J. EXPT. THEOR. ARTIF. INTELL. 12(2000)191–212

Engineering characteristics of autonomous agent architectures

PAUL SCERRI† and NANCY REED‡

Real-time Systems Laboratory, Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden
†email: pausc@ida.liu.se
‡email: nanre@ida.liu.se

Abstract. As the science of building agents and agent based applications improves, agents are gradually making the transition from research laboratory prototypes to industrial applications. In an industrial setting pragmatic engineering issues related to the development and maintenance of agents come to the fore. In this paper a list of important engineering characteristics of agent architectures is presented. The list has the dual aims of providing evaluation criteria for agent users and design issues for agent architecture designers. Evaluation of an architecture with respect to the presented characteristics will allow industrial project managers to better assess the trade offs between architectures. Designing architectures that better fulfill the characteristics should lead to more industrially relevant architectures and successful agent applications. This list is designed to start discussion and raise awareness of industrial issues. It is hoped that over time good metrics and an evaluation methodology evolve which provide a robust framework for evaluating agent architectures with respect to industrial concerns.

Keywords: agent architectures, designing architectures, chicken factory.

1. Introduction

Intelligent autonomous agents are increasingly used in applications in many diverse areas. As autonomous pieces of software acting towards internal goal(s) within an environment, agents offer great promise as a means of increasing the utility of a complex system while at the same time reducing its complexity. Agents often encapsulate chunks of artificial intelligence within some larger application, allowing the development of intelligent, flexible applications from existing applications, as well as the development of new applications. As the science of building agents and agent based applications improves, agents are gradually making the transition from research laboratory prototypes to industrial applications. Successful application areas include control of complex systems, e.g. ARCHON (Jennings 1995, Cockburn and Jennings 1996) through to more abstract tasks such as scheduling or routing, e.g. (Chaib-draa 1997). Furthermore the applications to which agents are being applied are larger and

Journal of Experimental & Theoretical Artificial Intelligence ISSN 0952-831X print/ISSN 1362-3079 online © 2000 Taylor & Francis Ltd
http://www.tandf.co.uk/journals

more ambitious than ever before, e.g. the Remote Agent project (Pell *et al.* 1998) in which agent software controlled a NASA spacecraft.

Because of the sheer size and complexity of industrial applications, as agents make the transition between research and industry, it is necessary to look at agents with an 'engineering' agenda as well as a 'research' agenda (Bussman 1998). Any migration of a technology from research to industry requires consideration of aspects of the technology that allow it to be viable in an industrial setting (Wooldridge and Jennings 1998). Agents are no exception. Hence, to show industrial applicability of agent-based technology, it is not sufficient to show that particular agent behaviour can be produced, it is also necessary to be able to estimate the time and effort involved in creating the agents, the development process to be followed, the maintainability of the agents and so on—i.e. pragmatic engineering issues. Clearly engineering and scientific concerns overlap, however, there are some engineering concerns that are not scientific agendas. This paper is concerned specifically with engineering concerns.

Computer science, like all sciences, is concerned with experimentation, finding properties and developing theories about the world (Chalmers 1982). Engineering, on the other hand, is about following a reliable method to develop products meeting defined requirements within temporal, financial, risk, etc. constraints (de Chamapeaux 1997). To see the difference between scientific and engineering concerns consider a Mars exploration robot and a robotics lab exploration robot. Although the functionality of the final robots is fairly similar the requirements on aspects of their development are vastly different, e.g. it must be almost certain that the Mars robot will work the first time whereas it is acceptable (maybe even expected!) to have the lab robot fail in its early trials. One believes that in order for agent technology to make the transition from AI buzzword to industrial practice, engineering aspects of developing agents and agent-based systems need to be addressed (see figure 1).

When a development project considers using agent technology for some purpose, 'agent architectures' are evaluated against their requirements. An agent architecture includes the philosophy, methodology, code library, user interfaces, documentation, etc. that forms part of the architecture's particular way of building agents. All of the elements that make up each agent architecture may be important factors in any decision to use a particular type of agent. When specifically considering agents for use in industry it is crucial to consider the engineering aspects of the agent architecture as a whole.

This paper presents a list of engineering characteristics and argues that agent architectures should be evaluated with respect to the characteristics. The creation of such a list has two purposes. First, an evaluation using the list would leave project managers in a better position to evaluate the trade-offs among architectures and determine the best agent architecture for their particular projects. Second, the

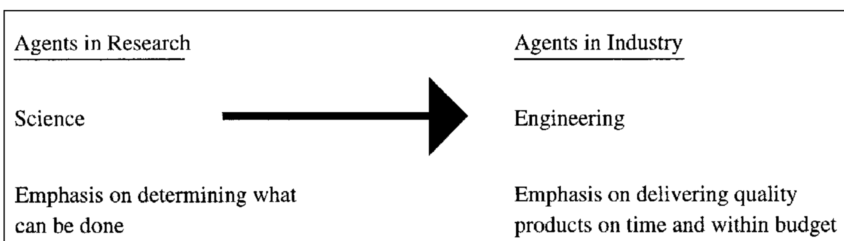

Figure 1. Taking agents from research to industry.

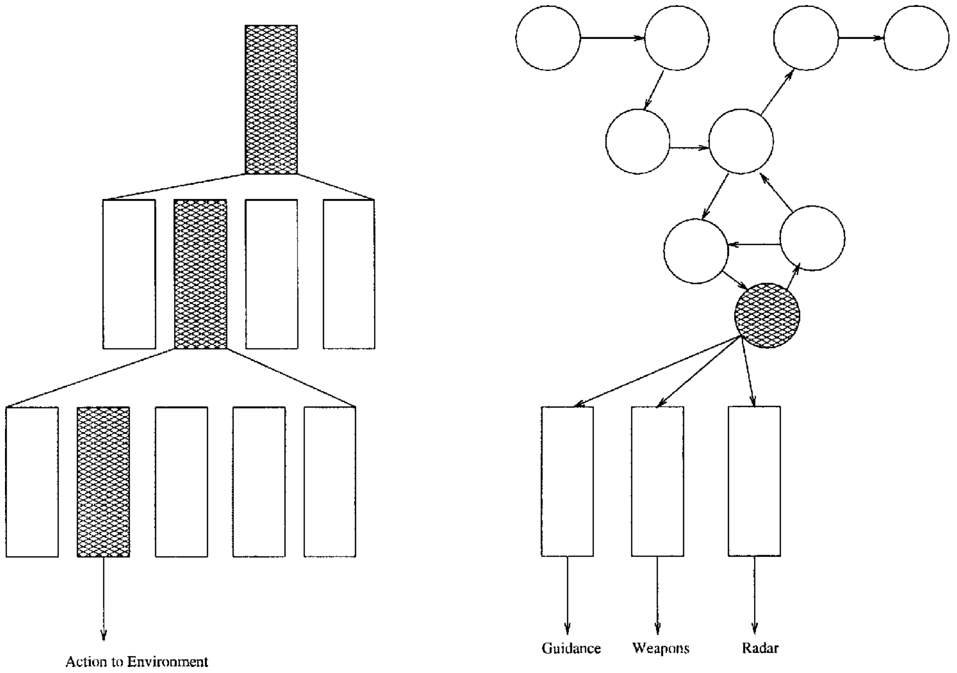

Figure 2. The structure of the two architectures. The Chicken Factory architecture structure is shown on the left. Behaviours are layered and one behaviour on each level is able to act by setting behaviours on the next level down. The shaded boxes represent selected behaviours, unshaded boxes represent other behaviours. The diagram on the right shows the TACSI agent architecture. Circles represent states and the shaded circle represents the current state. In the current state three behaviours are activated, one for each of the agent's resources.

