From SMART to Agent Systems Development

Ronald Ashri [a] Michael Luck [a] Mark d'Inverno [b]

[a] *School of Electronics and Computer Science, University of Southampton, Southampton, SO14 1JX, UK*

[b] *Cavendish School of Computer Science, University of Westminster, London, NW1 3ET, UK*

Abstract

In order for agent-oriented software engineering to prove effective it must use principled notions of agents and enabling specification and reasoning, while still considering routes to practical implementation. This paper deals with the issue of individual agent specification and construction, departing from the conceptual basis provided by the SMART agent framework. SMART offers a *descriptive specification* of an agent architecture but omits consideration of issues relating to construction and control. In response, we introduce two new views to complement SMART: a *behavioural* specification and a *structural* specification which, together, determine the components that make up an agent, and how they operate. In this way, we move from abstract agent system specification to practical implementation. These three aspects are combined to create an agent construction model, *act*SMART, which is then used to define the AgentSpeak(L) architecture in order to illustrate the application of *act*SMART.

1 Introduction

As computer systems become more sophisticated and complex, the associated difficulties of effectively managing the development process increase dramatically. This is further complicated by changes to computing environments, with increased distribution and openness, and embedded computers on, for example, everyday household appliances. Such changes have placed huge demands on system designers, who must now take into consideration a wide range of issues, such as ad-hoc networking, user and device mobility and intelligent or flexible behaviour, and integrate them within a single system design.

Email addresses: ra@ecs.soton.ac.uk (Ronald Ashri), mml@ecs.soton.ac.uk (Michael Luck), dinverm@wmin.ac.uk (Mark d'Inverno).

It has been argued that agent-based computing can make the task of designing systems for such environments easier [1]. The underlying concept of decentralised, autonomous control expressed through agents that are able to communicate and cooperate to achieve goals is especially appealing for applications in heterogeneous and dynamic computing environments. In particular, agent-oriented software engineering offers a way to manage the complexity of large-scale, distributed, complex software systems. Building on the agent paradigm in which these autonomous problem-solving components are used to decompose a system into its constituent parts, agent-oriented software engineering seeks to provide a more natural means and effective means of development.

In order to do this, however, two distinct aspects are required. First, it is necessary to provide a way of specifying the individual agents and the relationships between them in support of a general system architecture. Second, and an aspect that is often omitted from many agent-oriented frameworks, a means for moving from this specification to an implemented application is required. In this paper, we review the SMART agent framework that addresses the first of these requirements, and describe how it may be used as a basis for the second requirement. In this respect, we focus on an *agent construction model* and show how it may be viewed as a system specification from an implementation perspective.

1.1 From Framework to Construction

If a model or methodology for agent development is to be general, it should not lead to the development of only a limited range of agent types for limited applications and domains, but should allow the widest possible range of architectures to be defined using the same basic concepts. There are two possible avenues to explore in support of this aim. One option is to define a generic agent architecture and describe other architectures in terms of this generic architecture. Such an approach has been suggested by Bryson in [2]. However, apart from the inherent difficulty in constructing any general, all-inclusive model, the drawback of this approach is that there may be features of other architectures that cannot directly be *translated* to the generic one. The second option is to provide an architecturally-neutral model, to avoid this translation problem. Here, the challenge is to provide a model that is specific enough so that it actually offers something to the construction of agents, but general enough to support the development of a wide range of architectures. Through an appropriate *architecturally-neutral* model, we can consider a range of architectures based on a common set of agent-related abstractions and without losing expressive capability.

