Cognitive Modelling tools and their uses in game development

Kevin Wyckmans

Groenenborgerlaan 149, 2020 Antwerp, Belgium

Abstract

Rule based modelling can be used to describe and simulate the behaviour of humans. This is often done by using cognitive modelling tools. These are used to simulate the working of the brain and the decision making process of a human being. In this reading study, the applications of such tools in a game development context are explored. More specifically the possibility of using these tools and techniques to make the development of complex behaviour in video games easier. Another possibility is to use these tools to test the playability of a game, by running simulations and emulating human behaviour.

Keywords: modelling, ai, game development, cognitive artificial intelligence

Email address: `kevin.wyckmans@student.ua.ac.be` ()
URL: `msdl.cs.mcgill.ca/people/kevin` ()

Contents

1. Introduction

The current techniques to develop artificial intelligence to control non-player characters (NPCs) have proven to be very powerful. A lot of algorithms and efficient techniques exist to solve common problems such as decision making and path finding, but these require very specific knowledge. Knowledge that the people designing the npcs may not posses, they might have to resort to programmers to convert their ideas into working characters. This adds a layer of complexity that takes time, is expensive and may produce unwanted behaviour.

Another issue is that game designers may want to test the playability of their game. [Ambinder] show the importance of playtesting, and how far you can take this.. You want to be really sure that your game is fun, that it is not too easy, or too hard. Often this is don by letting humans play the game over and over again, which is a time consuming process that is very expensive. It can be more efficient to develop an npc that behaves like a human and let the game run many times automatically. The problem is that making an npc behave like a human is not an easy feat.

To try and solve both these problems, I will look into rule-based generation of behaviour for NPCs. In this case, a game developer just needs to define a set of rules that describe the behaviour of a character. Interaction of these rules should, ideally, show emergent behaviour. This may be easier to learn for a non-programmer. These rules can also be used to describe human behaviour. To test this, I will investigate two cognitive modelling tools, Act-R and SOAR. Both have already been used in a game development context to control an npc and to test playability.

In section one, a small introduction into said tools will be given. Section two will provide a small practical example in which both tools will be used to model the game pac-man. A comparison will be made, while in section three, the possibility of using these in a commercial game engine environment will be discussed. In section four we make a comparison with different existing techniques. Finally, before a conclusion is made, the different possibilities concerning future work are discussed.

2. Cognitive Modelling Tools

A single system (mind) produces all aspects of behavior. It is one mind that minds them all. Even if the mind has parts, modules, components, or whatever, they all mesh together to produce behaviour. - Newell

In [Feist and Rosenberg, 2009] cognitive psychology is described as a sub-discipline of psychology exploring mental processes. It is the study of how people perceive, remember, think, speak and solve problems. Historically, cognitive psychologists used to study each of these problems separately. We gained a lot of very in-depth knowledge about these processes, but we lacked a way of connecting everything. A human being can and will use different systems to achieve certain goals, using only one subsystem in isolation is not realistic.

Newell identified this problem and suggested several possible solutions [Newell, 1973]. This resulted in [Newell, 1990] in which a unified theory of cognition is described and SOAR is presented as an example of an actual implementation of such an architecture. This allows us to create models that simulate the behaviour of humans. This is exactly what we want when modelling the behaviour of a computer controlled character that has to behave in a realistic way. In the following years alternatives to SOAR have been proposed, for example ACT-R. Both architectures will be discussed in the following section. This provides the reader with a basic overview and understanding of how ACT-R and SOAR work. With the information provided, the reader will have a broad understanding of the internal processes of these tools and will be able to start creating small simulations.

2.1. Adaptive Control of Thought-Rational (ACT-R)

Historically, the ACT theory started with [Anderson, 1976] in which a general theory of cognition is first proposed by Anderson. Over the course of a couple decades this evolved into ACT-R, a theory and accompanying architecture for cognition [Anderson, 1993] that allowed researchers to program simulations and models of cognition. After a few years of development and multiple workshops it was further refined in [Anderson and Lebiere, 1998].

Basically ACT-R is a modular framework that consists of interacting modules, each representing a specific function of cognition with in the center a production system. First we will discuss the general structure of ACT-R, afterwards the most important modules will be discussed in greater detail. Finally some examples will

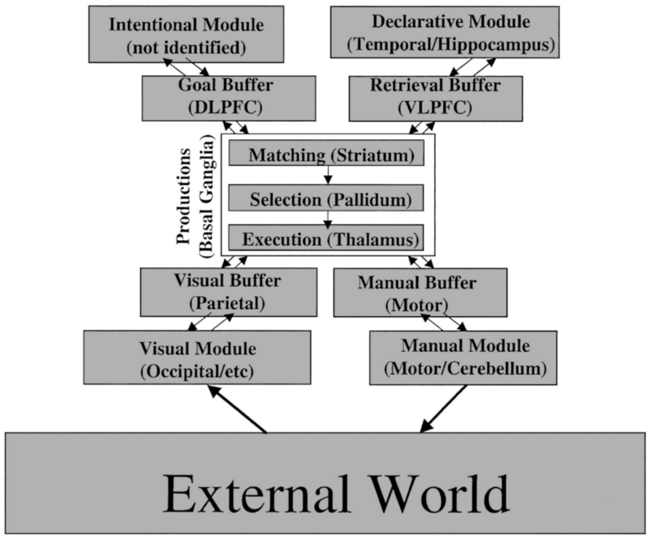


Figure 1: Internal structure of ACT-R - [Anderson et al., 2004]

be presented using pyACT-R, a python implementation of ACT-R. Most of the information in this section is based on [Anderson et al., 2004], which is the most up-to-date description of ACT-R.

Overview of ACT-R

Fig. 1 shows us the basic structure of ACT-R. Every gray block is a different module, dedicated to processing a different kind of information. Mentioned between brackets are the actual areas in the brain that we suspect are responsible for each function. As you can see, all these modules are coordinated by a central production system. The system is not limited to the modules shown in Fig 1, extensions and new modules can be connected to the existing system if needed.

An important thing to note is that each module more or less works independent from other modules. This goes for the production system as well, what happens in the modules is completely encapsulated. Communication is provided by buffers. Each module has a dedicated buffer available, in which a limited amount of information can be stored. In turn, the production system can recognize patterns and change the information present in these buffers to perform actions.

These buffers are a critical part of ACT-R. First, you have the goal-buffer, which keeps track of the internal state when solving a problem. Then there is the retrieval buffer that represents information from the long-term declarative (memories which can be consciously recalled such as facts and knowledge) memory. The manual buffer is responsible for hand-movements. On top of this there is the visual buffer, which actually contains two buffers: one for identifying location (where?) and one for keeping track of identity (who?). For completeness we mention that there are also rudimentary vocal and aural systems, but these will not be further discussed. The contents of all buffers are determined by the complex processes in their respective modules.

In turn the buffers interact using the central production system to determine cognition. The production system represents the procedural memory (memory for the performance of particular types of action). It consists of different production rules. An important feature of these rules is update the buffers in the system.

This leads us to the main process in ACT-R: the buffers hold information from the real world or internal modules, the production system recognizes patterns, fires a production and updates the buffers for the next cycle. One such cycle is assumed to take about 50ms to complete. This is the same amount of time as SOAR uses [Newell, 1990]. Each production rule specifies a pattern to which it will match, and an action that specifies the changes it will make to the buffers. It is important to note that a lot of systems can work in parallel, a lot of modules can work asynchronously and in parallel. For example: processing what we can see and retrieving memories can happen at the same time.

Two important limitations do need to be mentioned: first, the contents of a buffer is limited to a single unit of knowledge, called a chunk. Thus only one memory can be retrieved at any given time. Furthermore, only one production can fire at a time, if multiple matches occur, a conflict resolution system is in place.

The following sections discuss the most important parts of ACT-R in greater detail.

Perceptual Motor System The Perceptual Motor System is a computational model that approximates the basic timing of perceptual output and the input to the motor systems. The motor system is quite simple, you just have to find an abstract way to represent objects to interact with, for example a computer program, you represent this with chunks that the modules can process and eventually put into a buffer. The perceptual system has the same basic premise but is a bit more complex.

As stated before, we have to divide the visual system into two subsystems: visual-location (where) and visual-object (what). When the production system requests info from the where system, it provides some constraints based on the visual properties of an object (such as colour: green) or the spatial location (horizontal: bottom). the where system will then provide one result that adheres to the constraints. If there is more than one result, one is randomly selected.

On the other hand, the what system requires a visual location to be provided, which will cause the system to shift its focus to that location, and generate a declarative memory chunk containing the object description. The system provides two options: one where all attention shifts take the same amount of time (185ms) and one where eye-movement is taken into account and time increases in relation to distance and location.

The Goal Module The goal module represents the ability to sustain cognition while trying to achieve a goal without any change in external output. It does this by keeping track of what steps need to be taken to achieve the end goal and to make sure that the behaviour presented by a model gets closer to achieving this final state. Simply put: you provide a final condition that has to be met and any eventual sub-conditions, this module will make sure productions are fired that bring you closer to meeting that condition. More information can be found in 2.1-Procedural Memory.

The Declarative Memory Module Declarative memory represents our long-term personal and cultural memory. It is our actual knowledge, the facts we know. Accessing this memory and searching for specific information is not instant, this takes a certain amount of time. The activation process controls this access. In ACT-R this module consists of a collection of chunks, the activation of such a chunk *i* is given by:

$$A_i = B_i + \sum_j W_j S_{ji}$$

where B_i is the base-level activation of the chunk i, Wjs reflect the attentional weighting of the chunks that are part of the current goal and the S_{ji}s are the strengths of association from the elements j to chunk i.

More specifically: $W_j = 1/n$ with n is the number of sources of activation and $S_{ji} = S - \ln(fan_j)$ with S most often chosen as 2.

This leaves the Base level activation:

$$B_i = \ln(\sum_{j=1}^{n} t_j^{-d})$$

with t_j the time since the j-th practice of an item. For d a value of .5 is most often used.

Concerning the actual retrieval of a chunk, a chunk will only be retrieved once their activation is over a certain threshold. The probability that the activation will be greater than a threshold τ is:

$$P_i = \frac{1}{1 + e^{-(A_i - \tau)/S}}$$

In this case, s controls the noise in the activation levels and is typically set at about .4.

Once a chunk is retrieved there is some latency involved to model the retrieval of information. The time needed to retrieve a chunk is:

$$T_i = Fe^{-A_i}$$

As can been seen, there are a few parameters that can be tweaked: the noise parameter s, retrieval treshold τ and the latency factor F. The values of these parameters can be adjusted according to the experiment performed, but default values are provided that should produce acceptable results in most cases. On top of this, there is a connection between these values given by:

$$F \approx 0.35e^{\tau}$$

This results in a retrieval time of 0.35s if $A_i = \tau$.

All these formulas might seem complex, but once implemented they become clearer, it is suggested to play with the parameters in a simple use case to see what effect this causes on the simulation.

Procedural Memory Procedural memory or the production system is the module that combines all other modules by recognising patterns in buffers, processing information and changing chunks in buffers. At any point in time, only one production rule can be fired, even if multiple rules match an existing condition. The one with the highest utility value is chosen. We define the utility of a production i as

$$U_i = P_i G - C_i$$

where P_i is an estimate of the probability that if production i is chosen, the current goal will be achieved, G is the value of that current goal and C_i is an estimate of the cost to achieve that goal (often measured in time). P_i and C_i are learned from experience with that production rule.

If n productions match, the chance of selecting the i-th production is:

$$P_i = \frac{e^{U_i/t}}{\sum_j^n e^{U_j/t}}$$

with the summation is over all applicable productions and t is a parameter to control noise, often set at .5.

The values of C_i and P_i are adjusted over time using a Bayesian framework. The estimated value of P is:

$$P = \frac{\text{Successes}}{\text{Successes + Failures}}$$

To prevent P from taking extreme values in the beginning of the simulation, we define P to have a previous value of θ. We can achieve this by setting successes to $\theta V + m$ and failures to $(1 - \theta)V + n$, with m the number of experienced successes and n the number of experienced failures, and V is the strength of prior θ. These values can be used in the initial start of a simulation, when no prior experience is present. The process for C is identical.

This concludes the high-level overview of the core modules used in ACT-R. All basic formulas are present and with this information, you will be able to construct basic models. The parameters found in the last two modules can be quite confusing, but when starting construction on a model, these can all be kept default. It is only when improving the model that it is important to know what parameters you can adjust and what effect these have on the bigger picture. Some extended examples can be found in [Anderson et al., 2004], although these are still quite abstract. We will try to clarify everything using concrete examples using pyACT-R.

2.1.1. PyACT-R

PyACT-R is a rewrite of ACT-R in python described in [Stewart, 2006]. This was done to gain a better understanding about the inner workings of ACT-R. Each part of ACT-R was separated in an individual component that could be described by a simple algorithm. Each component was implemented individually, combined they give us the complete ACT-R system. Every part is thus clearly defined and separated, which allows modellers to turn parts on and off depending on the requirements. It also allows for easy customizations and adaptations.