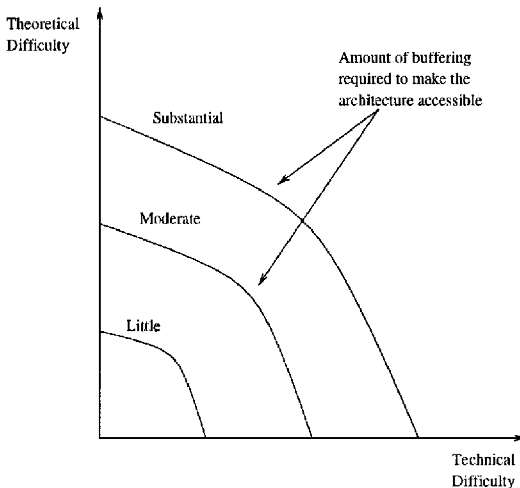

Figure 3. Relationship between technical and theoretical accessibility and the amount of buffering required to make the architecture accessible.

characteristics provide aims towards which agent researchers can work if it is intended that their architecture will be used in industry. The proposed list of characteristics is:

- Accessibility: how easy is the architectures for novices:
- Representational power: how powerful is the specification method?
- Reuse: how much of one agent can be reused in another agent?
- Generality: for what range of applications can the architecture be used?
- Project methodology: what process if followed to create agents with this architecture?
- Correctness measures: what facilities are available for ascertaining that the agent works properly?
- Computational requirements: how much hardware does the agent need?

The objective of this article is not to provide a definitive list of characteristics and metrics by which all agent architectures should be measured. Rather to raise awareness of industrial issues for agent architectures and begin the process of developing a framework within which architectures can be compared. Wherever possible metrics or methods for evaluating with respect to different characteristics have been proposed. However, clearly, a wide variety of experience and research will need to be brought to bear before a reliable, robust method for evaluating architectures for industrial use is available.

Many of the characteristics may be difficult or impossible to quantify. Although as scientists one would prefer objective, quantitative evaluations; years of largely unsuccessful work aimed at developing such metrics for programming languages suggests that quantitative metrics will be hard to come by. The approach below proposes non-quantitative characteristics, supplemented with an outline of the type of analysis that could be performed in evaluating the characteristic. Running examples of two agent architectures are used below to show how the measures can provide useful objective information which can improve a project manager's ability to make informed decisions about which agent architecture best suits their purposes.

The specific engineering characteristics one describes were identified while attempting to integrate agent technology into an existing aircraft simulation environment developed by Saab (1998). A critical aspect of that work was that the developers were not necessarily interested in developing agents per se, but rather in creating realistic simulated pilots to increase the utility of their simulator. It was found that when agents were to be used in industrial settings, requirements on the engineering aspects of the agents came to the fore. It is for those additional requirements that the characteristics above provide evaluation of an architecture's suitability.

2. Terminology

There is no standard terminology for describing agent architectures and the development process for agents. This section defines some useful terms as they are used in the rest of the paper.

An 'agent specification' is the definition an agent designer creates within an architecture to determine the precise behaviour of an agent. The form the specification takes, and the process of creating it vary from architecture to architecture. In dMars (Kinny 1993) for example, a specification is a set of partial plans and agent definitions created graphically whereas in some behaviour-based architectures, a specification is a set of behaviours (possibly coded in C).

A 'specification element' is one of the parts that makes up a complete specification. Depending on the architecture, a specification element may be a behaviour, a plan, a rule, a plan primitive, etc. A specification may contain several different types of specification elements.

The 'capabilities' of an agent are the external, observed abilities of an agent or architecture. When referring to the capabilities of agent architectures one generally talks about abstract capabilities such as 'social awareness' or 'reactivity', but when one talks about the capabilities of a particular agent, the terms are usually more concrete, for example 'will find the best price on all web sites' or 'can avoid unexpected obstacles in its path'.

3. Example architectures

This section describes two agent architectures that will be used throughout the rest of this article to illustrate the engineering characteristics and evaluation methodology/metrics proposed. The first, the Chicken Factory, was developed completely in academia but is intended to have industrial applications for creating agents for simulation environments. The second, has been developed completely in industry for use in one particular simulation environment. It is intended that evaluating the two architectures side by side will shed some light on the utility of the engineering characteristics described.

3.1. *The chicken factory*

The first example architecture is a layered behaviour-based architecture (Scerri *et al.* 1998, Scerri and Ydren 1999) versions of which have been entered in three RoboCup World Cup soccer competitions (Kitano *et al.* 1997).

The architecture includes a number of development and debugging interfaces allowing completely graphical development of the behaviour-based agents. Originally it was planned that non-expert end users would be able to create agents with the Chicken Factory but it has become clear that some agent experience is required to use effectively the architecture's development system.

Behaviour-based agents have their overall functionality divided into 'behaviours'. A behaviour encapsulates a single objective of the agent such as 'obstacle avoidance' or 'scoring a goal'. At each point in time the agent will have some number of concurrently active behaviours. A variety of different mechanisms may be used to combine the output of the currently active behaviours into a single agent action. The mechanism may, for example, select one of the behaviours and use the selected behaviour output as the overall output, alternatively the mechanism may somehow 'fuse' the outputs of all the behaviours into a single action. The overall behaviour of the agent is a complex result of the interactions among the behaviours and the world. Conceptually at least, there is no central decision-making process or central world model. For a more detailed discussion of behaviour-based agents see Mataric (1992).

The Chicken Factory behaviour-based architecture creates agents with several layers. The action of higher layers is to set new behaviours at lower layers. The bottom level behaviours interact directly with the world. Higher level behaviours are intended to be more abstract and lower level behaviours more focussed. The layers act asynchronously with respect to one another, i.e. an upper layer can activate new lower level behaviours at any time.

In the Chicken Factory, for each layer of behaviours there is a mediator, which selects one behaviour whose output will be the action for that level. The mediator chooses the active behaviour with the highest 'activation level'. The activation level of a behaviour is a function of the value of the behaviour's 'activation predicate' and the activation level of the behaviour at the previous step. A predicate is a function that maps sensor information to a value indicating the truth of a statement. For example a predicate 'near ball' has a higher value when a RoboCup agent senses the ball is close than when it senses the ball is far away. The predicates are predefined, hard coded into the system and are often parameterized.

To facilitate end-user development there is a completely graphical tool for specifying agents. Low-level skills are created in a low-level language and hard coded into the tool. An end user specifies agents in a bottom-up manner by creating low-level behaviours whose actions are to turn skills on or off. Higher level behaviours are then specified, their actions are groups of lower level behaviours which will be activated.- The end user selects an activation predicate in an attempt to make the mediator select behaviours at appropriate times that will achieve the desired results. At any point in time a partially specified agent is capable of being tested—allowing for rapid, incremental development.

To facilitate fast, effective testing and debugging, the Chicken Factory architecture includes a number of runtime interfaces. One interface displays the current values of the predicates in real-time, allowing the designer to check that the agent is interpreting its environment correctly. Another interface shows an image of the state of the environment as the agent understands it. The final, most important interface, shows the activation levels of all active behaviours in real-time. This interface allows the developer to watch the 'reasoning' of the agent, i.e. see which behaviours are active and which are being selected.