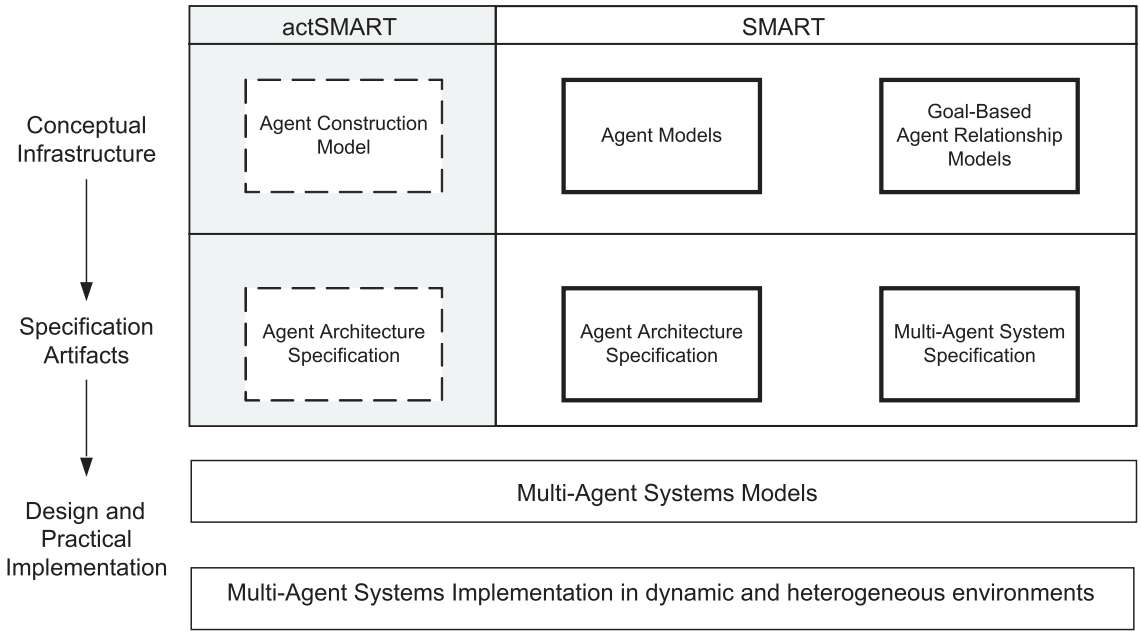

Fig. 1. The relationships between *act*SMART and SMART

The SMART agent framework provides us with a set of abstract, formal models to support the specification of individual agent architectures and multi-agent systems in just this way. However, the model should also allow for modular *construction* of agents, yet SMART is limited to considering *specification* alone. Modularity in construction is necessary both in order to meet general software engineering concerns and to delineate clearly the different aspects of an architecture. This approach calls for a separation between describing agents in terms of their *characteristics*, their *structure* and their *behaviour*. Such an approach leads to a better understanding of the overall functioning of the agent as well as how it can be altered, since the different aspects of the architecture are clearly identified and the relationships between them made explicit.

Thus, while previous work on SMART is suitable for *describing* agents, it lacks the necessary features for *constructing* agents. For the purposes of reasoning about systems, this is not a problem but, more generally from a methodological perspective, it is crucial to be able to provide tools that facilitate the construction of agent architectures. In this paper, therefore, we do not *replace* the descriptive capabilities of SMART but instead complement them with additional aspects, which are identified below.

1.2 Overview

In addressing the shortcomings identified above, we extend and refine SMART in two directions by both providing more practical models and adding to the abstract concepts already there, as illustrated in Figure 1. In the figure, we rep-

resent three different levels of abstraction. First, the conceptual infrastructure defines the models that can be used to specify an agent system. Second, the *specification artifacts* represent specific instantiations of these models in order to design an agent system. Finally, the *design and practical implementation* represents the resulting multi-agent systems developed using the specification artifacts from the level above. In this paper, we reflect these three levels in the consideration of SMART and its extensions culminating in *act*SMART.

The SMART framework provides the conceptual infrastructure for describing agents and goal-based agent relationships. These models *enable* the specification of agent architectures and multi-agent systems, respectively.

However, we also require appropriate *practical* models for agent construction to provide a clear path from the abstract agent models of SMART to their implementation. We therefore need to extend SMART in a more practical direction, while basing this extension on the existing abstract agent models. In the figure, this extension is under the heading of *act*SMART, (*A*gent *C*onstruction *T*oolkit for SMART). At the conceptual infrastructure level, *act*SMART provides a model for constructing agents which, at the specification artifact level, can enable the specification of agent architectures that can then find practical implementation at the lowest level.

In this paper, we begin by briefly reviewing the SMART agent framework, and explain how it can be viewed as a descriptive system specification. Then we introduce alternative views of structure and behaviour that facilitate the practical development of agent systems. We consider each in turn, showing how such systems are constructed with components based around a central agent shell. The paper ends with an example system, based on AgentSpeak(L), and shows how these three views offer different perspectives on agent systems development.

2 SMART as Descriptive Specification

We have previously proposed an agent framework, SMART, through which to address the lack of an unambiguous agent theory that could be used to describe and relate existing work in the field, as well as act as the basis through which to develop new systems and theories. A key benefit of SMART is that it avoids dependence on any specific agent architecture and does not make any limiting assumptions about the environment or agent societies. This is particularly important, since we aim to accommodate heterogeneous agent societies, in which a variety of agent architectures, and dynamic environments, need to be supported.