In this part we will describe how this implementation works. There will be some overlap with the previous part, but in this case the focus will be on the specific syntax and semantics, so that the reader knows how to actually use this system in practice. Basically, all major modules described in the previous section are present in pyACT-R, the most important ones will be discussed and small examples will be given. For more complete examples, you can check: `https://sites.google.com/site/pythonactr/home`. All examples given are taken from this website.

Important to know beforehand is that a difference is made between actual time needed to compute something and the simulated time. If it is mentioned that a certain action is instant, this simply means that computational time is ignored in the simulation, the system just makes sure nothing else happens during the computation and only continues once it is completed. This makes sure that in the actual simulation this appears instant.

Communication Between Modules As we already know, ACT-R consists of different modules, each performing a specific function. These modules exchange information in *chunks*. Each chunk can consist of a number of *slots*. 7 ± 2 is the recommended amount [Anderson and Lebiere, 1998], but you can choose your own amount if you want to. In python a chunk looks like:

```
chunk ='dog large friendly'
```

Another important part we mentioned were buffers. Each buffer can hold one chunk at a time. Modules can:

1. Place chunk in buffer
2. Modify the value of slots
3. Clear buffer
4. Retrieve a copy of the chunk in the buffer

It is important to note that the buffer always contains a copy of a chunk, thus changes in the buffer will not cause the original chunk still in the module to be changed. All buffers taken together represent more or less the current state of the entire system. Creating a buffer and putting a chunk in it is just as simple as:

```
focus=Buffer()
focus.set('sandwich bread')
```

Specific modules will have a buffer attached and fill it, the production mechanism will provide automatic matching to the contents, but more on that later.

Chunk Matching If we want to create a model, we need to be able to match chunks. This is used in two important areas:

1. To examine the current context to determine what action to take next (cfr. Production System).
2. To find a particular chunk in a larger collection (cfr. Declarative Memory).

In both cases, the matching process involves checking if a chunk matches a *pattern*. A pattern matches if each of the slots in the pattern has a rule that matches with the contents of the corresponding slot in the chunk. There are three possible rules:

1. Exact Match: each slot in the pattern corresponds exactly to each slot in the chunk.
2. Not Match: the pattern indicates that a slot can not have a certain value. A '!' is used to indicate a not match. For example: `'dog !small friendly'` matches with `'dog large friendly'` but not with `'dog small friendly'` or `'dog large vicious'`.
3. No Chunk: this handles the special case of an empty buffer.

Another important part of matching is a *variable*. This is a label that can take different values in the slot of a pattern. We can indicate variables with '?'. The first time a variable is used in a match, its value is bound to the value in the corresponding slot: `'dog large friendly'` would match `'dog ?size friendly'` and size would be bound to large. Once bound, the variable keeps the sama value within the same context, so it would no longer match to `'dog small friendly'`, because size has been bound to large.

Matching patterns must match specified buffers, this can be more than one buffer. In this case, the contents of each buffer must match and the variable binding syste:

```
buf1='dog ?size friendly', buf2='cat ?size sleepy'
```

would match to either of the following:

```
buf1='dog large friendly', buf2='cat large sleepy'
buf1='dog small friendly', buf2='cat small sleepy'
```

but would not match

```
buf1='dog large friendly', buf2='cat small sleepy'
```

because size would be bound to large and thus not match small. Variable and matching rules can be combined in any way, including multiple rules for the same slot. For example you can use multiple 'not' matches to exclude cats and dogs.

An optional partial matching system is present, in which case patterns do not have to match completely. You can indicate for each slot that they do not have to match exactly. This is currently only of use in the declarative memory system so you can recall chunks that are not exactly alike, but do resemble each other. This system is governed by the formula:

$$\sum_k P_k M_k$$

Each slot has a value P that indicates how important that slot is, this can be set by the modeller. The value M is 1 if slot k does not match and 0 if it does. We make the sum for all k slots in a chunk. The resulting value is subtracted from the chunks activation value (how likely it is to be recalled from memory). If needed M can be configured as well, for example if you request something `red` from the memory, you can state that the value `pink` has an M value of 0.5 and thus only receives half the penalty for that slot.

```
partial=Partial(DM, strength=1.0, limit=-1.0)         #
    turn on partial matching for Declarative Memory -
    strength is P, limit is how similar slots can be.
partial.similarity('customer1','customer2',-0.1)    # set
     the similarity between customer1 and customer2 -
    they are very similar, this is M
partial.similarity('customer1','customer3',-0.9)    # set
     the similarity between customer1 and customer3 - not
     so similar, again M
```

Modules As we mentioned in the previous part, each module represents a part of the brain. We can see the modules as functional systems, whereas the buffers are the communicative systems. When required, a module is activated instantaneously. Some actual processing time may be required, but in the simulation this is neglected and regarded as instant. Once activated, a model calculates the time needed to perform his task, after which they wait for this specific amount of time and then the task is carried immediately (once again, no time passes of simulation purposes, regardless of computational time). It is important to note that all modules and buffers operate in parallel.

The production system is the core that drives on the simulation, searching for matches and based on those fire rules that cause actions. Each action then causes changes in the state which produces new matches and so forth. It contains a set of *if-then* rules for performing tasks. This is what enables us to simulate cognition, perception and motor actions. It has to determine which of these rules to fire at any given moment. Firing a rule takes *50ms* and only one production can fire at a time. It might also be possible that there is no match, in this case the system does nothing. The if-part or Left-Hand-Side is a collection of patterns. Whenever these patterns match (on one of a subset or one of all buffers), a production can fire.

It is also possible to match the state of a module. Each module can have a separate buffer specific for that model, describing its state, for example whether it is performing an action or not. The then part, or Right-Hand-Side contains a series of actions to be taken when the production fires. These actions are commands for other modules or buffers. It is important to check ACT-R theory for specifics on these actions, because certain restrictions apply depending on what you want to simulate. One example is that only one request to declarative memory can be made in the action part. Any bound variable from the if-part can be used in the then part, but the bound value is discarded after the rule has fired.

Regardless of actual computational time, the production system immediately finds all matches after a change in a buffer occurs. Once a match is found, the production system will wait for 50ms, after which the production is fired and all actions are carried out. Depending on the actions and the modules required, this might take some simulation time. If during the 50ms waiting period a change in buffers is detected, the firing will be aborted and the system will search for a new match.

A basic production looks like:

```
class SimpleModel(ACTR): # everything in here defines
    the model
  goal=Buffer()  # create the goal buffer

  # now define the productions
  def first_production(goal='starting'): # IF the goal
      buffer matches this
    goal.set('ending')                   # THEN do this
```

Production Conflict Resolution As stated in ACT-R theory, multiple production rules can match at the same time, so conflict resolution must be performed. In the previous part this was quite a complicated piece of theory, but, fortunately, in practice this is a bit simpler. We want to fire the production with the highest utility, we recall the following formula:

$$U_i = P_i G_i - C_i + \epsilon$$

with P the probability that the production will lead to success, G the value of that outcome. C is the time between performing the rule and achieving the outcome. ϵ is a user defined noise factor, usually set to 0.

These values can be set manually, but recommended is that the system learns these by itself. This can be done by marking productions as successes and failures. This is as easy as calling `self.success()`, `self.fail()` or `self.reward(0.8)` (with positive numbers being good, and negative numbers being bad) in the body of a production. The system keeps a tally of the successes (s) and failures (f) and makes the following estimates:

$$P = \frac{s}{s+f}$$

$$C = \frac{t}{s+f}$$

t is known exactly because we are working with a simulation and so we know how long each action will take.

Declarative Memory Declarative memory is a system for storing and retrieving chunks. The less often a chunk is used, the harder it is to retrieve (less likely to be given as a result when possible matches occur). If a chunk is added, it is inserted immediately. If the chunk is already present in memory, it is not added again, it is treated as the same chunk, but its activation will go up. Adding chunks to this memory is the responsibility of the modeller. It might be needed to insert memory in the beginning, representing some facts that an entity already knows for example. Below you can find an example of how to initialize the declarative memory and all the parameters for configuring it, more info in the comments between the code.

```
class MyAgent(ACTR):
  focus=Buffer() #The goal buffer
  DMbuffer=Buffer() #Declarative Memory buffer

  #Initializing the Declarative Memory
  DM=Memory(DMbuffer,latency=0.05,threshold=-25,
      maximum_time=20,finst_size=10,finst_time=30.0)
  # latency: parameter F in activation formula
  # threshold: chunks must have activation greater than or
       equal to this to be recalled (can be set to None)
  # maximum time - no retrieval will take longer than
      this amount of time
  # finst_size - how many chunks can be kept track of
  # finst_time - how long a chunk can be kept track of

  #This adds noise to the DM parameters, so that some
      random changes can occur
  #In realistic scenarios not all params remain constant
  dm_n=DMNoise(DM, noise=0.0,baseNoise=0.0)

  #learning system, when a chunk is used, it's activation
      is increased. Conversely
  # it gradually decreases if it is not used at all. Needs
       to be turned on
  # to see chunks compete with one another.
  # Decay: parameter d in activation formula, almost
      always 0.5
  dm_bl=DMBaseLevel(DM, decay=0.5,limit=None)

  #Turns on spreading activation. This means that DM
      chunks that have slot contents
  #that match what's in the buffers will receive a boost
      in activation.
  dm_spread=DMSpreading(DM, focus)
  dm_spread.strength=2
  dm_spread.weight[focus]=.5

  partial=Partial(DM, strength=1.0,limit=-1.0)
```

```
partial.similarity('customer1','customer2',-0.1)
partial.similarity('customer1','customer3',-0.9)

# note that this model uses slot names - slotname:
    slotcontent
def init():
    #Add some initial facts to the memory.
    DM.add('isa:order customer:customer1 type:ham_cheese
         condiment:mustard')          # customer1's order
    DM.add('isa:order customer:customer2 type:ham_cheese
         condiment:ketchup')          # customer2's order
    DM.add('isa:order customer:customer3 type:ham_cheese
         condiment:mayonnaise')       # customer3's order
    DM.add('isa:order customer:customer4 type:ham_cheese
         condiment:hot_sauce')        # customer4's order
    focus.set('isa:ingrediant type:bread')
```

The `DMBaseLevel` function represents the basic learning function described in the previous section about ACT-R theory. As you can see there are a lot of parameters, but most of the time, the default values are good for general models.

Using pyACT-R

This concludes the description of the pyACT-R implementation of ACT-R. It is recommended to visit `https://sites.google.com/site/pythonactr/home` and run all examples at least once. This will greatly increase understanding about how the system works. At the end of this paper, the code for the simulation has been included that uses most of the systems described, this might also help if you want to create your own models.

2.2. SOAR

An alternative to ACT-R is SOAR. SOAR was actually the first computer-based architecture proposed in [Laird and Newell, 1987]. The most up to date and most complete source about SOAR is The Soar User's Manual `http://web.eecs.umich.edu/~soar/downloads/Documentation/SoarManual.pdf`. This document will be the primary reference for the following section and is the go-to reference if further information is required. In conjunction with this, you may need the SML Quickstart Guide, which describes how to interface SOAR with external environments (that can be written in different programming languages).

Combining these two should be enough to create basic models and interface the model with an external environment.

Some of the more recent evolutions of SOAR are explained in-depth in [Laird, 2012]. On top of this, there is a FAQ and a basic Tutorial (which uses an older version of SOAR, but still gets the point across quite nicely).

Once again a small overview will be given of the basic structure of SOAR and how to use it. There may be some overlap with parts from ACT-R, which is logical, since they are based on the same theories. Theory already explained in ACT-R will not be repeated, but it will always be clearly mentioned. An example will be provided to illustrate the basic concepts used in SOAR. For accompanying source code, please check the SOAR Users Manual for a definite specification. It also contains a completely implemented version of the example used in Appendix A.

2.2.1. Overview of SOAR

SOAR assumes that all *goal* oriented behaviour can be simulated by the selection and application of *operators* to *state*, where the goal is the desired outcome, a state is the representation of our current situation and an operator is something that changes this representation. It continuously tries to apply the current operator and select the next one until the goal is achieved. All currently available information in SOAR, from sensors, intermediate inferences, goals, active operators and so on, is called the *working memory*. This working memory is organized as *objects*, which are described by their *attributes*, which can be objects as well. This can be seen as the sort-term knowledge. Long-term knowledge on the other hand, specifies how to respond to different situations in the working memory. This knowledge has to be added as the modeller and can be seen as our actual SOAR program. These are the productions that describe what operators to use depending on the state and what actions to perform cfr. the production system in ACT-R. Four types of knowledge exist:

1. Operator Proposal
2. Operator Comparison
3. Operator Selection
4. Operator Application

All of these, except operator selection are similar to the if-then statements described in the previous part. If the *conditions* are met, the production will *fire* and

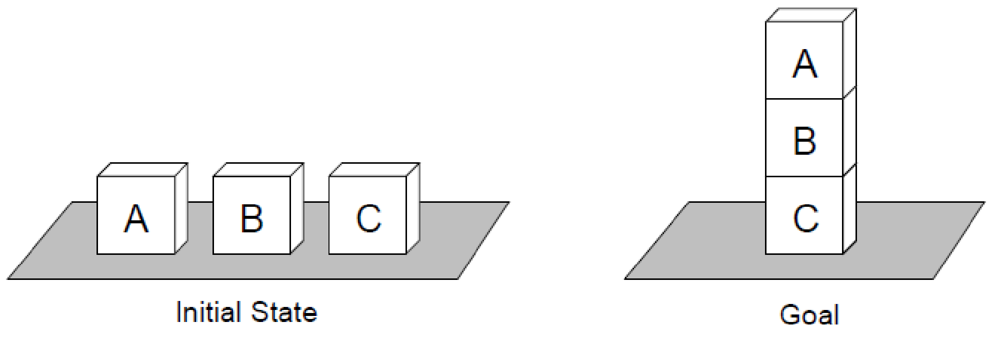


Figure 2.2: The initial state and goal of the "blocks-world" task.