3.2. *The TACSI agent architecture*

TACSI (TACtical SImulation) is the name of a high fidelity flight simulator developed at Saab AB for testing aircraft systems and military tactics, as well as man-in-the-loop training (Andersson 1995, Coradeschi 1997, Saab 1998). Agents can control simulated pilots within the simulator. The agent architecture has two layers, a hierarchical state machine layer and a behaviour layer. The agent architecture has been incrementally developed over a long period of time by a number of different development teams to meet very specific requirements.

A hierarchical state machine controls the high-level behaviour of the agent. At any point in time the agent is in 'one' state. Transitions can be defined to child, parent or sibling states in the hierarchy. A number of transitions can be defined for a single state to other states. If more than one transition is applicable, a user defined priority ordering is used to determine which is taken.

In each state there is a separate 'block' for each of the simulated aircraft's resources, i.e. guidance, radar and weapons, containing the rules that determine usage of that resource in that state. The rules are ordered by a static, user-defined priority. A rule consists of one or more preconditions connected by a boolean 'and' and one or more behaviours to be turned on when the rule is activated. The highest priority applicable rule fires and its consequent behaviour(s) is turned on. Behaviours are built into the architecture at a low level and are usually fairly low-level air combat specific control routines such as maintaining a heading from another aircraft or setting radar to 'sweep mode'.

In TACSI, agents can be specified and tested using a variety of very easy to use graphical user interfaces. A large number of parameterized preconditions and behaviours are built into the system. To create more preconditions or behaviours requires modifications to the low-level code. New states are created by selecting an appropriate menu item. Rules for the state, both delta rules (i.e. transition rules) and rules for selecting behaviours are defined with a simple table based specification system. Preconditions are listed along the left and behaviours/states at the bottom. The user marks check boxes next to preconditions (instantiating parameters via a popup window if required) as either 'true', 'false' or 'don't care' then selects the appropriate new state or resultant behaviour at the bottom. The preconditions are joined by an implicit 'and', e.g. if the precondition 'main target in firing range' is marked 'true' and 'fuel left $> 50\%$' is marked 'false' the total precondition for the rule is 'main target in firing range and not fuel left $> 50\%$'.

At run-time a number of interfaces are used to show the rules that are triggered in each of the agents. Additionally a large amount of information about the state of the agent's aircraft (and hence the information the agent has to work with) can be displayed and updated in real-time. The rule firings are also saved in a file so that they can be analysed after a simulation run. All parts of the agent's decision making are deterministic so every run of a scenario results in identical results.

4. Engineering characteristics

In the following sub-sections each of the important engineering characteristics on our list is considered in turn. For each characteristic a description is given of the characteristic and a justification supplied for its inclusion. Then example evaluations of the characteristics on the two running examples, the Chicken Factory architecture and the TACSI architecture are given.

4.1. *Accessibility*

4.1.1. *Characteristic 1.* The accessibility of an agent architecture is defined as the amount of knowledge a novice to the architecture needs in order to become proficient at creating agents with the architecture.

As agent technology makes the transition from research to industry, applications are encountered where agents are not the central focus of the application. For example in computer games (e.g. AURAN 1999) or interactive simulations, agents are simply another tool developers use in order to meet their requirements. An important implication of this is that agent experts will not always be involved in a project where agents are used. The lack of agent knowledge implies that the agent technology will sometimes need to be usable by non-agent experts in order to be useful (Schoepke 1999).

The smaller the amount of knowledge required to effectively use the architecture, the more accessible the architecture is. The knowledge required can be of a 'technical' nature and/or of a 'theoretical' nature. Technical knowledge includes skills like programming, control theory or mathematics. The amount of technical knowledge required to use an architecture ranges from advanced development skills to basic computer literacy. Theoretical knowledge includes background knowledge such as an understanding of utility theory, cognitive models or psychology. The required

theoretical knowledge ranges from a solid working knowledge of some cognitive model (as advocated by the Defense Modelling and Simulation Office (Pew and Mavor 1998) to little or no background knowledge.

An agent architecture is accessible if it requires little knowledge to become proficient with the architecture, the complexity of the architecture is 'buffered' from the user or a combination of both. An architecture is accessible if there is little knowledge required to understand the system, i.e. the architecture is technically straightforward and is based on simple theories. Alternatively an agent architecture is more accessible if it is not necessary for the agent developer to understand the agent architecture in order to create useful agents. In other words some buffer exists between the architecture and the user which means the user does not have to deal with the internal workings of the architecture in order to use it. A US Defense Modeling and Simulation Office (DMSO) report (Defense Modelling and Simulation Office 1993) suggests that buffered accessibility often equates to abstraction.

4.1.2. *Examples*. The Chicken Factory architecture has relatively high technical accessibility. Because the architecture includes support for completely graphical development on top of a set of prebuilt skills, little programming knowledge is required. Some technical skill may be required to develop a maintainable, extendible organization of behaviours, however any intuitive structuring is probably reasonable (Bryson 1998). Hence it can be said that the technical aspects of the architecture are accessible due to a graphical interface buffer between the system and the user.

On the other hand because an understanding of the concepts of emergent behaviour and distributed control are non-intuitive but necessary to create agents with the architecture, one can say that the architecture has low theoretical accessibility. There is no central control to a behaviour-based system and the overall behaviour is a result of the agent's interactions with the world, an idea often unfamiliar to those used to working with computers. Designing behaviours and assigning appropriate predicates to achieve the correct resulting behaviour in complex environments requires a somewhat novel way of thinking about complex behaviour. Some theoretical knowledge is required to find behaviour activation parameters that achieve desired results, although the debugging interfaces make the task more manageable.

The TASCI architecture has very high technical accessibility. All that is required of a user is the ability to select check boxes and occasionally type the value of a parameter in a popup window. The predefined behaviours are at a relatively high level of abstraction resulting in fairly large grained rules which can be easily designed. The theoretical accessibility, especially for the target user, is also very high. The idea of a state machine is quite simple to understand especially for the engineers and pilots who will be using the system. The state machine concept is directly represented in the specification interfaces.

This puts the TACSI system in the category of technically accessible systems due to its understandable underlying architecture. The Chicken Factory architecture is also relatively simple to use and technically accessible, but in contrast this is due to a buffer between the architecture and user. The TACSI architecture requires virtually no theoretical knowledge while the Chicken Factory requires substantial theoretical knowledge. The trade-offs made in TACSI to develop such an easy to use system result in substantially less generality than for the Chicken Factory, but this is because the TACSI system was purposely specialized for its application.

4.2. *Representational power*

4.2.1 *Characteristic 2.* The representational power of an architecture is defined as the ratio of distinct reactions to specification elements.

In theory, it may be possible to create interesting agents by hand-coding a long if-statement in C which determines how an agent should react to different situations. In practice however, this way of specifying complex agents is far too cumbersome and error prone. All modern agent architectures provide more powerful and concise mechanisms for a designer to use to specify the behaviour of an agent. The precise amount of specification required, however, to achieve a particular desired behaviour varies from architecture to architecture. The amount of specification required to achieve a certain desired behaviour is referred to as the 'representational power' of the architecture.

The 'distinct reactions' of an agent are defined as the number of situations for which the agent will follow a distinct line of reasoning. A 'situation' may include not only the observed current state of the world but also previous states and any commitments, plans etc. that could affect a decision. The number of different situations the agent needs to handle differently in order to produce the desired behaviour is application dependent. Generally, agents appearing more intelligent will handle more situations with distinct reactions. The number of distinct reactions gives some feel for the complexity of the final agent, i.e. the more distinct reactions the more complex the agent.