In this section, we briefly review the SMART framework, before moving to consider how it is used as the basis of agent construction. We do not provide a detailed analysis of the framework — the work is described elsewhere [3] — nor of the Z notation used to express it. Briefly, however, we have adopted the specification language, Z [4], in the current work for two major reasons. First, it is sufficiently expressive to allow a consistent, unified and structured account of a computer system and its associated operations. Second, we view our enterprise as that of building programs. Z schemas are particularly suitable in squaring the demands of formal modelling with the need for implementation by allowing transition between specification and program. Thus our approach to formal specification is pragmatic: we need to be formal to be precise about the concepts we discuss, yet we want to remain directly connected to issues of implementation and program development.

Based on set theory and first order predicate calculus, Z extends the use of these languages by allowing an additional mathematical type known as the schema type. Z schemas have two parts: the upper declarative part, which declares variables and their types, and the lower predicate part, which relates and constrains those variables. Since in this paper, our use of Z is relatively limited, we will not say more here, and assume familiarity on the part of the reader.

Below, we present the conceptual infrastructure of SMART, by considering the specification of individual agents. The modular approach used throughout SMART means that these concepts form the *foundation* for all subsequent definitions, including agent relationships (though we will not consider relationships in this paper). This is particularly useful because it is consistent with our aim of supporting *reusable* agent models. We begin with a detailed presentation of these foundational concepts that define the different types of entities and their relationship to the environment. Then, we discuss SMART's notion of agenthood and introduce a refinement that provides more granularity in the different types of agents that we can define.

2.1 Objects and Agents

At the base of SMART is a view of agents as *entities* attempting to satisfy goals, where goals are desirable states of affairs. Entities are hierarchically organised in four different types, with each type refining the previous one. These entity types are described using three primitives — *attributes*, *actions* and *motivations* — each formally represented as a given set with no restrictions on how it could be manifest in a particular system instantiation. A short description of the primitive types follows, before we go on to describe the different entity types. The formal specification in Z is shown in Figure 2.

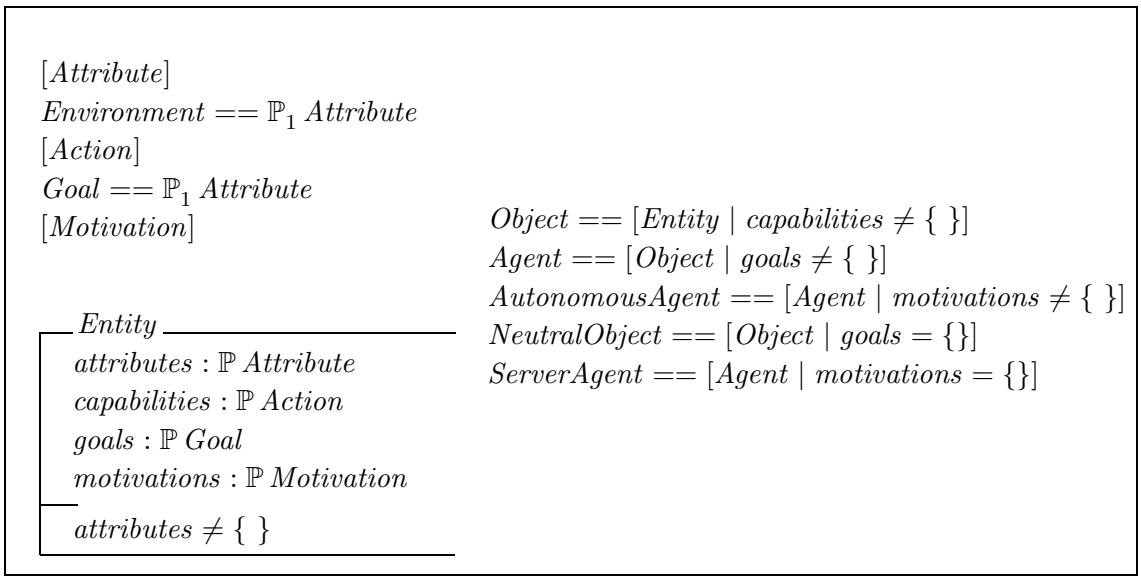

Fig. 2. Formal specification of the agent framework

Attributes are perceivable features of the environment and, through them, entities and the environment in which they are situated can be described. For example, if we consider a mobile device as an entity, then some of the attributes that can be used to describe it are the name of the owner of the device, the location of the device, and so forth. An environment can then be defined as a non-empty set of attributes. Actions are discrete events that can change the state of the environment. For example, a mobile device can perform actions such as communicating with other devices, storing information, and retrieving online documents.