Figure 2: The initial state of the system

its *actions* will be executed. Operator Selection is performed by SOAR's decision procedure, this happens when enough knowledge is gathered and the system needs to select which production to fire (which operator will be applied). This can go wrong in three ways:

1. An operator cannot be selected, because none is proposed
2. An operator cannot be selected, because multiple ones have been proposed, but not enough information is present to make a decision.
3. An operator has been selected, but the system does not know how to apply it.

We say that the system is in an *impasse*. There are ways to get out of an impasse, these will be discussed later.

An example: Blocks World Task A simple example is as follows: we have three blocks, A, B and C, on a table. This is the initial state. Operators move one block at a time to another location (on top of another block, or on the table). Each of these operators is represented by the name of the block being moved the current location of the block and the destination of the block. The goal is to make a tower A on top of B on top of C. A representation of this can be seen in Figures 2 and 3.

As mentioned in the introduction, the source code for this program can be found in the SOAR users manual.

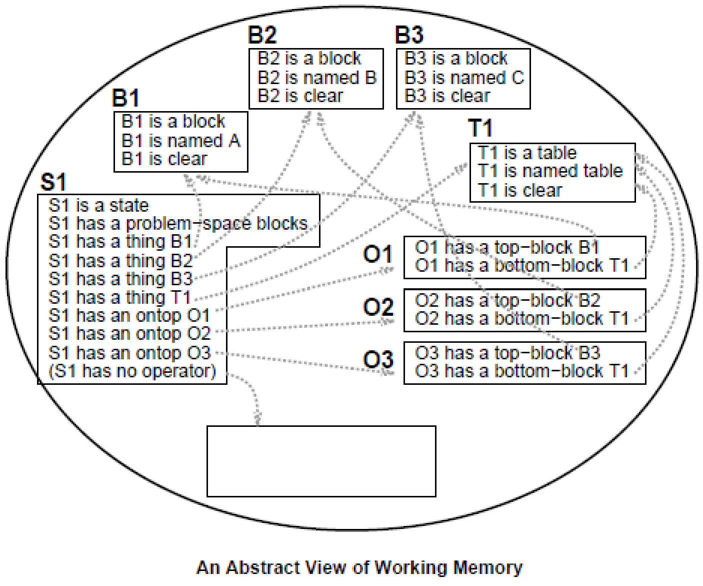


Figure 3: An abstract representation of the working memory at the beginning of the simulation, no operators have been selected

Representing states, operators and goals A state can have only one operator at a time, but it can also have a substructure of *potential* operators, as can be seen in Fig. 5

Goals can be represented explicitly as a substructure of state with general rules that recognize when the goal is achieved, or implicitly by goal-specific rules that test the state for goal-conditions. In this case the goal is implicit, as there is one specific production rule that inspects the state and terminates SOAR when the goal is achieved.

From proposing to applying an operator First, our production rules will test the state and propose all possible operators. Just as in ACT-R, one operator has to be selected from this set. This is done by using rules that test the state, but also test the proposed operators and create preferences. For example you may create a rule that sais that it is better to select A than B. This may lead to 4 different situations:

1. A single operator is selected
2. Multiple operators are suggested, from which a random selection can be made

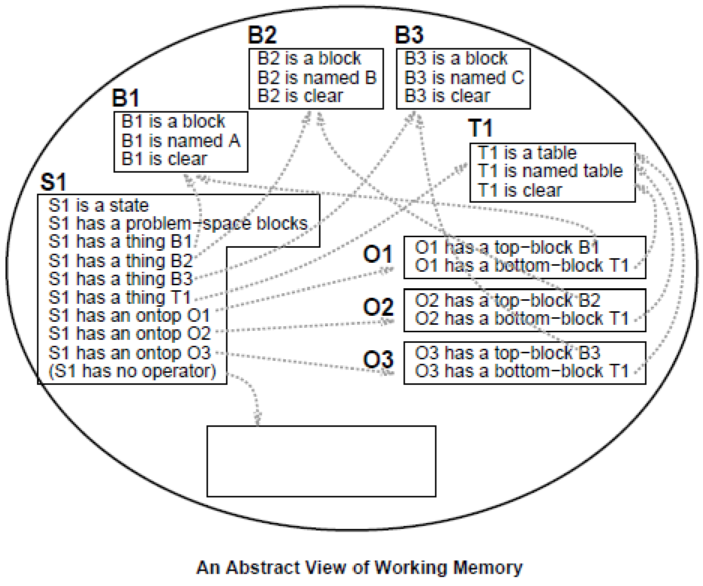


Figure 4: After selection of the first operator

3. Multiple operators are suggested, but a random choice can not be made.
4. No operators suggested

3 and 4 correspond with the aforementioned *impasse*. If an operator can be selected, its actions are performed and the state is adjusted accordingly. Modifying the state can be done directly and indirectly. Indirect changes are interactions with an external environment, while direct changes are done internally, directly to the state. This can be seen as SOAR doing problem solving 'in its own head'. Often a combination of both will be necessary.

Working Memory The working memory holds the current state and operator and is SOAR' short-term memory, reflecting our current knowledge about the task at hand. It consists of elements called working memory elements or WME's. Each WME contains a specific piece of information, such as: 'B1 is a block'. Multiple WME's may contain info about the same object, for example: 'B1 is on the table' on top of the info we already had. In this case we call B1 an *identifier*. All WME's that share the same identifier form an object in working memory, each WME is thus an attribute. Each attribute has a value attached to it, so a WME can be described by an *identifier-attribute-value*-triple. Objects can be linked,

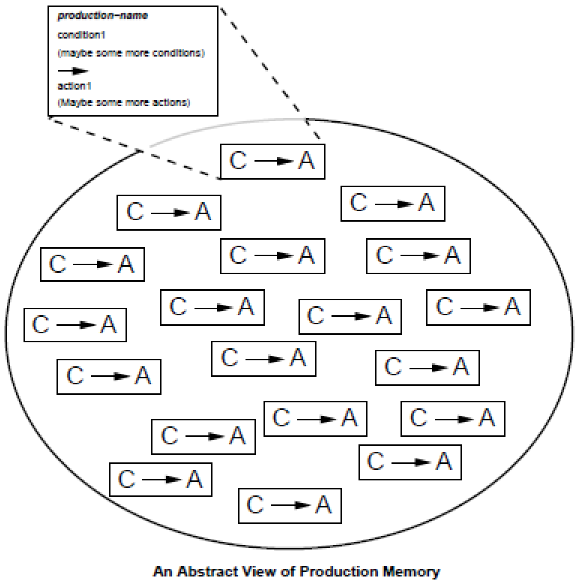

production-name
condition1
(maybe some more conditions)
action1
(Maybe some more actions)
C → A
An Abstract View of Production Memory

Figure 5: Abstract view of production memory

the value of a WME can be an identifier of another object. Each object must be linked to a state, otherwise it will be automatically removed by SOAR. WME's are sometimes also called *augmentations*

Important to note is that the working memory is a set, there can never be two elements with the same triple at the same time. However, it is possible to have multiple WME's with the same identifier and attribute, but with a different value, in this case they are called *multi-attributes*.

An object is defined by its WME's, not by its identifier, the latter one is merely an internal marker for SOAR, they can only appear in working memory, not in productions. On top of this, there is no connection between objects in SOAR and real-life objects. Objects in working memory might represent a real object, but also a relation between objects (on-top-of), a class of objects (blocks), etc. These are just labels and its up to the programmer to name them adequately.

Production Memory

The production memory represents the long-term memory of SOAR, consisting of productions with conditions and corresponding actions. An example of a production:

```
CONDITIONS: block A is clear
block B is clear
ACTIONS: suggest an operator to move block A on top of block B
```

The order of conditions does not matter, as long as the first one directly tests the state. If, when debugging, the order of conditions appears differently, this will probably be because soar reorders them for optimizations.

In practice, variables will be used instead of literal names as in the example. Thus a production may match multiple times, each with different variable bindings, and suggest multiple operators. Just as in working memory, the objects referred to in the conditions of a production must be linked to a state. One must test the state object and all others must test the same state, or objects linked to that state.

Most of the time, an action of a production creates or removes WME's, or creates a preference for operator selection. These preferences only match as long as the production instantiation that created them holds. If the situation changes, SOAR removes them automatically. These preferences are said to have *I-support*.

On the other hand we have productions that apply an operator, such as placing block A on block B. These changes need to persist, even after the production no longer matches. This is called *O-support*. These need to be explicitly removed by a reject-action.

When productions are matched, all fire at once, at the same time, adding and removing WME's, after which operators are selected. This is slightly different from ACT-R.

Preference Memory Once some rules have fired, we need to select an operator (or set of operators), this is done using preferences. Preferences are suggestions about the current operator, or how suggested operators compare to other operators. If you want an operator to be selected, it will need at least one preference. This is done when you write a suggest operator rule, in the action part you can indicate the preference rule of that specific operator.

An example is an *acceptable* preference state, indicated by the (+) symbol. This will indicate that a candidate is applicable for selection. On the contrary you have the *reject* (-) preference, indicating that a certain value is not up for selection. For example in the block-system, if you place block A on top of block B, you put a reject preference on block B, because something is already on top of it, thus you can not move. Listing all the different possibilities would be outside the scope of this document, I refer to section 2.4.1 of the user manual for further details.

Soar's execution cycle Each cycle has 5 phases:

1. Input
2. Proposal
3. Decision
4. Application
5. Output

this continues until the halt command is encountered, or until the user interrupts SOAR. While processing, some preferences that resulted in the current operator might change, in this case all preferences are re-evaluated and if another operator would have been chosen, current operator augmentation of state is immediately removed. A new operator is selected in the next decision phase, so that no knowledge is missed.

Impasses It can happen that the decision making procedure can not decide which operator to use after evaluating preferences due to incomplete, no `acceptable` operators are suggested, or inconsistent preferences. The system enters an impasse, there are four different types:

- Tie impasse: if the preferences do not distinguish between multiple operators with an `acceptable` operator.
- Conflict impasse: If there are conflicting preferences such as: A is better than B and B is better than A.
- Constraint-failure impasse: If there is more than one `required` preference for an operator, or both a `required` and a `prohibit` preference.
- No change impasse: If no new operator is selected during the decision procedure.

Soar attempts to resolve this by creating a new state in which the goal is to resolve the impasse. In this substrate, operators are selected and applied in order to try and discover which of the tied operators should be selected. The initial state in this sub-goal contains a complete description about the impasse, such as which operates where tied (of if there where no operators at all). The knowledge required to solve the impasse can originate from different sources, there are no boundaries as to how this can be tried to solve. This can be searching to find out the different implications of different decisions, or asking an outside source for more data. It does not really matter, as long as operators are applied to states.

While trying to solve this impasse, it is perfectly possible that other impasses occur, SOAR then builds a stack of sub-states, where each state has at max. one substate and one superstate. Even though SOAR will focus on the most recently created state, it will actually try to solve all impasses on all levels, if a match is found that could solve a higher level impasse, the operator is applied.

Solving the impasses is done using the following techniques:

- Tie impasse: can be resolved by productions that create preferences that prefer one option or eliminate alternatives.
- Conflict impasse: can be resolved by preferences to require one option.
- Constraint-failure impasse: cannot be resolved by preferences, but by changing productions so that less `require` or `prohibit` preferences are created.
- No change impasse: Resolved by creating `acceptable` or `require` preferences for operators.

Learning When SOAR was able to resolve an impasse, it has gained access to knowledge that it could not access before. This enables SOAR to learn from processing the substate. One of the learning mechanisms present in SOAR is called chunking, where it attempts to create a new production called a chunk. It's conditions are the elements of the state that allowed the impasse to be resolved, it's action is the WME of preference that solved the impasse. Variables are inserted into the condition, so that similar situations in the future can be recognised and an impasse can be prevented.