The number of 'specification elements' for an agent is defined as the number of pieces that go into making up a specification. A specification element for an architecture will be the smallest meaningful piece of a specification. For example in a behaviour-based architecture a behaviour is a specification element and in a state-machine a state is a specification element. The number of specification elements gives an indication of the size of a specification, the more elements the bigger the specification.

Although the actual ratio calculated for the representational power may not provide too much information, its derivation gives an indication of how specification elements are combined by the agent for different decisions. Hence it is as important when evaluating an architecture to describe how the number was obtained, as it is to present the number. The ratio is not biased against architectures that have more reasoning (because the number of distinct reactions increases), nor is it biased against more complex agents (the metric measures the ratio of action to specification). What the ratio does show is how flexibly and powerfully the architecture combines specification elements. This in turn gives an indication of how much effort is required to achieve required behaviour of an agent.

Achieving the required agent behaviour with minimal effort can be supported by an architecture in a variety of ways. The simplest way is to have knowledge built into the agent architecture either in the form of reusable libraries or hard-coded into the architecture itself. The representational power of an architecture will also be high if the architecture has flexible, powerful ways of manipulating a small specification to handle a range of situations. Alternatively there may be a specification system which is very powerful, allowing a range of behaviours to be specified easily (before, perhaps being translated into some other format).

If the representational power of the architecture is low, the architecture is unlikely to be appropriate for building agents with a wide range of behaviour. Building very complex agents within architectures with low representational power will require a

very large number of specification elements and is hence likely to be difficult and time consuming, if feasible at all.

4.2.2. *Examples*. For the Chicken Factory architecture a specification element is a behaviour. If one ignores the temporal impact on the behaviour activation levels for the Chicken Factory architecture then this architecture has very low representational power. For some distinct reactions there may be more than one behaviour created specifically for that reaction. On the other hand, low-level behaviours may be used in a number of different high-level behaviours, possibly with different activation parameters or predicates, hence giving more distinct reactions without more specification elements.

The temporal aspect of the activation level of a behaviour (the previous activation level is one of the factors affecting the current activation level) means that there are actually more distinct reactions than stated above, because to some extent the decision made by the agent depends on previous states of its reasoning, i.e. the same world state may result in attacking or defending, depending on the previous reasoning state.

Overall, the behaviour-based architecture described does not have high representational power. The ratio of specification elements, i.e. behaviours, to distinct lines of reasoning is relatively close to one. This implies that for agents requiring a very broad range of behaviours, the Chicken Factory is probably not the best architecture.

For TACSI, the number of specification elements is the number of states plus the number of rules, each specifying the behaviour to take given some preconditions. The number of distinct reactions is also equal to the number of states plus the number of rules. One might perhaps argue that the number of distinct reactions is actually larger because a state could be reached via a number of different paths, i.e. the agent can be in the same state for different reasons and each different reason represents a distinct reaction. However the philosophy of the TACSI state machines is that in a particular state the agent is handling a particular situation. Therefore each state represents one distinct reaction of the agent. A behaviour, i.e. the consequent of a rule will produce a continuous range of different values depending on the state of the world, e.g. the thrust sent to the engine as a part of maintaining contact with a wingman will vary depending on the relative position of the wingman. However one does not consider all the possible output values as distinct reactions, rather the whole behaviour as a single distinct reaction. The rationale is that the 'reaction' of the agent in this case is the function, rather than the output of the function.

TACSI adopts a different method for achieving high representational power, however. The built-in behaviours and pre-conditions mean that complex agents can be built with only a few specification elements. TACSI has solved the representational power problem by building in the knowledge. Thus although the representation power ratio is fairly close to one, little effort is required to create complex agents (provided appropriate behaviours and preconditions exist).

4.3. *Reuse*

4.3.1. *Characteristic 3.* Reuse in terms of agent architectures refers to the process of taking part of a specification created for one agent and using it in another agent.

The ability to effectively reuse code is a driving force behind the design of modern programming languages and development processes. The emphasis on reuse is even more evident in industrial settings where reuse can substantially affect development and testing costs (Booch 1994). Accordingly, one would expect that agent architectures

will need to support good levels of reuse in order to achieve widespread industrial acceptance.

Reuse may be both in the form of agent specifications, such as specific states, behaviours etc. and in the form of the architectural framework, e.g. the state machine run-time engine, development environment, etc. Both types of reuse are important. It is becoming more common that agent frameworks are reused but it is less common that parts of agent specifications are reused.

Lorenz and Kidd (1994) distinguishes two different types of reuse: white box and black box. White box reuse is where specification elements are copied and pasted into new specifications from existing specifications. Black box reuse is the direct referencing of existing code in a new application. Clearly black box reuse is more desirable as changes need only be made in one place thus reducing errors and maintenance costs, however even white box reuse is useful in reducing development time.

For application areas where a large number of agents need to be created or where there are a number of similar agents, reuse is especially important. Application areas with such properties include computer games, military simulations and disaster management simulations. In application areas such as manufacturing management where testing is critical and expensive, reuse of agent specification elements is also important.

Lorenz and Kidd (1994) describes an object-oriented metric for measuring black box reuse in terms of the number of references to a class both from within the one application and from other applications. By substituting 'specification element' for 'class' a rough measure of agent architecture reuse is found. I.e. in how many distinct reactions in a single agent and in how many different agents is a particular specification element used. White box reuse can similarly be thought of as the number of specification elements copied and pasted into a new specification with little or no editing.

As with any numerical metric related to software development a number of factors need to be taken into account before any meaning can be ascribed to the number produced. In the case of agent specifications there are (at least) four important, interesting factors that impact the possible reuse: the difference between application areas, the granularity of the specification elements being reused, the impact of the quality of the design and the abstraction level of the specification elements being reused.

One would expect that reuse would be lower in cases where the source and destinations applications were more disparate. A large amount of reuse between very different applications or a low amount of reuse between similar applications is more significant than high reuse between similar applications or low reuse between different applications.

The granularity of a specification element is the amount of functionality it encompasses. In a simulation domain, moving a group of agents to a new position is at a low level of granularity while refuelling a vehicle is at a high level of granularity. The level of reuse is likely to vary with granularity, e.g. for a particular architecture it may be easy to reuse low granularity specification elements in new higher level strategies but difficult to reuse higher level strategies with different low level implementations.

Clearly even within the same architecture there will be specifications that result in more or less reuse. The difference in the amount of reuse may, at least in part, be due to the 'quality' of the particular specification. In object-oriented programming, it is

believed that a good software engineer is able to design classes leading to more generality and hence more reuse (Booch 1994).

Another important factor related to reuse is the abstraction level of reused elements. The abstraction of a specification element is how directly the specification element(s) affect the behaviour of the agent. Things like teamwork (e.g. STEAM (Tambe 1997)), aggression and cooperation are very abstract concepts. The abstract concepts influence overall behaviour rather than control it. Reuse of abstract elements is likely to be very useful as there are probably more opportunities for reuse.

All the different aspects of reuse affect the overall reuse of the architecture. The importance of each of the aspects will be application specific. For example for specialized embedded applications reuse across applications is unlikely to be an important factor while for simulation environments it is. Therefore for a specific application the project manager will need to look at all the aspects of reuse and decide on their relative importance in order to determine the overall reuse factor for their application. Figure 4 summarizes this idea.