In the traditional artificial intelligence sense, goals are desirable state of affairs in the environment, and are represented as non-empty sets of attributes. For example, a goal to find a particular online document can be described as a state of affairs in which the location of the document is known. Finally, motivations are any *desires* or *preferences* that drive an agent to set its own agenda, as opposed to having goals dictated to it by a user or by other agents. It is defined as a given set.

The four different *entities* can now be considered using these primitive types. First, the abstract *Entity* schema of Figure 2 defines an entity to have a set of attributes, a set of actions (their capabilities), a set of goals and a set of motivations. The only restriction for something to be of type *Entity* is that it must have a non-empty set of attributes, as stated in the predicate part of the schema. Entities in this sense are just placeholders.

Objects, which can correspond to real artifacts in the environment, are entities with some capabilities that make it possible for them to perform actions that can change the environment. Thus, the *Object* schema includes the *Entity*

schema, and further restricts it by requiring that the set of capabilities is non-empty.

Building on this, *agents* can then be defined as objects that are attempting to achieve goals. This means that there is a desirable state of affairs in the environment that they are attempting to bring about. Correspondingly, the *Agent* schema includes the *Object* schema and constrains the set of goals to be non-empty. Agents can have or be *ascribed* goals that they retain over any instantiation or lifetime.

The definition of agents given above relies on the existence of other agents to provide the agent's goals or ascribe goals to the agent. This means that some other entity is *always* required to provide or ascribe the goals. In order to ground the entity hierarchy, therefore, some agents must be able to *generate* their own goals. These agents are defined as *autonomous* since they do not depend on the goals of others, and possess goals that are *generated from within* rather than adopted from other agents. Such goals arise through *motivations*, which both cause an autonomous agent to generate its own goals and guide it in choosing the goals to adopt when interacting with other agents. Formally, the *AutonomousAgent* schema requires the set of motivations to be non-empty. We will not discuss the *generation* of goals by motivated agents, but an extensive analysis is available elsewhere [3].

In this way, the notions of agents and autonomous agents are clearly distinguished. Agenthood is ascribed to any entity that acts in order to satisfy some goal, and motivations are required to support the self-generation of goals by agents. The ability of an agent to generate its own goals is what defines it as autonomous.

In addition to these basic concepts, SMART also considers how non-autonomous agents are created. In order to achieve this, the basic framework is further refined to accommodate more sophisticated analyses of agent interaction by introducing additional definitions of *neutral objects* as those objects that are not agents, and *server agents* as those agents that are not autonomous. The relationship between neutral objects and server agents is complementary and dynamic. Neutral objects become server agents when they are given or ascribed goals. Once these goals are achieved, or when pursuing them is no longer feasible, the server agent reverts back to a neutral object. The schemas in Figure 2 formalise these concepts. A *NeutralObject* is an *Object* with empty sets of goals and motivations while a *ServerAgent* is an *Agent* with an empty set of motivations.

The relationships between all the different entity types are illustrated in Figure 3, in which they are shown as a Venn diagram. As indicated, the most general notion of entity subsumes all other notions, while neutral objects

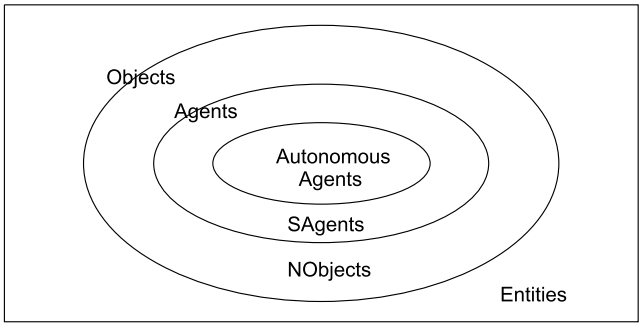

Fig. 3. The entity hierarchy

(NObjects) and server agents (SAgents) lie in the space between objects and agents, and between agents and autonomous agents, respectively.