Input and Output Input can be seen as SOAR's perception and output can be seen as SOAR's motor abilities. This can be used to interact with external environments, such as controlling a robot or controlling a flight simulator. It can receive input and send output using *input and output functions*. The former adds and deletes elements from workign memory in response to the external environment, the latter attempts to cause a change in this environment. More information on how to use these functions can be found in: the SML Quickstart Guide

2.3. Comparing SOAR and ACT-R

SOAR and ACT-R are two different interpretations of a cognitive architecture. They differ in the way they handle everything. [Muller et al., 2008] gives a nice overview. A nice overview of the theoretical differences in ACT-R and SOAR can be found in [Johnson, 1997]. A single model is implemented in ACT-R and SOAR, to learn about the differences between both. Some of the differences:

- ACT-R only allows access to only one chunk at a time is allowed, while SOAR has no limit. This can cause a different type of behaviour: if only a limited number of elements is accessible, reasoning will be restricted to these elements.
- ACT-R uses an activation function to retrieve items from the working memory. There is no way to request a specific item based on a value. SOAR on the other hand allows items to be retrieved based on a value, by letting you order your facts in the conditional statement of a production.
- ACT-R matches rules sequentially, while SOAR matches rules in parallel.
- There is a python implementation of ACT-R, and while SOAR can be controlled from a plethora of programming languages, creating the models has to be done in a SOAR-specific language.

The impasse is detected during the selection of the operator.

3. Applications in game engines

The use of cognitive architectures to model humans playing games is extensively discussed in [West et al., 2006]. They also provide us with an insight in how humans play.

3.1. Rock Paper Scissors

The noble sport of roshambo[1] was modelled by [Lebiere and West, 1999]. After providing us with an in-depth discussion on how the model works, extensive testing results are discussed to measure the performance of the model. In the end it closely resembled human behaviour.

3.2. ACT-R Unreal

In [Best and Lebiere, 2006] the goal was to develop agents capable of interacting in small teams in real-time environments, in which agents have to cooperate or try and hinder one another. This was done by implementing a model for Unreal Tournament[2] bots. The focus was on creating and testing the model, so no playability tests were done. An interesting fact: if low level perception is sufficiently separated from high level cognition, the code from this model can, apparently, easily be ported, to be used for, let's say, a robot.

3.3. SOAR Games Project

The SOAR games project had the goal of developing simulations for existing computer games [Lent and Laird, 1999]. Even though not much new information can be found, it indicates that SOAR is indeed a viable tool for simulating human behaviour and thus can be used for playtesting. Some of the simulations that inspired or originated from this project:

[1] `http://en.wikipedia.org/wiki/Rock-paper-scissors`

[2] `http://en.wikipedia.org/wiki/Unreal_Tournament`

3.3.1. Quake 2 and Descent 3

[Lent et al., 1998] describes the origin of the SOAR Games project and gives a short overview of how SOAR is used to create models for Quake and Descent. It gives a rather superficial explanation of how to connect SOAR to an external source.

The Quake part, SOARbot, is discussed extensively in [Laird et al., 2001]. It suggests a very interesting methodology for testing the performance of the bot, using an experiment resembling the Turing test. The performance of players of different skill levels, alongside the SOARbot were recorded. These recordings where then shown to a jury consisting of people with different levels of familiarity with the game. They had to vote whether a certain performance was done by a computer, or by a human. On top of this, it is explained how multiple skill levels are achieved and several performance metrics are suggested. This is the best documented example available.

3.3.2. USARSim

USARSIM was a model for Unreal Tournament 2004. All information can be found on: `http://code.google.com/p/soar/wiki/USARSim`. Unfortunately, more details are lacking.

3.3.3. Infinite Mario

Infinite maria is a simulation in which Agents play a variant of Super Mario, a complete side-scrolling video game with destructible blocks, enemies, fireballs, coins, chasms, platforms, etc. The state space is complicated, but factored in an object-oriented way, which captures many aspects of the real world. More information on `http://code.google.com/p/soar/wiki/Domains_InfiniteMario#Infinite_Mario`. SOAR was used to write such an Agent for the RLCompetition20090

This is one of the most recent examples that we found, accompanied by a few academic publications such as: [Mohan and Laird] and [Mohan and Laird, 2011].

3.3.4. TacAir SOAR

TacAirSOAR uses SOAR to power a flight simulator. This was the next step after the first person shooters previously implemented. Where the latter still have relatively simple AI, the complexity of the AI of the former is magnitudes larger. This AI is described in [Laird et al.] and [Laird et al., 1998].

3.4. General

We will list some remaining general information pertaining to using cognitive models in game development. They delve deeper into specific techniques or requirements. These are not discussed here, but we will look back at them if deemed necessary in the future.

Modelling human performance is discussed in [Ritter et al., 2002] and [Pronovost and West, 2010], while [Lebiere and Best] gives an idea about using cognitive architectures to model Adversarial Behaviour.

In [Ritter and Wallach, 2001], a model for two-person games is superficially discussed. The interesting part is that they have created both an act-r and a soar model, so a comparison can be made. Neither of both really stands out, but there is a slight preference towards ACT-R for this specific problem.

Furthermore [Gluck, 1988] provides us with a general description of using cognitive architectures in flight simulators. Regrettably, specifics are not discussed.

Finally a general enquiry about connecting ACT-R and dynamic gaming environments can be found [Shah et al.].

4. A Pac-Man Casy Study

After providing an explanation about the internals of SOAR and ACT-R, a small case-study is provided as an example of how to use these tools in a game-development context. We will base our implementation on [Syriani and Vangheluwe, 2008], in which graph-rewriting is used to implement and simulate a basic version of the game Pac-Man. We will use the language and semantics used in this paper, so that results can be compared. In this section we will describe the structure and semantics of the game, in the following sections the actual implementation and simulation will be discussed.

4.1. The Pac-Man Language and Semantics

We can distinguish five different elements in Pac-Man: Pac-Man, the ghost, food (called pellets), GridNode and ScoreBoard. For our version, we only need to consider Pac-Man and Ghost as important entities, the other three elements are just part of the game-framework. Pac-Man and Ghost will be represented by different agents, controlled by the player or the ACT-R/SOAR system.

The operational semantics are defined by the production rules that are the core of all cognitive modelling tools. Rules will exist to move Pac-Man, to flee from a ghost, to eat food. Priorities can be given to rules to control decision flow and eventually to make an agent smarter (priorities in ACT-R and preferences in SOAR).

4.2. Using Cognitive Modelling tools to simulate behaviour

The basic idea is that we write a framework providing the basic Pac-Man game and the necessary functions to efficiently perform some analysis on. Then we can plug-in SOAR, ACT-R, or any other tool for that matter, providing the intelligence behind the ghost or pac-man. We can then let it run once, visual if possible, or multiple times so that we can draw conclusions about our system or that we can test the playability of our game.

Each pass of the game-loop the current state of the game is passed on to whatever system is chosen, which in turn returns the necessary commands to move pac-man and/or the ghost (cfr. the Motor System in ACT-R and SOAR).

4.3. Creating a playable game model

If we want to create a game that is actually playable we have to take into account two factors:

- Firing rules in ACT-R and SOAR takes a specific amount of time (50 ms). All of the systems besides the production system use time as well, to represent delays in human thinking. This is not necessarily applicable to all game-agents and all situations in a real time game.
- You need to take into account playability issues, such as a ghost moving to fast in relation to a human player. Your ghost can not be too smart, or the game will not be fun to play at all.

4.4. Modelling the player and the game

We would like to be able to quantify the quality of the game. This means we should be able to let the game run automatically and thus we should create a model for our players. Concerning pac-man, there are two important factors to be taken into account: reaction speed and decision analysis (e.a. path finding). If we want to be able to evaluate playability, we need a performance metric, in this case the number of wins vs. losses. We should be able to win often enough, but not all the time.

Different players may use different strategies, the following three will be used:

- Random: PacMan will move randomly, inlcuding illegal moves.
- Dumb: PacMan will take into account its directly adjacent nodes, it moves to the adjacent node that has food on it, but not a ghost.
- Smart: PacMan recieves a global view of the playing field. You use pathfinding to compute the shortest path to the nearest food.

Implementing complex pathfinding in SOAR and ACT-R is not easy and a topic on itself. An easy solution is to just use the manhattan distance. Even better would be to use the internal memory and learning mechanisms of SOAR and ACT-R, to let them learn that ghosts are bad and pellets are good (and closer ones even better). It might be possible that a couple of runs are needed beforehand to tune the learning mechanisms before the actual simulation can be performed.

Exactly the same strategies can be used to describe different levels of intelligence in Ghosts, replacing food by Pac-Man.

4.5. Explicit use of time

Time is a critical aspect in this simulation since playability depends on the relative speed of the player (controlling PacMan) and game (Ghost). Our framework provides the game loop controlling the game, which is for all intents and purposes represents the continuous passing of time. The difficulties arise with the production system. In ACT-R each rule matches instantly, but takes 50ms to fire and top of this, when using complex modules from SOAR or ACT-R, time is added depending on which system is use, while in SOAR a decision cycle takes avout 50ms, although more rules can fire leading to the decision.

We want to be able to tweak these parameters, since in real life payer's reaction speed differs from game to game, and even within one game. The game will run at speeds starting at 100ms to 400ms using a 5ms interval. Player speed will be determined by using the inverse cumulative method and the following distribution:

$$F(x) = e^{-e^{\frac{b-x}{a}}}$$

with a = 33.3 and b = 284 for a slow user, a = 19.9 and b = 257 for a Normal user, a Fast user has a = 28.4 and b = 237 and a VeryFast user fas a = 17.7 and b = 222. This value will be used to configure the speed in ACT-R and SOAR.

For each game-speed we will average 100 samples simulated with different seeds. The parameter we will use as a performance metric is the *time until game ends*. We will also take note of the frequency with which a player will win a game. The goal is a win-percentage of 75%.

4.6. Extra information

ACT-R and SOAR both have simple games as part of the examples in the suites you can download. In the previous section some implementations of bigger projects using these tools have been mentioned, they all have papers describing the implementation. While ACT-R has a python implementation and thus makes it easy to do computations and extend the system just using python, it is a bit more complex in SOAR, since it has its own specific syntax and can not easily be extended, thus some extra information about path finding in SOAR can be found in

[Hangartner et al., 1994], `http://web.eecs.umich.edu/~soar/sitemaker/workshop/23/Kerfoot%20-%20High%20Level%20AI.pdf` and `http://web.eecs.umich.edu/~soar/sitemaker/workshop/21/Laird-Learning.pdf`.

IMPORTANT NOTE: The implementation of Pac-Man described in the following two sections does not adhere to the description of the system given in the previous section. From this follows that these are not a good representation of the capabilities of SOAR and ACT-R and as such the results are not usable or reliable. They do provide an example of working with ACT-R and SOAR and provide some useful references for future use.

5. Simulating Pac-Man in Act-R

The goal is to recreate the experiment as described by Syriani and Vangheluwe [2008] in act-r as well as in soar. Hopefully this gives some insight into the usability of both these tools for game development. First, the process in act-r is described, followed by the implementation in soar. As stated previously the python implementation of act-r will be used to implement the pac-man game. Once finished, playability tests will be performed.

5.1. The PacMan Language

The model for simulating PacMan consists of 2 agents, PacMan and Ghost. Fortunately, a 2D-grid based environment is already present in python act-r. GridNode is just a subclass of already existing code that is based on a model. Score and Pellet are just parameters of this model, they are not represented by individual entities.

5.2. Semantics

Each agent in a model contains a set of rules [3]. These rules specify the behaviour of the Agent. On top of this, each agent contains a goal-buffer, called focus. Different rules fire depending on the contents of this buffer. An example of a rule:

```python
def look_for_pellet(focus='looking',vision='busy:False',visual=
    None):
        vision.request('pellet:True x:?x y:?y')
        focus.set('found')
```

Listing 1: Example rule from the PacMan agent

[3]The full set of rules and the environment can be found in Appendix 1

This rule fires if the focus is looking for a pellet. If this is so, this rule fires, and requests from the vision system the location of a pellet. The focus is changed and a new rule will fire. It is important to note that in act-r rules are fired sequentially. If two rules happen to match the same buffer, one is selected randomly from the matches. A system is in place to give preference to rules that are called more often, but in this model this has no effect and all rules have the same chance of being selected. Just as in the aforementioned paper, the focus will be on playability, so score will be ignored.

5.3. The PacMan case study

There are two main agents in the PacMan model, PacMan and Ghost. Both have a Motormodule, a visual memory and a module to detect obstacles; PacMan wanders around, asks is visual system for the location of a pellet and directly heads to the location reported while trying to evade the Ghost. The Ghost requests the location of PacMan and heads towards the reported result, updating its course if PacMan moves. The motormodule is just a high level representation of movement. It is an abstraction of moving in different directions. On top of this the obstaclemodule just makes sure that PacMan or Ghost can not move through walls.

PacMan selects a pellet based on a property called salience. Salience indicates how much an item stands out in a collection [Stewart and West, 2007]. An illustration: in a row with blue balls, a red ball has a high salience. Standard, chunks with a high salience have a high preference. Since there is only one node with a ghost present, this will be chosen every time. Obviously this is not the desired behaviour, on the contrary, in this case PacMan would move straight to the Ghost every single time. With some workarounds it is possible to use a user-defined function to calculate salience in stead of the built-in salience mechanism. Now, salience is calculated by taking into account the distance to a square and the presence of a ghost in the neighbourhood. If different squares have identical salience, one is randomly chosen. Ghost just spots PacMan, and approaches using the manhattan distance.