4.3.2. *Examples*. The Chicken Factory architecture has fairly high amounts of reuse within the same application. Experience in the RoboCup domain showed that a relatively small number of low-level behaviours could be combined by a designer in different ways in high level behaviours, with different parameters and predicates to create a wide range of overall behaviours. As the granularity of the behaviours increases, the amount of black box reuse decreases while the amount of white box reuse increases. For example in the RoboCup 99 competition, approximately 60 behaviours were used. Some behaviours were reused up to 20 times in a single player, and were reused for each of the 11 players in a team, and also across many team designs. Most of the reuse was of behaviours at a low level of abstraction.

Across application areas it is expected there will be very little, if any, reuse. The environmental dependency of behaviours within any behaviour-based system is likely to make it difficult to reuse behaviours across domains. As abstract concepts such as teamwork are not possible (or are at least very difficult) to represent explicitly within the layered behaviour-based framework. As they are not present, they cannot be reused.

Lack of reuse is acknowledged as one of the limitations of the current TACSI agent architecture. System supported reuse is limited to the copying of rules and states. Copying of single rules is white box reuse. Black box reuse is fairly low. A hierarchy of states can be copied (via indirect means) to another agent specification but that is fairly cumbersome, mainly because an agent is not simultaneously in the parent and

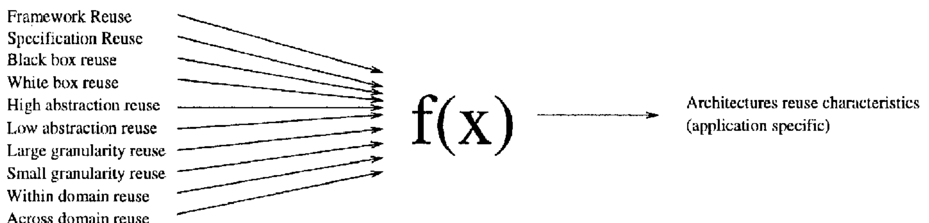

Figure 4. The factors that influence overall reuse. The function *f* will be application specific, i.e. it will reflect the relative importance of each of the factors to the particular application.

child state, instead it is in either the parent or the child. Hence from a child state it must explicitly make the transition up to the parent state before a sibling of the parent can be entered. The result is that when reusing a hierarchy of states the transitions into and out of the hierarchy need to be explicitly taken care of as well.

The TACSI environment is very application specific meaning that there is little potential for reuse across applications. Generally single rules are the only reused elements, hence both the abstraction level and granularity of reuse are low. A state breakdown where the number of transitions between parent and child states was strictly limited would improve the amount of potential black box reuse. In other words the quality of the design will effect the amount of reuse.

4.4. *Generality*

4.4.1. *Characteristic 4.* The generality of an agent architecture is defined as the set of applications for which the agent architecture can reasonably be used to create useful agents (see Figure 5).

The high cost of training and the time involved in learning a new agent architecture means it makes good sense for software developers to learn architectures that can be used across a variety of different projects (Pew and Mavor 1998, p. 337). One only needs to look at the use of programming languages to see that developers will sometimes pass over languages with advantages for a specific task in order to use languages that can be used for a variety of problems.

Clearly the set of applications for which an architecture can be used as impossible to enumerate. However the generality of the architecture can be defined by a list of properties that an application must have, or not have, in order that the architecture can be used to create useful agents. The properties listed need to include the assumptions that the architecture makes about the environment and the types of actions that will/will not be required of an agent. Creating such a list of properties for a particular architecture to be useful is far from trivial. Wooldridge and Jennings (1998) warn that most agent architecture designers over estimate the generality of their architectures. A careful examination of the architecture's assumptions and an honest appraisal of its usefulness is needed to give software developers a good idea of the generality of the architecture.

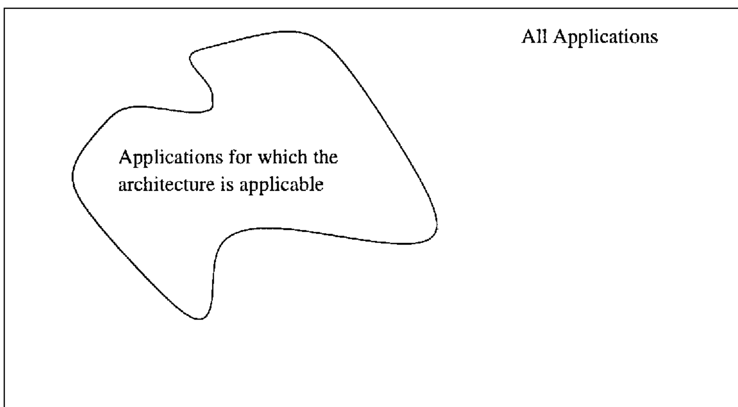

Figure 5. The generality of an agent architecture.

4.4.2. *Examples*. The Chicken Factory architecture described above is designed for fairly dynamic environments where the immediate reactions of the agents are more important than careful long-term planning. The architecture makes few assumptions about the environment, except that it can be sensed often with limited, if any, cost. The sensing may be incomplete and/or uncertain. Although plan-like behaviour can be hand coded in by a designer, the process is cumbersome, hence the architecture should not be used for applications where significant amounts of forward planning are required. In general the Chicken Factory architecture will be more appropriate in complex, dynamic domains where agents change behaviour often and the lack of genuine cognitive intelligence will not be so apparent.

The generality of the TACSI architecture is very low. It is an architecture specialized for a particular application. The interface is carefully tailored to creating the exact type of missions that are usually required for the simulator. Although the Chicken Factory architecture was somewhat tailored to the RoboCup application, it is far less tailored than TACSI is.

The TACSI architecture makes several assumptions. For example, the architecture requires that the resources of the agent can be treated independently at a fairly high granularity. Furthermore it assumes that it is sufficient to attend to one goal at a time. The domains for which the architecture will be appropriate will be relatively simple and only moderately dynamic.

In tandem with the accessibility criteria the generality characteristic shows a distinct difference in the academically developed architecture and the industrially developed one. The Chicken Factory aimed for (and achieved) more generality than the TACSI architecture but fulfilled that goal only at the expense of accessibility. For the developers of TACSI, the ability for the architecture to be reused in other projects was not as important as creating an interface usable by non-computer experts. The project was big and important enough, and the application simple enough, to justify the development of an agent architecture specifically tailored to the application's precise needs. It may be interesting for agent architecture developers to see that for a sufficiently sized application they may be justified in developing a very specific agent architecture.

4.5. *Agent development process*

4.5.1. *Characteristic 5.* The agent development process is the process that developers follow from receiving requirements on agent behaviour to releasing completed agents.

All good system development efforts are carefully planned and follow well established procedures. The procedures are aimed at delivering a quality product on time and within budget. As systems grow larger and more complex, the importance of having a good development process increases further. Planning for a project takes into account many factors including an estimation of the time and costs involved, setting of project milestones, assessment of risks to the project's success and establishment of criteria for success. If agents are to become a core technology, the agent architectures need to support planning and development methodologies or at the very least, be easily integratable into widely used development methods.

The methodology of building agents has received a considerable amount of interest recently in the literature. Bryson (1998), among others, argues for the need for development methodologies. Wooldridge and Jennings (1998) remind developers to keep in mind that they are developing software that should be subject to the same development practices as any other software. Kinny *et al.* (1996) is one of many papers

describing a methodology for building successful agent systems with a particular architecture.

