [18]. It simulates a radar screen with information about the contacts in the surroundings. The simulation environment reacts on certain actions, e.g. clicking on a contact will provide detailed information about it.

For letting Soar communicate with this Game Maker environment an interface is needed, which is implemented in Java. The actions of the agent are written to a text file by the Java interface, read by Game Maker and consequently performed in the environment. An example of an agent action is the request for detailed information, which simulates a mouse click by a human. Any input from the environment, such as a new contact or the position of the own ship, is written to a text file by Game Maker, read by the Java interface and presented as input for the agent. This form of communication slows the execution of the agent down, since it continuously waits for reactions from the environment.

To perform complex calculations, user-defined functions in Java are needed, similar to one of the issues when implementing the formal model in ACT-R (see subsection 3.2). These functions are called from inside the agent, but can only be used in the actions of a production. Consequently, if a calculation needs to be performed as part of the condition of a production, it has to be executed by another production and the result needs to be made available through the working memory.

5. CONCLUSION AND DISCUSSION

This paper presents an agent built in the cognitive architecture Soar, capable of performing a complex real-world task. The implementation is based on a formal model of the task and has previously been implemented in ACT-R. The remainder of this paper will present the lessons learnt on several aspects of agent practice and links them to cognitive theories.

Two implementations of a single cognitive model give only one point of view: a different model may have different demands, especially when a different framework is used. Additionally, the implementations have not been validated – further work in this direction may consist of experiments with subject-matter experts comparing the performance of BOA, Boar and humans performing the tactical picture compilation task.

Nevertheless, this paper shows that one should consider the functionalities requested by the model and the possibilities an architecture offers to implement those demands.

Working Memory Access.

Most of the cognitive theories about human working memory agree on a storage capacity of multiple but limited amount of items [16, 3, 11, 5]. This assumption was used in designing the cognitive model: several rules in the cognitive model need multiple beliefs at the same time for reasoning (an example of such a rule is in Fig. 1).

The ACT-R theory used for implementing BOA proved to be too restrictive: access to only one chunk at a time is allowed. In order to access more beliefs, a work-around was used in the ACT-R implementation. On the other hand, Soar does not limit the number of accessible working memory elements, so this did not pose any problems implementing Boar. Different approaches to the working memory theory result in different types of behavior: if only a limited number of elements is accessible, reasoning will be restricted to these elements, which can cause a different way of acting than when all elements are available.

Retrieving Beliefs.

In order to reason over time the retrieval of specific beliefs with a relative value is needed, such as the 'last' or 'one-but-last' belief of some kind – a capability humans apply unconsciously. Unfortunately, this type of retrieval operator is not yet available in either architecture.

In ACT-R the working memory items are retrieved by means of an activation function. However, this function does not guarantee the retrieval of a memory item based on such a relative value. The solution was to create LISP functions for retrieving beliefs. Soar allows ordering the beliefs in the conditional statement of a production rule, making it possible to retrieve beliefs with a relative value. However, operators need to be created for each relative value, making the translation from model to Boar somewhat inefficient.

Control.

A linear control was modeled in favour of event-driven parallel control. This choice was made in order to simplify the process of committing to an intention, even though human decision-making will be more reactive to cues from the environment. ACT-R's sequential execution of production rules fits this simplified model, but as a result the BOA agent reacts slowly on important changes in the environment, because the agent's behavior cannot be interrupted by these external events [10]. In Soar the sequential execution of plans is forced by letting production rules fire in an explicit order (as shown earlier in Fig. 4), but this architecture more easily allows event-driven control.

Calculations.

The tactical picture compilation task contains many sit-

uations in which the human expert makes estimations, for example on how close a ship is to the the own ship or whether ships are moving in formation. Currently there is no method available to model the process of these estimations. Instead, the estimations are replaced by exact calculations and made into an 'estimation' by adding the notion of uncertainty to the resulting belief. Modeling these calculations as cognitive tasks in an architecture would require an infeasible amount of productions, without actually copying human behavior. Therefore, the execution of complex calculations is done externally by LISP functions or Java methods.

Speed.

Humans are able to use multiple beliefs that were gathered over time for reasoning. This is represented in the belief framework by adding a time tag to every belief and storing all beliefs in memory. As a result, the agent can access multiple beliefs over time for reasoning. For example, it can access several beliefs about the position of a contact to reason about the contact's speed and movement behavior.