This is the method used to model the *smart player*, modelling the random and dummy player is perfectly possible as well, by restricting the view of PacMan to only adjacent nodes or by just creating a move rule for every direction, such that each one of these rules has the same chance of firing. Given the fact that only the smart user is discussed in the simulations, this is not further discussed.

5.3.1. *Use of time*

We need to make sure that we do not change the semantics as described in [Syriani and Vangheluwe, 2008]. In that simulation, every transition happened instantaneously except when making a decision. This parameter can be varied for simulation purposes. We will do the same by waiting a specified amount of time when making a decision:

```
def look_for_pellet(focus='looking',vision='busy:False',visual=
    None):
        vision.request('pellet:True x:?x y:?y')
        focus.set('wait')

def wait(focus='wait'):
        import time
        begin = time.time()

        while(time.time() - begin < pac_production_time):
            pass

        focus.set('found')
```

Listing 2: Example rule from the PacMan agent

Requesting the location of PacMan corresponds to the moment in time where you make a decision. After this the production process continues as normal.

5.4. *Experiment*

Player decision time is taken from the same distribution as in Syriani and Vangheluwe [2008], with the exact same parameters:

$$F(x) = e^{-e^{\frac{b-x}{a}}}$$

where a slow user has a =33.3 b = 284, a normal user a =19.9 and b = 257, a fast user a =28.4 and b = 237 and a VeryFast user with a =17.7 and b = 222. We sample from this distribution using the Inverse Cumulative Method.

Only the smart movement strategy is considered and the length of the simulated game is measured on a board consisting of 22 food pellets. Ghost decision time was varied from 100ms to 400 ms, but due to timing constraints, the average time of a game is quite a bit longer than in the original results, an average over

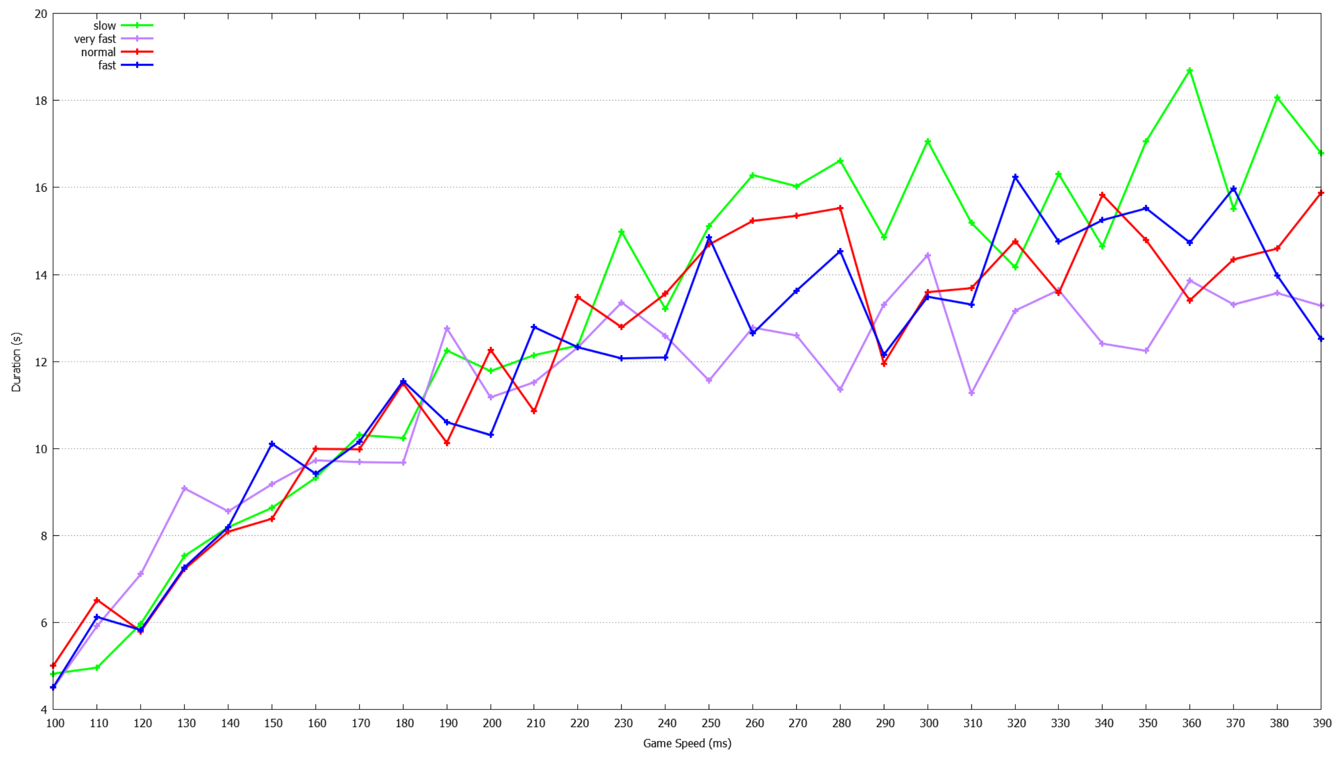


Figure 6: Time till end

50 samples was taken instead of 100 samples. The results are shown in 6. A reminder: speed here is the time it takes for a Ghost to make a decision. On first sight our results are comparable with the results from the original experiment, except the fact that games last 2.5 times longer for slower reaction times. Despite this, the same idea is valid, the slope of the curve implies that the slower the decision time, the longer the game lasts and after a certain limit, we get to see the same plateau, but this occurs 100ms sooner, at 250 ms. An explanation for this is that PacMan does not follow an optimal strategy and, more importantly, by letting each rule take quite a lot of time, games will last longer automatically. This is inherent to the way act-r works and how this model was made.

7 depicts the frequency with which a player will win a game as function of the time spent on the Ghost's decision. A result of 75 % is considered a good amount of wins to keep a game interesting, yet challenging enough. The graph shows that this model does not even reach this percentage. The best result is a 50% win rate.

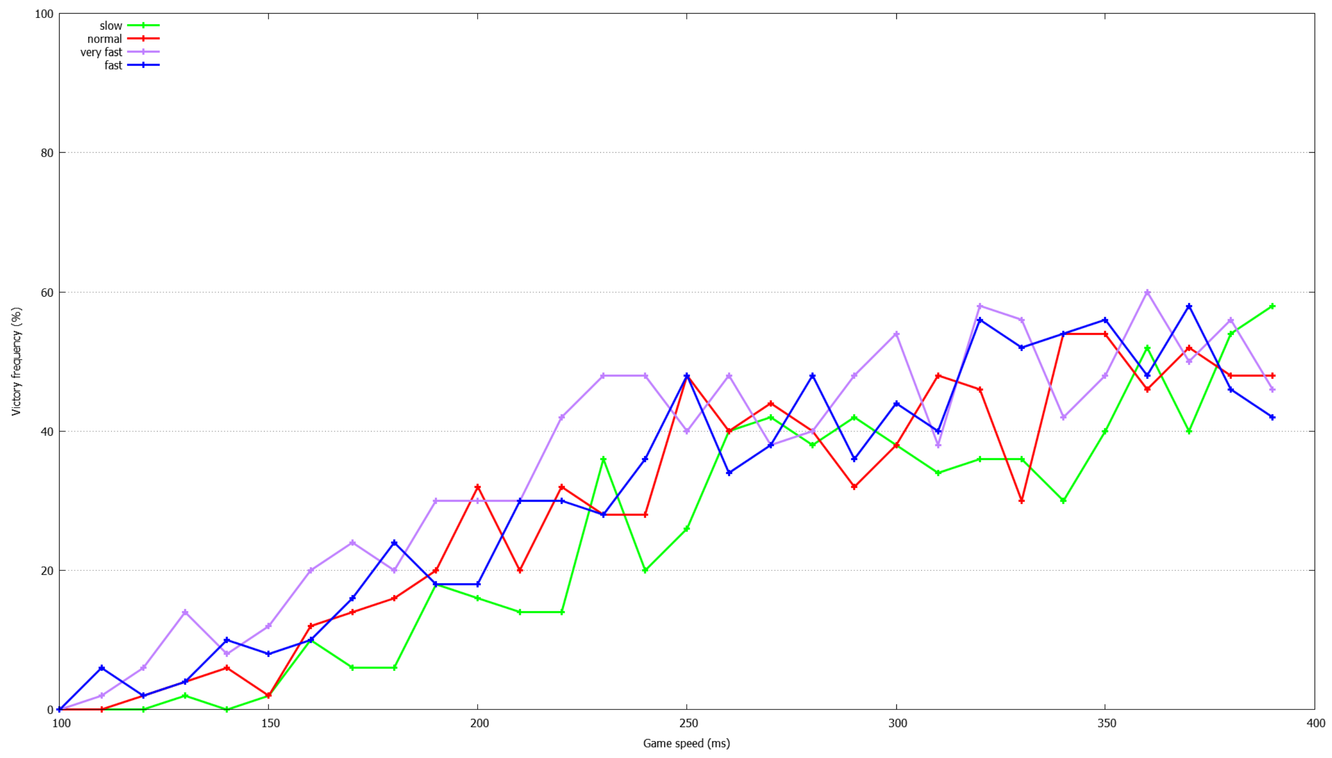


Figure 7: Victory Frequency

Obviously this is not enough. The main cause for this is the fact that the salience system used for pathfinding is not adequate enough. If the game is run in visual mode, so that the progress of the game is visible, it is clear that the model of a human player still shows some quirks and that, once in a while, stupid moves are made. This reduces the win rate drastically. Even raising the Ghost Decision time has no remarkable effect.

5.5. Conclusion

It is definitely possible to recreate the experiment and develop a basic AI in act-r. The results were not as good as in the original experiment, but the model is to blame for this, not act-r. If the model would be further refined, I am certain these problems can be solved. This is in itself the biggest shortcoming of this tool: the lack of examples and documentation. There are some examples available, but these only show a basic use of act-r, making a complex model is an art in itself.

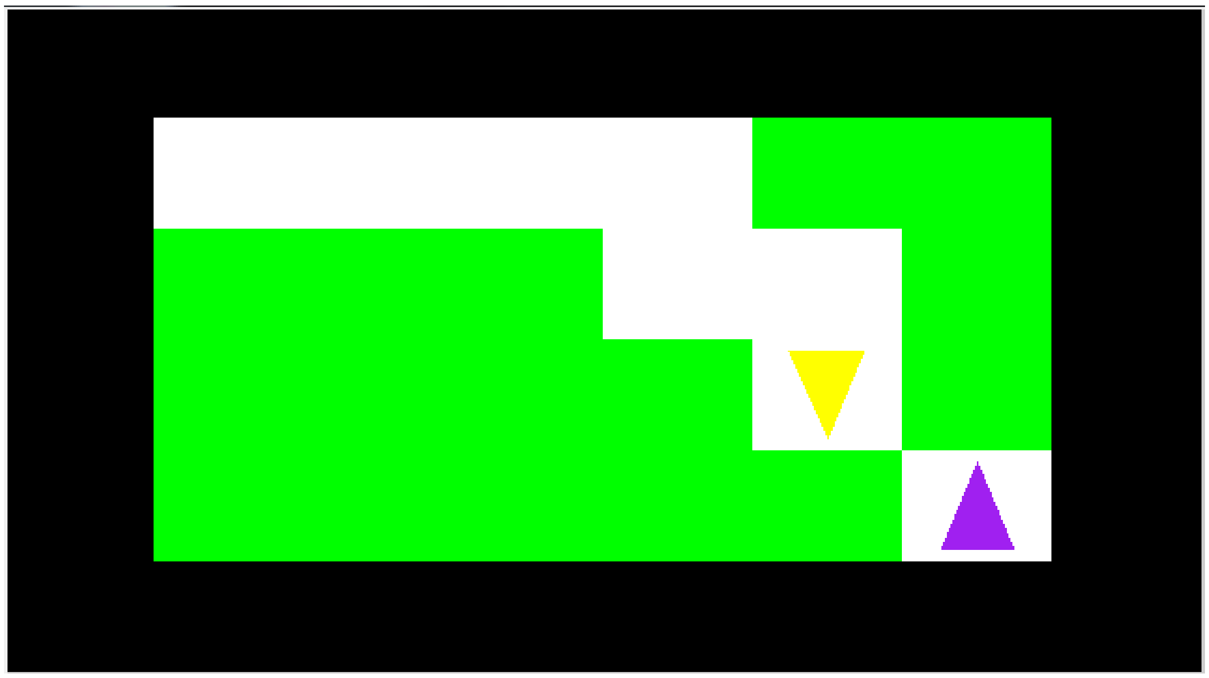


Figure 8: The ACT-R Pacman environment. The yellow arrow represents PacMan, while the purple arrow represents a Ghost. If a square is green, it means there is still a pellet present

A lot of papers can be found, describing different solutions for various problems, but these are only descriptions, the code is, more often than not, not available. In the short timespan of this research it was not possible to contact authors and developers requesting more information.

Concerning the explicit use of time, a better solution may exist. As mentioned before in 3.2, an unreal tournament bot has been implemented using Act-R and sockets, where the rules keep on firing and at certain intervals the act-r system is polled and the necessary information is passed along, though it was not feasible implementing this for a project of this scope. If act-r would be used in a bigger project, this will probably work better than the method used in this experiment, even though, again, documentation is severely lacking.