The term 'development process' covers a plethora of different issues. Three in particular are most critical when using agent architectures in industrial projects. First and foremost it is necessary to know 'which' development methodologies can effectively be used to develop agents with the architecture. Second, it is important to know whether the architecture can scale to the size of development the project will require. Third, for purposes of planning the project, one needs to know information like the expected development time, cost and associated risk, and the reliability of this information. Different projects will have different requirements on the development process, no one process will be the optimal for all projects. Hence an evaluation of an agent architecture with respect to a development process should describe a variety of factors and allow a project manager to assess the architecture according to the particular requirements of the domain.

In industrial applications, agents are not developed for their own sake, rather they are part of some larger application. It is essential that the development process for the agents is integrated with the development process of the rest of the application. It could be that an architecture includes a development methodology (as for example dMars (Kinny and Georgeff 1996)) or endorses a particular way of going about the agent construction process. If so, it may be a standard software development methodology or it may be specialized for the architecture requiring additional training for project managers. As the agent development will usually be part of a larger development process it is also important to look at the feasibility of successfully and efficiently developing agents with standard development methodologies. For example, quite often agent architectures seem to encourage a bottom up development methodology of building and testing simple parts then gradually adding functionality. The bottom up approach contrasts with the usual way large scale software development is approached, i.e. a top down specification that has its details gradually filled in.

An important project management factor that becomes relevant when an architecture is taken from the relatively small projects of the laboratory into larger industry projects is the scalability both of the architecture itself and of any development methodology it supports. The architecture and its development process need to scale with an increasing number and complexity of agents, number of developers, expected agent lifetime, etc. For example, behaviour-based systems and rapid incremental prototyping may be an acceptable development methodology for building intelligent robots for walking around a laboratory but may not be acceptable for developing autonomous spacecraft, although the required functionality of the respective agents may have much in common.

Project managers usually plan according to estimates of the time and effort required to achieve different phases of the overall project. The progress of a project is tracked via milestones that indicate the end of one stage of development and herald the start of the next. To support this type of planning process project managers need to be able to estimate the resources required to create agents meeting their requirements. The accuracy of the predictions can impact heavily on the outcome of the project. Some agent architectures are more predictable than others with respect to the time taken to develop agents. Agent architectures also vary in how difficult it is to assess whether a certain milestone has been reached, for example having a complete design. In the commercial world it is important to accurately assess risks to a project so that the

project can be terminated when it becomes unlikely that some threshold economic benefit will be achieved. With respect to agent architectures, risk assessment involves being able to predict whether pursuing some line of work will result in agents with the desired functionality both before development starts and during development.

4.5.2. *Examples*. Most behaviour-based architectures, including the Chicken Factory, are designed for developing agents via an iterative cycle of specifying, testing and debugging the behaviours. Simple behaviours are created and tested in the target environment then more complex behaviours are created by combining the simpler behaviours or adding new behaviours. It may be possible to develop behaviour-based agents in a top down manner by creating the top of the hierarchy first and gradually filling it in (as for example suggested in (Pirjanin 1998) but this is not standard practice (Bryson 1998). Regardless of the development method, it is not possible to test, or effectively develop low level behaviours for a system of this type without the target environment being available.

The Chicken Factory architecture allows an expert programmer to define new low level skills for the agent as required. This mechanism means that a development process is unlikely to fail due to low level difficulties, however if it is found that the agents are not intelligent enough there may be little that can be done to improve them. It is still contested exactly what behaviour-based agents can be designed to do, hence it may be difficult to predict at the outset of a project whether a behaviour-based agent will perform its assigned tasks appropriately. In summary, the risk of using a behaviour-based architecture is high when the agent is in a complex environment or the behaviour requires what appears to be cognitively demanding tasks, but not otherwise.

The process for finding an effective partitioning of required functionality into behaviours is a rather haphazard process usually requiring either experimentation or expert 'intuition'. This haphazard process makes it difficult to predict the amount of time required for producing a certain behaviour, doing detailed designs or determining if a certain percentage of the agent is complete.

An ad hoc process for building agents, combined with the limited ability to properly test them is likely to mean that the Chicken Factory will not scale up to large projects. It would be difficult to plan a development with any confidence, spread work between developers and even to assess the current status of the work.

TACSI defines no clear process for the development of agents. The predefined structure for the different agent types give a designer a clear guide when starting out. Agents will usually be developed in an incremental fashion with design decisions being made in a fairly ad hoc way. The specification method for rules supports incremental development well. Initial general rules can be created first then more specific rules added when special situations are observed during testing.

The predefined high level states provide clear modularity. Potentially different designers could work on different parts of a specification. Anecdotal evidence suggests that usually a developer works alone on a particular scenario specification (a scenario consists of a number of aircraft). This does not appear to be due to an inability to divide the task up efficiently rather simply that the specifications are small enough that a single developer can complete the task in a reasonable time. However the fairly simple nature of the architecture means it is also unlikely to scale up to very complex agents.

4.6. *Agent correctness*

4.6.1. *Characteristic 6.* Correctness issues for an agent architecture involve whether the architecture supports verification and validation and how easy it is to test and debug agents developed using the architecture.

At the outset of a project, a designer will have some requirements (more or less formalized) on the expected behaviour of the agent(s) being developed. In industrial projects it is important that developers ensure that the developed software meets the original requirements. 'Validation' is the process of ensuring that the behaviour of the agent fulfills the original requirements. 'Verification' shows desirable properties of the agent software, for example that it is free from deadlock. 'Testing' is the process of determining whether the agent performs acceptably across some range of situations (Beizer 1990). 'Debugging' is the process of fixing the agent specification when tests reveal incorrect behaviour. The nature of the architecture can influence the efficiency of achieving each aspect of agent correctness. Architectures aiding these aspects of agent development can help in reducing the high costs of assuring agent correctness (Rao *et al.* 1993).

Validation is often an aspect of the development process of the agents as a whole. The DMSO indicates validation is an important, difficult problem facing agents for simulation environments (Pew and Mavor 1998). If an architecture has facilities for supporting validation it would be a big advantage for some projects.

Verification is a formal process used to show certain properties of the agent software, e.g. proving logically that there are no deadlock states or that a particular communication protocol has certain properties. With some architectures agents are difficult or impossible to verify, for example it is beyond the state of the art to completely verify any architecture where there are complex, asynchronous interactions between different parts of an agent. Some architectures, however, do support formal verification of certain properties, for example (Bouchefra *et al.* 1998, Brazier *et al.* 1998).

To properly test an agent, one should create a carefully designed set of tests and observe the actions of the agent. The tests should be designed according to the application reliability requirements (Beizer 1990). The ability to create straightforwardly and execute a set of situations for a test is dependent both on the application and on the agent architecture. For example to create a particular test for ground avoidance of a simulated pilot requires both getting the simulated environment into a position where the aircraft needs to avoid the ground and the simulated pilot into the state of reasoning that should be tested, e.g. realizing that the ground is near and action must be taken, even if other hazards are simultaneously present. Important factors affecting the testability of an architecture include: whether the resulting agents are deterministic; whether pieces of a specification can be tested individually; and, whether it is possible to manually set an agent to a particular reasoning state (e.g. current knowledge and goals). Low *et al.* (1999) describe a technique for testing BDI agents that gives a certain level of confidence to the developers.

The process of testing identifies situations where the agent does not perform as expected. Incorrect behaviour may be due to a variety of reasons. Reasons for incorrect actions include: an agent's sensors not providing accurate enough information for the decision-making process; the agent's behaviour specification being incorrect or incomplete; or the architecture having fundamental weaknesses that do not allow the correct decision to be specified. Debugging an agent involves determining the reason for the incorrect behaviour and fixing it, if possible. Three important,

interdependent, architectural factors affect the ability of a designer to identify the reasons for the incorrect behaviour from the observation of that behaviour.