Unfortunately, this creates a practical problem: there is an exponentially growing amount of beliefs, which means no system will eventually be able to cope with the resulting CPU-expensive searches for the necessary beliefs during real-time simulation. It is necessary to deal with this more efficiently. Even though certain facts in the past need to be remembered by the agent, it is not necessary to remember every specific detail, which is the case in this implementation. Humans do not remember every detail exactly, but compress their memories by conceptualizing or clustering them. Future agents that incorporate the belief framework used in this research will need some form of compression or smart discarding of beliefs to copy this behavior. We are currently developing a method to cluster and abstract beliefs over time, sources and certainties, in order to form a more realistic model of episodic memory.

Clearly, this problem has its effect on the implementations. The ACT-R agent becomes slow over time, even though some functionality to remove unimportant beliefs had been implemented. This slowness makes the observed behavior of the agent not very human-like, especially in reacting to changes in the environment [10]. On the other hand, Boar has been used in a demonstration of about twenty minutes, in which the agent showed no reduction in speed. To draw general conclusions about the performance of both architectures, further research is needed.

6. ACKNOWLEDGEMENTS

This research has been supported by the research program "Cognitive Modelling" (V524), funded by the Netherlands Defence Organisation.

7. REFERENCES

[1] J. R. Anderson. *How Can the Human Mind Occur in the Physical Universe?* Oxford University Press, 2007.

[2] J. R. Anderson and C. Lebiere. *The Atomic Components of Thought*. Lawrence Erlbaum Associates, 1998.

[3] A. D. Baddeley and G. J. Hitch. Working memory. *Recent Advances in Learning and Motivation*, 8:647–667, 1974.

[4] F. Both and A. Heuvelink. From a formal cognitive task model to an implemented ACT-R model. In *Proceedings of the 8th International Conference on Cognitive Modeling*, 2007.

[5] N. Cowan. *Working memory capacity*. Psychology Press, New York, NY, 2005.

[6] B. J. v. Dam and H. F. R. Arciszewski. Studie commandovoering do-2: Beeldvorming. Technical Report FEL-02-A242, TNO-FEL, 2002.

[7] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 677–682, Menlo Park, California, 1987.

[8] K. A. Gluck and R. W. Pew, editors. *Modeling Human Behavior With Integrated Cognitive Architectures: Comparison, Evaluation, and Validation*. Lawrence Erlbaum Associates Inc, 2005.

[9] A. Heuvelink. Modeling cognition as querying a database of labeled beliefs. In *Proceedings of the 7th International Conference on Cognitive Modeling*, 2006.

[10] A. Heuvelink and F. Both. BOA: A cognitive tactical picture compilation agent. In *Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2007*, Silicon Valley, California, Nov. 2007. IEEE Computer Society Press.

[11] C. Hulme, S. Roodenrys, G. Brown, and R. Mercer. The role of long-term memory mechanisms in memory span. *British Journal of Psychology*, 86:527–536, 1995.

[12] R. M. Jones, C. Lebiere, and J. A. Crossman. Comparing modeling idioms in ACT-R and Soar. In *Proceedings of the 8th International Conference on Cognitive Modeling*, 2007.

[13] J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[14] J. F. Lehman, J. Laird, and P. Rosenbloom. A gentle introduction to Soar, an architecture for human cognition: 2006 update, 2006.

[15] R. L. Lewis. Cognitive theory, Soar, Oct. 1999.

[16] G. A. Miller. The magical number seven, plus or minus two: Some limtis on our capacity for processing information. *Psycological Review*, 63:81–97, 1956.

[17] R. L. Oser. A structured approach for scenario-based training. In *Proceedings of the 43rd Annual Meeting of HFES*, volume 43, pages 1138–1142, Oct. 1999.

[18] M. Overmars. Game maker: http://www.yoyogames.com/gamemaker/.

[19] R. W. Pew and A. S. Mavor, editors. *Modeling Human and Organizational Behavior*. National Acadamy Press, 1998.

[20] F. E. Ritter, N. R. Shadbolt, D. Elliman, R. M. Young, F. Gobet, and G. D. Baxter. Techniques for modeling human and organizational behaviour in synthetic environments: A supplementary review. Technical report, Human Systems Information Analysis Center, June 2003.