6. Simulating Pac-Man in SOAR

In this section, we will try to implement the same experiment as in the previous chapter, but using SOAR instead of ACT-R. SOAR contains an example called Eaters. This is actually an implementation of PacMan, but without Ghosts. The behaviour for PacMan is thus already present, the only two things lacking were chasing (for Ghosts) and running away. Luckily, SOAR had another example in the same environment called TankSOAR. This contained a model for chasing and running away. With all this information already present the final model for PacMan was easily created.

6.1. The PacMan case study

PacMan in SOAR is not a really complex environment. We need two agents, namely PacMan and Ghost. The procedures for moment are the same for both. They only differ in the fact that PacMan wants food and Ghost wants PacMan. The actual model will be provided as an addendum.

Once again, the idea was to only use the *smart* player model. As in ACT-R, the other two, random and dummy are easy to implement.

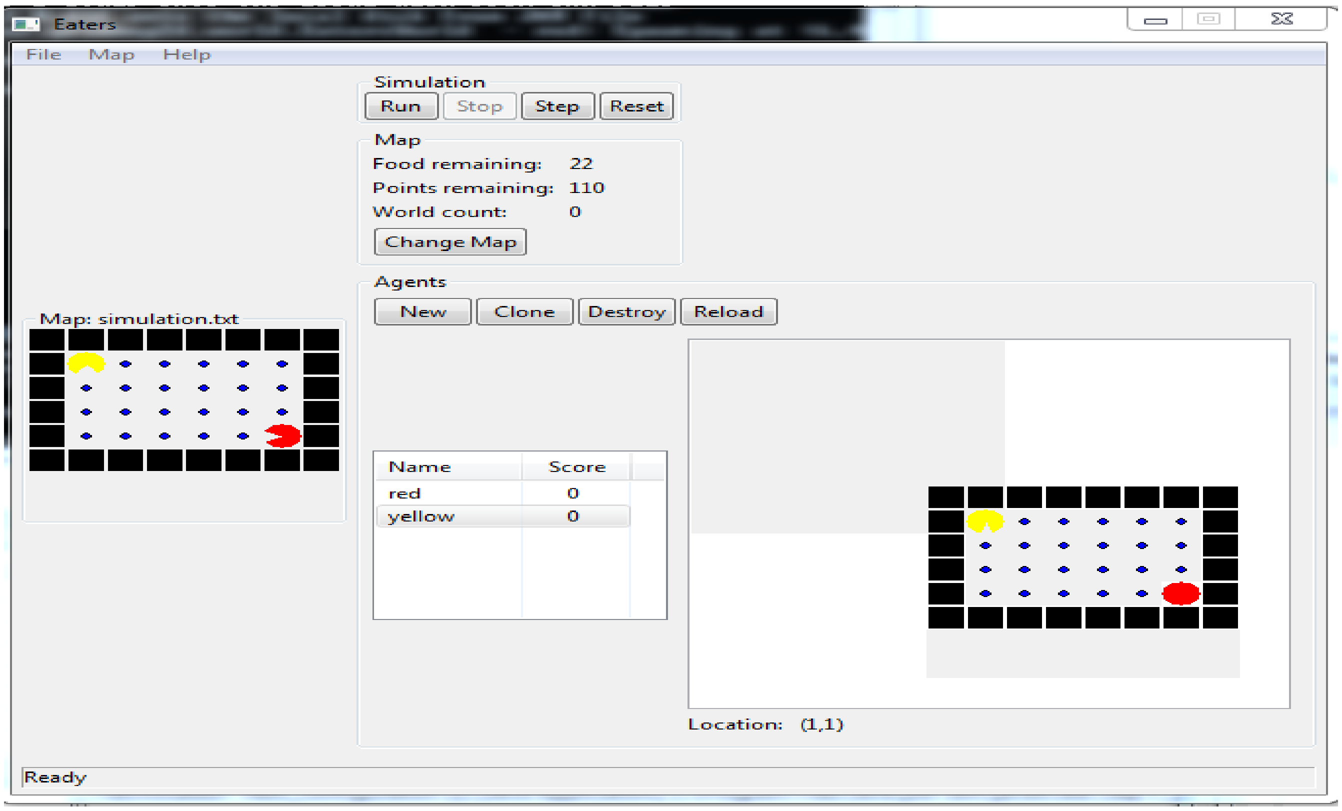
6.1.1. Use of time

A big advantage of SOAR is that you can easily link it to other languages. Using swig [4] a wrapper is provided so we can interface with SOAR from Java and Python amongst others. This allows to just program a game loop. At the beginning of the loop we can give the state of the environment to SOAR, if we run out of time, we request the results. This is an improvement over how we implemented time-slicing in the act-r version.

6.2. Experiment

Implementing the model might have been easy, but recreating the experiment proved very difficult. We could not recreate the same circumstances. While the implementation of PacMan used configuration files that contained game-speed

[4] `http://www.swig.org/`



Food remaining: 22
Points remaining: 110
World count: 0

Map: simulation.txt

Name	Score
red	0
yellow	0

Location: (1,1)

Ready

Figure 9: The PacMan environment. On the left you can see the game field. On the right is a representation of what pacman can see, in this case the entire playing field

parameters, changing them had no effect at all on the speed the game ran. After checking the code, it became obvious that this parameter is never used. We made the necessary changes to support this, but than we stumbled across another problem. Both PacMan and Ghost use the same game-speed. This means you can not change them independently. This makes it impossible to study the effects of changing the Ghost speed on the performance of PacMan. We did not succeed in rewriting the system on time.

6.3. Preliminary conclusion

Even though we were not able to execute the experiment, we remain convinced that this was caused by a lack of knowledge and experience. We strongly believe that creating an environment that supports running two agents separately from each other is feasible. If we would use SOAR again in the future, we will make another attempt at this, but this time starting from scratch and writing a new PacMan version supporting Ghosts from the start. The fact that it is so easy to interface with SOAR, really is a big advantage.

6.4. Suggested solution

It is possible to implement the notion of time, or ordering in a rule based environments. It is possible to use rules to create a counter. Certain rules can then only fire if the coutner reaches a certain value, after which the counter is reset. By using this technique you can mimic the effect that a Ghost needs to think longer (or faster). Unfortunately, we were not able to adjust the soar model in time for the deadline. The ACT-R model did get adapted in time, so an example of this technique can be found in . It follows that we are convinced that this is definitely a plausible solution.

7. Overview of already existing techniques

Many different techniques exist to describe the behaviour of entities in video games. We will discuss some of the most frequently used techniques. This is not by any means an exhaustive list, nor do we want to give one, the goal is to give an idea about what different techniques exist.

7.1. Plain old programming

One of the oldest ways to control an AI is to just program it.Typically, A game designer describing a character does not have the required knowledge to implement this in a programming language. This typically requires a programmer, preferably specialised in AI. Another downside is that it is error prone. All this extra code needs to be tested and bugs might prove hard to find. It can be very expensive as well, not only do you have to pay a programmer, repairing mistakes is a very costly process. On the other hand, for certain critical parts that need to be highly efficient it might be better to code them than to model them.

Often there is some overlap with 7.3. Even when not explicitly using statecharts, most of the time, some form of event-driven system is in place to control and direct the actions of agents present.

7.2. Scripting

In modern game engines the core functionality, like graphics, is separated from the gameplay code. The core is written in a highly efficient programming language like c++, while (parts of) the gameplay are written in a scripting language. These are often simpler to use and easier to learn. Game designers can often do this task themselves, it is easier to teach a designer a scripting language than finding one that has mastered C++. On top of this, if wanted,it allows users to easily create their own content. A lot of contemporary games support the use of scripting in modifications written by users.

Lua [5] is such a scripting language that is used quite a lot[6]. A description of the language can be found in [Ierusalimschy, 1996]. Some advantages are: Portability, ease of embedding, simplicity and efficiency.

[5] `http://www.lua.org/`

[6] `http://en.wikipedia.org/wiki/Category:Lua-scripted_video_games`

7.3. Statecharts

Behaviour of characters is inherently event driven. When an event occurs, a character reacts. Statecharts (see [Harel, 1987] are thus an excellent tool to model this behaviour. In a statechart the system is described in a finite number of states. In between different states there can be transactions. It is then possible to define events that allow the system to change from one state into another. A small example: A guard may be in the *exploring* state. On the *I hear something* event he makes a transition to the *Searching* state. If, after a while, nothing is found he returns to the exploring state. On the other hand, if something is found he transitions into the *Shoot* state and so on.

These statecharts can then be compiled using a statechart compiler, such as scc[7]. This is currenlty being researched by Gino Wuytjens of the msdl research group at the University of Antwerp. A preliminary version of his master's thesis on this subject can be found at `http://msdl.cs.mcgill.ca/people/gino/thesis_intro.pdf`.

The big advantage over programming is that, using a statechart compiler, the event driven system does not have to be coded anymore. Designers can draw the necessary diagrams and code is generated automatically, which is obviously an improvement.

7.4. Behaviour Trees

Behaviour Trees are a relatively new way to represent the behaviour of agents. They focus on increasing the modularity of states by encapsulating logic transparently. It is an evolution from state machines by removing the transitions from states. This results in a collection of actions that can execute. Transitions are inherently contained in the scope of which a node is a member. [8]

A lot of new games use these: Halo [Dyckhoff, Max], Crysis 2 [Pillosu], Defcon [Lim et al.], Grand Theft Auto 4 and so on. It is a widely discussed topic on game developer conferences and a lot of information can be found. They have the big advantage of being simple. At Big Huge Games, designers create their own behaviour trees [Champandard and Dawe] using lua for non-critical sections.

[7] `http://msdl.cs.mcgill.ca/people/tfeng/uml/scc/`

[8] `http://www.altdevblogaday.com/2011/02/24/introduction-to-behavior-trees/`

7.5. Rule Databases for Contextual Dialog and Game Logic

A technique, only very recently described by Valve[9] at the Game Developers Conference of 2012 in [Ruskin, 2012]. Orignally a system designed to enable dynamic conversations in video games, it allows for the creation of dynamic behaviour (amongst others) as well.

The basic idea is that every object has a state. Every few seconds, a database containing lines, audio files or scripts is queried with a query constructed depending on the state of the object in question. Each match receives a score, the best one is returned. If multiple results have the same score, the answer is randomly chosen.

This allows for a very reactive, dynamic system that can easily be adapted. Many extensions have been made to the core system to allow it to be even more flexible.

Up until now, Valve has used it to power the dialogue in their games, but they created a proof of concept for using this system to power behaviour.

[9]`http://www.valvesoftware.com/`

8. Future work

It is not yet clear what exactly will be done in the future. The results and conclusions of this research have to be processed first. What will definately happen next is recreating the PacMan experiment using TCore [Syriani and Vangheluwe, 2010] to try and find out if model transformations may be applied to generate behaviour for npcs.

Whatever tool will be used, the experiment mentioned in [Laird et al., 2001] may be interesting to implement in the future to test the how human-like our simulation behaves. This does require an implementation in a game that is complex enough.

Once the choice is settled, implementation in a contemporary, commercial game engine should be the goal to achieve. If we want to convince people that our methods are of any practical use, we need to show it to them with the tools they already use.

9. Conclusion

Currently there are many techniques available to allow developers to more easily create realistic behaviour in characters. Despite this, it is still a complex and daunting task. Rule-based modelling can abstract away implementation details and provide a new level of abstraction and provide us with new methods to test playability in video games.

Two conclusions can be made from this reading study. They apply to both ACT-R and SOAR. One of the goals was to make the development of AI easier, to devise a way that developers without specific knowledge could create their own behaviour without relying on programmers. Do cognitive architectures make this easier? This is not an easy question to answer. The core of these systems, the production system, is very straightforward. It is easy to see how rules work and interact together, it is basically an action and reaction system. All the extra modules that are available are not that complex when we look at them individually. The complexity lies in adding them all together and the effect that they have on each other. There are a lot of parameters that can be tweaked and the effect they have is not always clear from the start. Once you want to create more complex models, creating one that is realistic is a matter of experimenting a lot with these parameters. Creating a good model that simulates a specific human task well is a challenge, it requires good insight into the architecture and a lot of experience.

It might be possible to combine certain parts, like the learning module and the memory module, with different ways of implementing an AI. You could combine a behaviour tree with a production system, where each node executes a set of rules while in the background the learning and memory module can affect the decisions you make. The feasibility of such a system needs to be investigated before further comments can be made.

Whether or not they are good tools for creating AI, if you want to simulate your game, it might be worth developing an agent using a cognitive architecture. In this context it does not matter that much that it takes more time, you just want a model that is as realistic as possible.