The first architectural characteristic affecting the ability of a designer to debug incorrect behaviour is the complexity of the transformation between an agent specification and the output behaviour of the agent. The more complex that transformation is, the more difficult it is to identify the specification elements involved in causing the error. For example a planner which performs a complex process on the plan primitives which constitute the specification will probably be difficult to debug.

The second architectural characteristic affecting the ability of a designer to debug an agent is the number of elements involved in a particular agent decision or action. The more specification elements that are involved, the more elements the programmer needs to consider in order to find the problem.

Third and finally, debugging difficulty is reduced by any system support that gives a designer feedback on the agent's reasoning. The system support could be in many different forms ranging from real-time graphical feedback to log files of the reasoning process.

4.6.2. *Examples*. Verification is likely to be almost impossible for agents created with the Chicken Factory architecture because the actual behaviour of the agents is often very difficult to predict, let alone try to prove properties about. The difficulty is partially due to the uncertain and complex environments in which the agents are usually applied, but it is also due to the complex interactions among the behaviours themselves.

The Chicken Factory architecture is very cumbersome to test but relatively easy to debug once an error is found. Testing any behaviour-based system is difficult (Strippgen 1997). Testing needs to be done in the final environment because of the close connection between the agent and environment. Despite the Chicken Factory itself being deterministic, the dynamic, non-deterministic, complex and incomplete environments in which the agents will usually be acting means that situations are unlikely to be exactly repeatable. There is no architectural support in the Chicken Factory for setting the reasoning process of an agent to some state. This means it will be difficult and probably time consuming to arrange some tests. On the other hand, testing can be done in a modular way. If a previously tested element is used as part of a larger specification there is no need to retest the element, however the larger specification needs to be tested as the interactions between the behaviours can be difficult to predict.

As the Chicken Factory provides no support for verification or validation and makes extensive, systematic testing very cumbersome it is unlikely to be a good candidate for applications requiring very reliable agents.

Debugging is relatively straightforward within the Chicken Factory. There is no transformation between an agent specification and a running agent. Graphical interfaces are provided which allow a user to determine which behaviours are currently affecting the agent output. At each layer of an agent, only one behaviour is chosen to act. This means that there is one specification element per layer affecting the actions of the agent. The combination of no transformations, a small number of impacting specification elements and debugging interfaces makes debugging agents rather straightforward.

TACSI provides no support for validation or verification. Testing and debugging support is extensive. The transformation between the specification and the run-time

representation is very small, i.e. the designer explicitly specifies a state machine which is responsible for decision making at run-time. When the agent is in a particular state there are usually very few rules to choose from, in most cases less than five. The number of available rules is the number of elements impacting on a decision. The number is low indicating that debugging is likely to be fairly straightforward. Finally TACSI provides a number of interfaces to view the status of the simulated aircraft and the simulated pilot. The view of the simulated pilot gives a designer feedback on the current state and rule firings. Information on rule firings is available both at run-time and as a log file for post analysis. Overall TACSI provides substantial support for debugging.

4.7. *Computational requirements*

4.7.1. *Characteristic 7.* It is necessary to have some idea of the computational requirements before agents are developed, to prevent the development of a system that can't feasibly be fielded once developed.

Analysing the computational requirements of intelligent agents has mainly focused on bounding computational requirements, e.g. Dongha (1994). Other work has been done to create agents fulfilling real-time requirements, e.g. CIRCA (Musliner *et al.* 1993). In between these two extremes, very little is said about the computational efficiency of the agents. However project managers are likely to be interested in more detail about exactly what hardware will be required to achieve a particular purpose. In many types of emerging applications, designers require agents that use very small amounts of computational time, for example in computer games or telecommunications network management. In other emerging applications, it may not be so important that the usage is low but it is important to know approximately what the computational requirements are, for example to know what machine needs to be acquired and thus the cost involved.

From the study of algorithms in computer science one might be tempted to borrow complexity analysis to measure the computational requirements of an architecture. There are, however, usually too many interdependent factors when it comes to agent architectures for such an analysis to be feasible, e.g. the 'quality' of the decision making. Rather it seems that benchmarking of architectures combined with an analysis of how the performance is expected to change along different axes, for example specification size, would provide more useful, usable information.

The following two benchmark tests are proposed to form the base of a computational benchmark analysis:

- The time between an unexpected event being detected by an agent and that agent performing its first action in response to the event. The test provides a benchmark on the reactive computational requirements of the agent.
- The time between the adoption of a new high level goal to the beginning of actions performed in pursuit of that goal. In agents that determine their own high level goals, this should be the time between realizing a new high level goal is appropriate, and actions beginning towards that goal. The high level goal should be something that requires the high level reasoning of an agent if it exists. This test provides an indication of the computational requirements of the reasoning of the agent.

The benchmarks should be supplemented with an analysis of how the experimental values are expected to vary with factors such as the size of the specification, the

complexity of the environment, the complexity of the task and any other relevant factors.

4.7.2. *Examples*. For agents created using the Chicken Factory, the appropriate reaction for a situation should be in the current set of low level behaviours. Hence the reaction time will be the time necessary to check all the behaviours on the bottom level. The exact times are influenced slightly by how much computation is required by the predicate associated with the behaviour. Every new behaviour added at the bottom level increases the complexity by only a linear factor.

A high level goal is adopted by changing behaviours at a higher level of the system. Once the behaviours have changed at the top level, a new set of behaviours is produced at the next level from which a behaviour is chosen, which in turn produces new behaviours at the next level down. If a goal is more complex it may either be at a higher level or it may be implemented with more sub-behaviours. Either way, the increase in computational time is linear with the increase in complexity of the goal.

The TACSI agent architecture is extremely efficient. At every cycle only the transitions and rules for the current state need to be computed. Handling an unexpected event is somewhat difficult to measure within TACSI. If a rule is available to handle the situation in the current state, the reaction time may be very fast. However often the event will be handled in a different state requiring a number of state changes in order to handle the unexpected event. Adopting a new high level goal in TACSI involves a change in top level states. Again due to the nature of the architecture this time will vary greatly. A state transition up to a parent state must explicitly occur as must a state transition to a child state. The time taken to change high level goals depends on how deep into the hierarchies the agent is.

Having a larger number of states has no direct influence on the computational requirements of the agent, although indirectly it may mean there are more transitions hence higher computation. Hence the increase in computation time with rules is linear.

5. Discussion

The characteristics presented in the previous section are intended as guidelines, rather than a checklist, for the evaluation of an agent architecture with respect to engineering and industrial concerns. These characteristics complement existing methodologies for agent-oriented design such as Wooldridge *et al.* (1999). This list of characteristics is not intended to cover the behaviour of the agents, i.e. it specifically excludes evaluation of the functionality of developed agents, which has been covered elsewhere (e.g. Johnson 1995, Bates 1993).

Much engineering work is about trade-offs, finding solutions that meet a range of constraints other than just functionality, including maintainability and costs. Some constraints are contradictory. It follows then that one cannot simply state engineering requirements on an agent architecture and any architecture fulfilling the requirements is appropriate for all industrial tasks. Instead, there will be a range of characteristics that architecture possess, which will make different architectures more suitable for different tasks.