2.2 From Description to Structure and Behaviour

SMART allows systems to be specified from an observer's point of view. Agents are described in terms of their attributes, goals and actions, not in terms of how they are built or how they behave. In other words, the focus is on the *what* and not the *why* or *how*. We call this a *descriptive specification*, since it essentially describes a situation without analysing its causes nor the underlying structures that sustain that situation. For example, if we return to the issue of neutral objects becoming server agents when *engaged*, we can see that SMART says nothing about what happens structurally within the entity that has changed status, nor how the mechanisms controlling its behaviour have brought about this change. These are the types of issues we need to address within an agent construction model. Therefore, *along* with the descriptive specification we need to have the ability to specify systems based on their structure, i.e. the individual building blocks that make up agents, as well as their behaviour. We call these other views the *structural specification* and the *behavioural specification*, respectively.

The structural specification enables the identification of the relevant building blocks or components of an agent architecture. Different sets of building blocks and different ways of connecting them can enable the instantiation of different agent types. By contrast, the behavioural specification of an agent addresses the process through which the agent arrives at such decisions as which actions to perform. Along with the descriptive specification, these specifications provide a more complete picture of the system from different perspectives. It is interesting to note that it is possible to begin from any one of these views and derive the remaining two, but the correspondence is not one-to-one. Several behavioural and structural specifications could satisfy a single descriptive specification and *vice versa*.

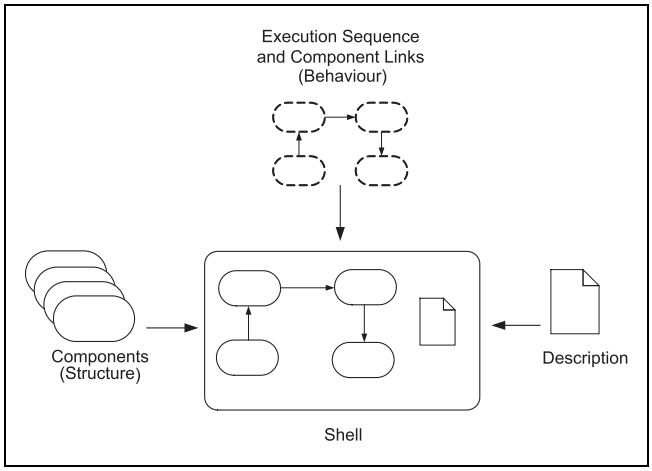

Fig. 4. Agent construction model overview

To support these different views, and to bring them together in a practical way that facilitates development, we need a particular focus for them in a computational sense: the *shell*. Thus the main concepts and the relationships between the descriptive, structural and behaviour specifications are illustrated in Figure 4, in which the central artifact is the *shell* that manages an agent architecture, with the architecture being made up of *components*. Components are placed within this shell and the *links* for data-flow between components are defined through the shell. In addition, the *execution sequence* of components is defined by the shell. The components form the structural specification of the agent, while the links and execution sequence define the behavioural specification. Finally, a description of the overall agent is also stored within the shell to complete the descriptive specification of the agent. These features provide for a *modular* architecture with clear distinctions between the different aspects of the architecture.

Now, since individual components are independent of the existence of other components, and all links between them are managed by the shell, we can more easily replace components or change data-flow between components in the shell, as well as alter the execution sequence. These features allow us to *reconfigure* the architecture in response to changing application requirements or changing environmental needs.

Throughout, the main concepts that underpin the development of agent architectures are the abstract agent model provided by SMART, and a functional separation of components into four generic types, described in the next section. The different component types allow us to define architectures without needing to specify the *internal* behaviour of components in great detail. These features support the requirement for an *architecturally-neutral* model that can be applied in a wide range of situations. The structural and behavioural views are considered in more detail below.

3 Structural Specification through *act*SMART

The aim of the agent construction model for SMART, *act*SMART, is to embody the design principles discussed in Section 1 as well as support construction based on the underlying concepts of SMART, while providing a direct route to implementation. Central to these concerns is the distinction between the structural, behavioural and descriptive specifications and a modular, reconfigurable approach. In this section, we examine the structural specification, which is concerned with the different types of components that can be used to make up an agent architecture, as well as the types of information exchanged between them, and the operational cycle of a component.

3.1 Components

In order to support the division of an architecture's different aspects as described above, and to satisfy the requirement for modularity and re-configurability, we take a component-based view of agent architectures [5,6]. Components are understood as units of composition that can be deployed independently from each other, through a third-party that coordinates their interactions [7]. Interaction with a component takes place through a well-defined interface, which allows the implementation of the component to vary independently of other aspects of the system.

The are several benefits of decomposition through a component-based approach, in line with our aims.