10. PyACT-R PacMan model

10.1. Model Code

```
pac_production_time=0.2
ghost_production_time=0.4
user="normal"
'''
Created on 30-jul.-2012

@author: Kevin
'''
import ccm
import itertools
log = ccm.log(data = True)

from ccm.lib import grid
from ccm.lib.actr import *
from ui import swi
from ccm.lib import cellular
map = """
########
# ppppp#
#pppppp#
#pppppp#
#ppppp #
########
"""

#==============================================
# map = """
# ####
# #pp#
# #pp#
# ####
# """
#==============================================

class MyCell(grid.Cell):
    '''
    classdocs
    '''
    pellet = False
    def color(self):
        if self.wall: return 'black'
        elif self.pellet: return 'green'
        else: return 'white'

    def load(self, char):
        if char =='#': self.wall = True
        elif char =='p': self.pellet = True

class GhostBody(grid.Body):
    def _list_visible_objects(self):
        for row in self.world.grid:
            for cell in row:
                if not cell.wall is True:
                #if cell.pellet is True:
                    if self.cell != cell:
                        yield cell

        for obj in self.world.agents:
            yield obj

class PacManWorld(grid.Body):
    def _list_visible_objects(self):
        for row in self.world.grid:
            for cell in row:
                if not cell.wall is True:
                #if cell.pellet is True:
                    if self.cell != cell:
                        yield cell

        for obj in self.world.agents:
            yield obj

    def go_towards(self,target,y=None):
        for obj in world.agents:
            if obj.color is "purple":
                ghostx = obj.cell.x
                ghosty = obj.cell.y
               # print "Location of ghost is", ghostx, ghosty
        bestDist = 0
        if not isinstance(target,cellular.Cell):
            target=self.world.grid[int(y)][int(target)]

        if self.world is None: raise cellular.CellularException(
            'Agent has not been put in a World')
        if self.cell==target: return
        best=None
        for i,n in enumerate(self.cell.neighbours):
            if n==target:
                best=target
                bestDir=i
                break
            if getattr(n,'wall',False): continue
            dist=(n.x-target.x)**2+(n.y-target.y)**2
            if (best is None or bestDist>dist):
                if not (n.x == ghostx and n.y == ghosty):
                    best=n
                    bestDist=dist
                    bestDir=i
        if best!=None:
            if getattr(best,'wall',False): return False
            self.cell=best
            self.dir=bestDir
            return True

class GhostWorld(grid.Body):
    pass

class PacVisionScanner(grid.VisionScanner, MemorySubModule):
    def activation(self, chunk):
       # print "Returning activation", chunk["salience"]
        return chunk['salience']

    def start(self):

        while True:
            for obj in self._body._list_visible_objects():
                if hasattr(obj,'cell'):
                    if getattr(obj,'x',None)!=obj.cell.x: obj.x=
                        obj.cell.x
                    if getattr(obj,'y',None)!=obj.cell.y: obj.y=
                        obj.cell.y
                try:
                    pellet = obj.cell.pellet
                    ox,oy=obj.x,obj.y
                    x,y=self._body.x,self._body.y
                    salience=self.salience(ox-x,oy-y, ox, oy,
                        pellet)
                    obj.salience = salience
                except AttributeError:
                    salience=1
                    obj.salience = salience
            #Items are added to the memory here
                self._visual.add(obj)

            yield self.scan_time

    def salience(self,dx,dy, ox, oy, *pellet):
        #print "Calculating Salience"
        import math
        for obj in self._body.world.agents:
            if obj.color is "purple":
                ghostx = obj.cell.x
                ghosty = obj.cell.y

        ghostx = 5
        ghosty = 5
        #print self._body.x
        dgx = ghostx - ox
        dgy = ghosty - oy

        dist=dx*dx+dy*dy
        dist_g = dgx*dgx + dgy * dgy

        s_d=  2/(math.sqrt(dist)+1)
        s_g = (math.sqrt(dist_g) +2)/5

        s = s_d + s_g
        #if pellet: s += 1
        #if s>1: s=1
        return s

class GhostVisionScanner(grid.VisionScanner):
    def salience(self,dx,dy):
        return 1

class GhostMotorModule(ccm.Model):
    busy=False

    def go_towards(self,x,y):
        if self.busy: return
        self.busy=True
       # print 'Ghost going towards %s %s'%(x,y)
        self.action='going towards %s %s'%(x,y)
        #yield 0.01#self.parent.production_time
        self.parent.body.go_towards(x,y)
        self.action=None
        self.busy=False


class PacMotorModule(ccm.Model):
    pellets = 22
    busy=False

    def go_towards(self,x,y):
        if self.busy: return
        self.busy=True
       # print 'going towards %s %s'%(x,y)
        self.action='going towards %s %s'%(x,y)
        #yield 0.01# self.parent.production_time
        if self.parent.body.cell.pellet == True: self.eat()
        self.parent.body.go_towards(x,y)
        self.action=None
        self.busy=False

    def eat(self):
        if self.parent.body.cell.pellet:
            self.parent.body.cell.pellet=False
            self.pellets = self.pellets - 1
            if self.pellets == 0:
                log.win = 1
           #  print "pellets left:", self.pellets

class ObstacleModule(ccm.ProductionSystem):
    production_time=0
    ahead=False
    left=False
    right=False
    def check_ahead(self='ahead:False',body='ahead_cell.wall:
        True'):
        self.ahead=True
    def check_left(self='left:False',body='left_cell.wall:True')
        :
        self.left=True
    def check_right(self='right:False',body='right_cell.wall:
        True'):
        self.right=True

    def check_ahead2(self='ahead:True',body='ahead_cell.wall:
        False'):
        self.ahead=False
    def check_left2(self='left:True',body='left_cell.wall:False'
        ):
        self.left=False
    def check_right2(self='right:True',body='right_cell.wall:
        False'):
        self.right=False

class Ghost(ACTR):
    ghost_production_time = ghost_production_time
    waitime = 0
    focus = Buffer()
    body = GhostBody()
    motor = GhostMotorModule()
    obstacle=ObstacleModule()

    visual=Buffer()
    vision=Memory(visual)

    visionScanner = GhostVisionScanner(body,vision)

    def init():
        focus.set('wander')
        visual.clear()

    def look_for_pacman(focus='wander', vision='busy:False',
        visual=None):
        vision.request('color:yellow')
        focus.set('wait')
        #time.sleep(ghost_production_time)

    def wait(focus='wait'):
        import time
        begin = time.time()

        while(time.time() - begin < ghost_production_time):
            pass

        focus.set('found')

    def pacman_found(focus='found', visual='color:yellow x:?x y
        :?y'):
        waittime = 0
        focus.set('go towards ?x ?y')

    def going_to_pacman(focus = 'go towards ?x ?y', motor='busy:
            False'):
            if (self.body.x == x) and (self.body.y == y):
                self.stop()
            motor.go_towards(x, y)
            visual.clear()
            vision.clear()

    def arrived(focus = 'go towards ?x ?y', body='x:?x y:?y'):
        vision.request('color:yellow')

    def checking_location(focus = 'go towards ?x ?y', vision="
            busy:False", visual = None):
        vision.request('color:yellow')
        if (self.body.x == x) and (self.body.y == y):
            self.stop()

    def updating_location(focus = 'go towards ?x ?y', visual ='
            color:yellow x:?dx y:?dy'):
        for obj in self.body._list_visible_objects():
            if obj.color is "yellow":
                pacx = obj.cell.x
                pacy = obj.cell.y

                if (pacx == self.body.x) and (pacy == self.body.
                    y):
                    focus.set('lost')
                    break
        else:
            if (x != dx) or (y != dy):
                motor.go_towards(dx, dy)
                focus.set('go towards ?dx ?dy')

                visual.clear()
                vision.clear()
            else:
                focus.set('go towards ?x ?y')

    def finished(focus = 'lost'):
        self.stop()


class MyAgent(ACTR):
    waittime = 0
    pac_production_time = pac_production_time
    focus=Buffer()
```

```
body= PacManWorld()
motor = PacMotorModule()
obstacle = ObstacleModule()

visual=Buffer()
vision=Memory(visual)

visionScanner=PacVisionScanner(body,vision)
vision.add_adaptor(visionScanner)

def init():
    focus.set('looking')
    visual.clear()
   # salience.context()

def look_for_pellet(focus='looking',vision='busy:False',
   visual=None):

    vision.request('pellet:True x:?x y:?y')
    focus.set('wait')

def wait(focus='wait'):
    import time
    begin = time.time()

    while(time.time() - begin < pac_production_time):
        pass

    focus.set('found')

def found_pellet(focus='found',visual='pellet:True x:?x y:?y
   '):
   # vision.request('color:purple')
    focus.set('go to ?x ?y')
    #visionScanner.salience(x, y)
    #vision.clear()

def go_to_location(focus='go to ?x ?y',motor='busy:False'):
    import math
    visual.clear()
    vision.clear()
    for obj in self.body._list_visible_objects():
        if obj.color is "purple":
            pacx = obj.cell.x
            pacy = obj.cell.y
            if (pacx == self.body.x) and (pacy == self.body.
                y):
                focus.set('lost')
                break
    else:
        #=========================================
        # # if(math.fabs(self.body.x - int(gx)) == 1) and (
            math.fabs(self.body.y - int(gy)) == 1):
        #       self.body.turn_around()
        #       focus.set("looking")
        #
        # else:#
        #=========================================
         motor.go_towards(x,y)
         focus.set('go to ?x ?y')


def arrived(focus='go to ?x ?y',body='x:?x y:?y'):
    for obj in self.body._list_visible_objects():
        if obj.color is "purple":
            pacx = obj.cell.x
            pacy = obj.cell.y

            if (pacx == self.body.x) and (pacy == self.body.
                y):
                focus.set('lost')
            else:
                motor.eat()
                vision.clear()
                visual.clear()
                focus.set('looking')

#===============================================
# def checking_location(focus = 'go to ?x ?y ?gx ?gy',
    vision="busy:False", visual = None):
#     vision.request('color:purple')
#
# def updating_location(focus = 'go to ?x ?y ?gx ?gy',
    visual = 'color:purple x:?dx y:?dy'):
#
#     if (gx != dx) or (gy != dy):
#         print "UPDATING"
#         focus.set('looking')
#         visual.clear()
```

```
    #          vision.clear()
    #===============================================

    def lost(focus='lost'):
        self.stop()

    def finished(motor='pellets:0'):
        self.stop()

world=grid.World(MyCell,map=map, directions = 4)

pacman = MyAgent()
pacman.body.color='yellow'
world.add(pacman,x=1, y=1, dir=2)

ghost = Ghost()
ghost.body.color = "purple"
world.add(ghost, x=6, y=4, dir=0)

#ccm.log_everything(pacman,log.yellow)
#ccm.log_everything(ghost, log.purple)

#display = ccm.display(world)
world.run()
ccm.finished()
```

Listing 3: The PacMan model in ACT-R

10.2. Simulation Code

```
import ccm
import math
import random
slowa=33.3
slowb=284
normala=19.9
normalb=257
fasta=28.4
fastb=237
vfasta=17.7
vfastb=222

#Test for the slow user
for i in range(0,100):
    print "run", i + 1, 'from', 50
    for x in range(100,400,10):
        u = random.uniform(0, 1)
        pac_speed = -(math.log(-math.log(u))*slowa) + slowb
        ccm.run('pacman',1, pac_production_time=pac_speed*.001,
            ghost_production_time=x*.001, user='slow' )


#Test for the normal user
for i in range(0,50):
    print "run", i + 1, 'from', 50
    for x in range(100,400,10):
        u = random.uniform(0, 1)
        pac_speed = -(math.log(-math.log(u))*normala) + normalb
        ccm.run('pacman',1, pac_production_time=pac_speed*.001,
            ghost_production_time=x*.001, user='normal' )

#Test for the fast user
for i in range(0,50):
    print "run", i + 1, 'from', 50
    for x in range(100,400,10):
        u = random.uniform(0, 1)
        pac_speed = -(math.log(-math.log(u))*fasta) + fastb
        ccm.run('pacman',1, pac_production_time=pac_speed*.001,
            ghost_production_time=x*.001, user='fast' )


#Test for the very fast user

for i in range(0,100):
    print "run", i + 1, 'from', 50
    for x in range(100,400,10):
        u = random.uniform(0, 1)
        pac_speed = -(math.log(-math.log(u))*vfasta) + vfastb
        ccm.run('pacman',1, pac_production_time=pac_speed*.001,
            ghost_production_time=x*.001, user='vfast' )
```
Listing 4: Simulating the PacMan model