Clearly conflicts will occur when trying to evaluate and select an architecture using the characteristics described. No architecture will be the best in all characteristics for an application. Prime examples of characteristics that interact with one another are

representational power and accessibility, i.e. architectures with higher representational power are likely to be more difficult to understand and vice versa. However, the fact that trade-offs appear is not a problem in itself. Industry is about engineering products and engineering is, in turn, about trade-offs. Architecture designers need not attempt to design architectures that cover all possible industrial requirements rather they should be making explicit the strengths and limitations of their architectures, so that engineers can select architectures that are strong in the characteristics that are most important to their current project.

6. Summary

One has proposed a set of engineering characteristics as a guide that managers can use to select which architecture to use for a particular project. The other side of the coin is that the characteristics can be thought of as issues for consideration for designers of agent architectures for industrial purposes. Clearly an architecture evaluating highly against many of the characteristics described would be suitable for a wide range of industrial projects. However, it is not necessary that an agent architecture excel for all characteristics. Good performance for one or more of these characteristics may make an architecture a prime candidate for some set of applications regardless of the performance with respect to the other characteristics.

7. Acknowledgements

This work is supported by Saab Corporation, Operational Analysis Division, the Swedish National Board for Industrial and Technical Development (NUTEK) under grants IK1P-97-09677, IK1P-98-06280 and IK1P-99-6166, and the Center for Industrial Information Technology (CENIIT) under grant 99.7.

References

Andersson, J., 1995, Plan oriented and rule-based system for generation of robust tactics in long range air combat with multiple targets (in Swedish) MA thesis, Linköping University.

AURAN, 1999, The official tactics engine artificial intelligence personalities and scenario end conditions guide. http://www.auran.com

Bates, J., 1993, The nature and characters in interactive worlds and the OZ project. Technical Report CMU-CS-92-200, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, October 1992. Also to appear in C. Loeffler (ed.) *Virtual Realities: Anthology of Industry and Culture.*

Beizer, B., 1990, *Software testing and techniques* (New York: Van Nostrand Reinhold).

Booch, G., 1994, *Object-Oriented Analysis and Design* (Redwood, CA: Benjamin Cummings pub.).

Bouchefra, K., Auge, P., Maury, T., Rozoy, B., and Reynaud, R., 1998, Multi-agent based architecture specification and verification. In *Proceedings of Eleventh Workshop on Knowledge Acquisition and Modeling and Management.*

Brazier, F., Cornelissen, F., Gustavsson, R., Jonker, C., Lindeberg, O., Polak, B., and Treur, J., 1998, Compositional design and verification of a multiagent system for one-to-many negotiation. In *Third International Conference on Multi-agent systems*, Paris, 1998, pp. 49–67.

Bryson, J., 1998, Agent architectures as object oriented design. In M. Singh (ed.) *The Fourth International Workshop on Agent Theories, Architectures and Languages* (*ATAL97*), Springer Verlag.

Bussman, S., 1998, Agent oriented programming of manufacturing control tasks. In *Third International Conference on Multi-agent systems*, Paris, pp. 57–63.

Chaib-draa, B., 1997, *Readings in Agents*, chapter Industrial Applications of distributed AI. Morgan Kaufmann.

Chalmers, A., 1982, *What is this thing called Science?* (Indianapolis: Hackett Pub. Co.).

Cockburn, D., and Jennings, N., 1996, ARCHON: A Distributed Artificial Intelligence System For Industrial Applications, *Foundations of Distributed Artificial Intelligence*, Wiley, pp. 319–344.

Coradeschi, S., 1997, A decision-mechanism for reactive and coordinated agents. MA Thesis, Department of Computer and Information Science, Linköping University.

de Champapeaux, D., 1997, *Object-Oriented Development and Process Metrics* (NJ: Prentice Hall).

Defense Modelling and Simulation Office, 1993, DMSO survey of semi-automated forces. Technical report, Defense modelling and simulation office.

Dongha, P., 1994, Toward a formal model of commitment for resource bounded agents, *Intelligent Agents* (Heidelberg: Springer-Verlag)

Jennings, N., 1995, The ARCHON system and its applications. Project Report, International Working Conference on Co-operating Knowledge based systems, Keele, UK.

Johnson, W., 1995, Pedagogical agents in virtual learning environments. In *Proceedings of the International Conference on Computers and Education*.

Kinny, D., 1993, The distributed multi-agent reasoning system architecture and language specification. Technical report, Australian Artificial intelligence institute, Melbourne, Australia.

Kinny, D., and Georgeff, M., 1996, Modelling and design of multiagent systems. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*. Lecture Notes in Computer Science.

Kitano, H., Asada, M., Kuniyoshi, Y., and *et al.*, 1997, RoboCup: a challenge problem for AI. *AI Magazine*, **18**(1): 73–85.

Lorenz, M., and Kidd, J., 1994, *Object Oriented Software Metrics* (Prentice Hall).

Low, C., Chen, T., and Rönnquist, R., 1997, Automated test case generation for BDI agents. In *Autonomous Agents and Multi-Agent systems*, **2**(4): 311–333.

Mataric, M., 1992, Behavior-based systems: main properties and implications. In *IEEE International Conference on Robotics and Automation*, *Workshop on Architectures for*, Nice, France, pp. 46–54.

Musliner, D., Durfee, E., and Shin, K., 1993, CIRCA: a cooperative intelligent real-time control architecture. *IEEE transactions on Systems, man and cybernetics*, **23**(6): 1561–1574.

Pell, B., Gamble, E., Gat, E., Keesing, R., Kurien, J., Millar, W., Nayak, P., Plaunt, C., and Williams, B., 1998, A hybird procedural/deductive executive for autonomous spacecraft. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 369–376.

Pew, R., and Mavor, A., 1998, *Modeling Human and Organizational Behavior* (Washington, DC., National Academy Press).

Pirjanin, P., 1998, *Multiple objective action selection and behavior fusion voting*. PhD Thesis, Department of Medical Informatics and Image Analysis, Aalborg University.

Rao, A., Lucas, A., Morlay, D., Selvestrel, M., and Murray, G., 1998, Agent-oriented architecture for air combat simulation. Technical Report Tech. Rep. 42, Australian Artificial Intelligence Institute, Melbourne, Australia.

Saab, 1998, The TACSI users guide. Technical report GDIO-MI-98:356. Saab Military Aircraft Edition 5.2, in Swedish.

Scerri, P., and Ydren, J., 1999, End user specification of robocup teams. In *Workshop on Robocup, Sixteenth International Joint Conference on Artificial Intelligence*, August, Stockholm, Sweden.

Scerri, P., Coradeschi, S., and Törne, A., 1998, A user oriented system for developing behavior based agents. In *Proceedings of RoboCup '98*, Paris.

Schoepke, S., 1999, Facilitating the deployment of intelligent agents in the application development mainstream. *Agent Link News*, **3**: 10–12.

Strippgen, S. 1997, Insight: a virtual laboratory for looking into behavior-based autonomous agents. In *Autonomous Agents '97 Online Proceedings*, CA, pp. 474–476.

Tambe, M., 1997, Agent architectures for flexible, practical teamwork. In *National Conference on AI*, Providence, RI (Menlo Park: *AAAI97*), pp. 22–28.

Wooldridge, M., and Jennings, N., 1998, Pitfalls of agent oriented development. In *Proceedings of the Second International Conference on Autonomous Agents*, Seattle WA.

Wooldridge, M., Jennings, N., and Kinny, D., 1999, A methodology for agent-oriented analysis and design. In *Proceedings of the Third International conference on Autonomous Agents*, pages 69–76.