11. SOAR PacMan model

11.1. Model Code

```
################################################################################

# From Chapter 9 of Soar 8 Tutorial
#
# These are the final versions of the rules for the generalized
    advanced move
# operator.

sp {initialize*state*directions
   (state <ss> ^type state)
   -->
   (<ss> ^directions <n> <e> <s> <w>)
   (<n> ^value north ^opposite south)
   (<e> ^value east  ^opposite west)
   (<s> ^value south ^opposite north)
   (<w> ^value west  ^opposite east)}

# Propose*move*no-backward:
# If there is normalfood, bonusfood, eater, or empty in an
    adjacent cell,
#    and there is no last direction equal to the opposite
    direction for that #    cell,
#    propose move in the direction of that cell, with the cell's
     content,
#    and indicate that this operator can be selected randomly.

sp {propose*move*no-backward
   (state <s> ^io.input-link.my-location.<dir>.content { <co> <>
        wall }
              ^directions <d>
             -^last-direction <o-dir>)
   (<d> ^value <dir>
        ^opposite <o-dir>)
-->
   (<s> ^operator <o> +, =)
   (<o> ^name move
        ^direction <dir>
        ^content <co>)}

# Apply*move
# If the move operator for a direction is selected,
#    generate an output command to move in that direction.

sp {apply*move
```

```
   (state <s> ^io.output-link <ol>
              ^operator <o>)
   (<o> ^name move
        ^direction <dir>)
-->
   (write | | <dir>)
   (<ol> ^move.direction <dir>)}

# Apply*move*remove-move:
# If the move operator is selected,
#    and there is a completed move command on the output link,
#    then remove that command.

sp {apply*move*remove-move
   (state <s> ^io.output-link <ol>
              ^operator.name move)
   (<ol> ^move <direction>)
   (<direction> ^status complete)
-->
   (<ol> ^move <direction> -)}

# Apply*move*create*last-direction
# If the move operator for a direction is selected,
#    create an augmentation called last-direction with that
    direction.

sp {apply*move*create*last-direction
   (state <s> ^operator <o>)
   (<o> ^name move
        ^direction <direction>)
-->
   (<s> ^last-direction <direction>)}

# Apply*move*remove*last-direction
# If the move operator for a direction is selected,
#    and the last-direction is not equal to that direction,
#    then remove the last-direction.

sp {apply*move*remove*last-direction
   (state <s> ^operator <o>
              ^last-direction <direction>)
   (<o> ^direction <> <direction>
     ^name move)
-->
   (<s> ^last-direction <direction> -)}
```

```
# Select*move*bonusfood-better-than-normalfood
# If there is a proposed operator to move to a cell with
   bonusfood and
#   there is a second proposed operator to move to a cell that
   is empty or
#   has normalfood
#   prefer the first operator.

sp {select*move*bonusfood-better-than-normalfood-empty
   (state <s> ^operator <o1> +
              ^operator <o2> +)
   (<o1> ^name move
         ^content normalfood)
   (<o2> ^name move
         ^content empty)
-->
   (<s> ^operator <o1> > <o2>)}

# Select*move*avoid-empty-eater
# If there is a proposed operator to move to an empty cell or a
   cell with
#   another eater,
#   then avoid that operator.

sp {select*move*avoid-empty-eater
   (state <s> ^operator <o1> +)
   (<o1> ^name move
         ^content << empty eater >>)
-->
   (<s> ^operator <o1> <)}

# Select*move*reject*backward
# If there is a proposed operator to move in the direction
#   opposite the last move,
#   reject that operator.


sp {select*move*reject*backward
   (state <s> ^operator <o> +
              ^directions <d>
              ^last-direction <dir>)
   (<d> ^value <dir>
        ^opposite <o-dir>)
   (<o> ^name move
          ^direction <o-dir>)
-->
   (write | Reject | <o-dir>)
   (<s> ^operator <o> -)}
```

Listing 5: The PacMan model in SOAR

```
###########################################################################

# From Chapter 9 of Soar 8 Tutorial
#
# These are the final versions of the rules for the generalized
    advanced move
# operator.

sp {initialize*state*directions
   (state <ss> ^type state)
   -->
   (<ss> ^directions <n> <e> <s> <w>)
   (<n> ^value north ^opposite south)
   (<e> ^value east  ^opposite west)
   (<s> ^value south ^opposite north)
   (<w> ^value west  ^opposite east)}

# Propose*move*no-backward:
# If there is normalfood, bonusfood, eater, or empty in an
    adjacent cell,
#    and there is no last direction equal to the opposite
    direction for that #    cell,
#    propose move in the direction of that cell, with the cell's
     content,
#    and indicate that this operator can be selected randomly.

sp {propose*move*no-backward
   (state <s> ^io.input-link.my-location.<dir>.content { <co> <>
        wall }
                 ^directions <d>
                -^last-direction <o-dir>)
   (<d> ^value <dir>
        ^opposite <o-dir>)
-->
   (<s> ^operator <o> +, =)
   (<o> ^name move
        ^direction <dir>
        ^content <co>)}
```

```
# Apply*move
# If the move operator for a direction is selected,
#    generate an output command to move in that direction.

sp {apply*move
   (state <s> ^io.output-link <ol>
              ^operator <o>)
   (<o> ^name move
        ^direction <dir>)
-->
   (write | | <dir>)
   (<ol> ^move.direction <dir>)}

# Apply*move*remove-move:
# If the move operator is selected,
#    and there is a completed move command on the output link,
#    then remove that command.

sp {apply*move*remove-move
   (state <s> ^io.output-link <ol>
              ^operator.name move)
   (<ol> ^move <direction>)
   (<direction> ^status complete)
-->
   (<ol> ^move <direction> -)}

# Apply*move*create*last-direction
# If the move operator for a direction is selected,
#    create an augmentation called last-direction with that
   direction.

sp {apply*move*create*last-direction
   (state <s> ^operator <o>)
   (<o> ^name move
        ^direction <direction>)
-->
   (<s> ^last-direction <direction>)}

# Apply*move*remove*last-direction
# If the move operator for a direction is selected,
#    and the last-direction is not equal to that direction,
#    then remove the last-direction.

sp {apply*move*remove*last-direction
   (state <s> ^operator <o>
              ^last-direction <direction>)
   (<o> ^direction <> <direction>
     ^name move)
-->
   (<s> ^last-direction <direction> -)}

# Select*move*eater-better-than-normalfood
# If there is a proposed operator to move to a cell with
    bonusfood and
#    there is a second proposed operator to move to a cell that
    is empty or
#    has normalfood
#    prefer the first operator.

sp {select*move*bonusfood-better-than-normalfood-empty
   (state <s> ^operator <o1> +
              ^operator <o2> +)
   (<o1> ^name move
         ^content eater)
   (<o2> ^name move
         ^content << normalfood empty >>)
-->
   (<s> ^operator <o1> > <o2>)}


# Select*move*reject*backward
# If there is a proposed operator to move in the direction
#    opposite the last move,
#    reject that operator.


sp {select*move*reject*backward
   (state <s> ^operator <o> +
              ^directions <d>
              ^last-direction <dir>)
   (<d> ^value <dir>
        ^opposite <o-dir>)
   (<o> ^name move
        ^direction <o-dir>)
-->
   (write | Reject | <o-dir>)
   (<s> ^operator <o> -)}
```

Listing 6: The Ghost model in SOAR

12. References

M. Ambinder. Valve's approach to playtesting: The application of empiricism.

J. R. Anderson. *Language, Memory and Thought.* Mahwah, NJ: Erlbaum, 1976.

J. R. Anderson. *Rules of the mind*. Erlbaum, Hillsdale, NJ [u.a.], 1993. John R. Anderson.

J. R. Anderson and C. Lebiere. *The atomic components of thought*. Lawrence Erlbaum associates, Mahwah, New Jersey, 1998.

J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological review*, 111(4):1036–60, Oct. 2004. ISSN 0033-295X. doi: 10.1037/0033-295X.111.4.1036. URL http://www.ncbi.nlm.nih.gov/pubmed/15482072.

B. J. Best and C. Lebiere. Cognitive agents interacting in real and virtual worlds. 2006. URL http://books.google.com/books?hl=en&amp;lr=&amp;id=V1RyhTamPkgC&amp;oi=fnd&amp;pg=PA186&amp;dq=Cognitive+Agents+Interacting+in+Real+and+Virtual+Worlds&amp;ots=imCXmLuGHR&amp;sig=W4YLuKkzmrK7Y7TmiCdEI_GGr88.

A. J. Champandard and M. Dawe. Behavior Trees : Three Ways of Cultivating Game AI .

B. S. Dyckhoff, Max. Evolving Halo's Behaviour Tree AI.

G. J. Feist and E. L. Rosenberg. *Psychology: Making Connections*. McGraw-Hill, 2009.

K. Gluck. Cognitive Architectures for Human Factors in Aviation. pages 375–399, 1988.

W. Hangartner, P. Dr, and R. Pfeifer. Navigat-soar - a path-finding agent using soar, 1994.

D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 1987. URL http://www.sciencedirect.com/science/article/pii/0167642387900359.

R. Ierusalimschy. Lua-an extensible extension language. *Software Practice and* . . . , 26(June), 1996. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.3859&rep=rep1&type=pdf`.

T. R. Johnson. Control in act-r and soar, 1997.

J. Laird. The Soar Cognitive Architecture. *The Newsletter of the Society for the Study of Artificial Intelligence and Simulation of Behaviour*, (134), 2012. URL `http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&amp;tid=12784`.

J. E. Laird and A. Newell. Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1987):1–64, 1987. URL `http://www.sciencedirect.com/science/article/pii/0004370287900506`.

J. E. Laird, K. J. Coulter, R. M. Jones, P. G. Kenny, F. Koss, and P. E. Nielsen. Integrating Intelligent Computer Generated Forces in Distributed Simulations : TacAir-Soar in STOW-97.

J. E. Laird, R. M. Jones, and A. Arbor. Building Advanced Autonomous AI Systems for Large Scale Real Time Simulations. 1998.

J. E. Laird, A. Arbor, and J. C. Duchi. Creating Human-like Synthetic Characters with Multiple Skill Levels : A Case Study using the Soar Quakebot. *Methodology*, pages 54–58, 2001.

C. Lebiere and B. J. Best. From Microcognition to Macrocognition. pages 1–29.

C. Lebiere and R. West. A dynamic ACT-R model of simple games. *Proceedings of the Twenty-first Conference of the* . . . , 1999. URL `http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:A+Dynamic+ACT-R+Model+of+Simple+Games#0`.

M. V. Lent and J. E. Laird. Update on the Soar / Games Project What is the Soar / Games Project ? In *Proceedings for the 19th Soar Workshop*, 1999.

M. V. Lent, J. E. Laird, J. Buckman, J. Hartford, S. Houchard, K. Steinkraus, and R. Tedrake. Intelligent Agents in Computer Games. *Artificial Intelligence*, 1998.

C.-u. Lim, R. Baumgarten, and S. Colton. Evolving Behaviour Trees for the Commercial Game DEFCON.

S. Mohan and J. Laird. An Object-Oriented Approach to Reinforcement Learning in an Action Game. *Seventh Artificial Intelligence and Interactive Digital* . . . , pages 164–169, 2011. URL `http://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/download/4069/4430`.

S. Mohan and J. E. Laird. Reinforcement Learning in Infinite Mario.

T. J. Muller, A. Heuvelink, and F. Both. Implementing a Cognitive Model in Soar and ACT-R : A Comparison. *Control*, 2008.

A. Newell. You Can't Play 20 Questions With Nature and Win. *Visual Information Processing*, page 25, 1973.

A. Newell. *Unified theories of cognition*. Harvard University Press, Cambridge, MA, USA, 1990. ISBN 0-674-92099-6.

R. Pillosu. Coordinating Agents with Behaviour Trees, Sept. . ISSN 1540-5834.

S. Pronovost and R. West. Improving Usability and Integration of Human Behavior Representation Engineering across Cognitive Modeling , Human Factors , and Modeling and Simulation Best Practices. (March):21–24, 2010.

F. E. Ritter and D. P. Wallach. Models of Two-person Games in A CT -R and SOAR. *Cognitive Science*, (June), 2001.

F. E. Ritter, N. R. Shadbolt, D. Elliman, R. M. Young, F. Gobet, and G. D. Baxter. Techniques for Modeling Human Performance in Synthetic Environments: A Supplementary Review. Technical report, 2002. URL `http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstract`.

E. Ruskin. Rule Databases for Contextual Dialog and Game Logic. *Game Developers Conference*, 2012.

K. Shah, S. Rajyaguru, R. S. Amant, and F. E. Ritter. Connecting a Cognitive Model to Dynamic Gaming Environments : Visual environments for cognitive modeling. *Information Sciences*, pages 189–194.

T. Stewart. Deconstructing ACT-R. *Proceedings of the Seventh International*, 2006. URL `http://actr.psy.cmu.edu/papers/641/stewartPaper.pdf`.

T. Stewart and R. West. Cognitive redeployment in ACT-R: Salience, vision, and memory. *8th International Conference on Cognitive ...*, 2007. URL `http://bicasociety.org/models/sites/default/files/2007-CognitiveRedeploymentACT-R.pdf`.

E. Syriani and H. Vangheluwe. Programmed Graph Rewriting with Time for Simulation-Based Design. *Proceedings of the 1st international conference on Theory and Practice of Model Transformation*, pages 91–106, 2008.

E. Syriani and H. Vangheluwe. De-/re-constructing model transformation languages. *Electronic Communications of the ...*, pages 1–20, 2010. URL `http://journal.ub.tu-berlin.de/index.php/eceasst/article/view/407`.

R. West, C. Lebiere, and D. Bothell. Cognitive architectures, game playing, and human evolution. *... cognitive modeling to social ...*, (2), 2006. URL `http://books.google.com/books?hl=en&lr=&id=V1RyhTamPkgC&oi=fnd&pg=PA103&dq=Cognitive+Architectures,+Game+Playing,+and+Human+Evolution&ots=imC1oQwECT&sig=2QnwKlggcAoA3KJSzPK2RPMtHSk`.