- Describing an agent architecture through the composition of components promotes a clearer identification of the different functionalities, and allows for their reuse in alternative contexts.
- Different types of components can be composed in a variety of ways to achieve the best results for the architecture at hand.
- By connecting the abstract agent model of SMART to component-based software engineering, we bring it much closer to practical development concerns within a paradigm that is not foreign to developers.

Components are the basic buildings blocks for an agent, and they can be considered as the structural representations of one or more related agent functionalities, which are considered at two different levels. At an *abstract level*, the functionality is described in generic terms, which we present below. At the *implementation level*, the abstract functionality is instantiated through the actual computational mechanisms that support it. The reason for distinguishing between these different levels is so that we can use *generic* component types to specify an agent architecture at a high level of abstraction without making

direct reference to the detailed behaviour of each component. This allows us to move between the different levels while retaining a good understanding of the overall architecture, and identifying which specific components best suit each of the generic functionalities.

At an *abstract* level, we can divide components into four generic types, each representing a class of *generic* functionality for the agent: information collection (sensors); information storage (infostores); decision-making (controllers) and directly effecting change in the environment (actuators). These four generic types of components, elaborated in more detail below, can be used to describe a very wide range of agent architectures.

- *Controllers* are the main decision-making components in an agent. They analyse information, reach decisions as to what action an agent should perform, and delegate those actions to other components. Controllers are *stateless*, since each decision is taken depending just on information provided through inputs at any given execution, and not on previous decisions that a controller has taken. Information that may affect decisions over time should be stored in infostores so that it can be provided to controllers as required.
- *Sensors* are able to sense environmental attributes, such as signals from the user or messages received from other agents. They provide the means through which the agent gains information from the environment. Similarly to controllers, sensors are stateless.
- *Actuators* cause changes in environmental attributes by performing actions. Actuators are also stateless, since every action they perform is not influenced by previous actions.
- *Infostores* are components whose main task is that of storing information. This information could be anything from the beliefs of an agent about the world, to plans, to simply a history of the actions an agent has performed, or a representation of its current relationships with others. In contrast to the other components, infostores are not stateless. The information they store represents their current state, and the manner in which information changes results from the way the component manipulates and updates information. For example, in the case of a BDI architecture, there may be various ways of representing and updating beliefs, such as dealing only with beliefs referring to the current state of the environment [8].

3.2 Component Statements

The *internal* operation and structuring of components, irrespective of their type, is divided into a functionally-specific part and a generic part. In this subsection, we describe the generic part that is common to all components, and outline the types of information that components can exchange.

Each component accepts a predefined set of inputs and produces a predefined set of outputs. A component generates an output either as a direct response to an input from another component, a signal from the environment or an internal event. For example, a sensor component attached to a thermometer may produce an output every five minutes (based on an internal clock), or when the temperature exceeds a certain level (an external signal), or when requested from another component (as a response to the other component).

In *act*SMART, inputs and outputs share a common structure; they are *statements*, and have a *type* and a *body*. The body carries the information content (e.g., an update from a sensor), while the type indicates how the information in the main body should be treated. We make use of three types of statements, described below.

- INFORM-type statements are used when one component simply passes information to another component. In order for one component to inform another of something, it must be able to produce the INFORM-type statement as an output, and the other must be able to accept it as an input.
- REQUEST-type statements are used when one component requires a reply from another component. In this case, the receiving component processes the REQUEST and produces an INFORM statement that is sent to the requesting component. The mechanisms through which statements are transmitted from one component to an other are introduced in Section 4.2.
- EXECUTE-type statements are used to instruct another component to execute a specific action. Typically, controller components send such statements to actuators so that changes can be effected in the environment.

It should be noted that this list of statement types is not exhaustive, and they are simply representative of the needs of most applications due to their generic nature. Some domains may benefit from more specific statement types. It should also be noted that message-passing between components at this level should not be compared with message exchange as defined in high-level agent languages such as KQML/FIPA [9]. Typing statements simply provides additional information to aid control of component behaviour.

The information within a statement's body is, in its most general form, described through *attributes*, as per the definitions given in Section 2.1. For the purpose of practicality, we divide attributes along the lines of *architecture-specific* attributes and *domain-specific* attributes. Architecture-specific attributes are thoses that are only relevant within the internal scope of an agent architecture. For example, a BDI-based architecture could define attributes such as *plans*, *beliefs*, *intentions* and so forth. [1] Architecture-specific attributes can be considered as defining the *internal* environment of an agent. Domain-specific

[1] This approach was adopted by d'Inverno and Luck when formalising AgentSpeak(L) [10]

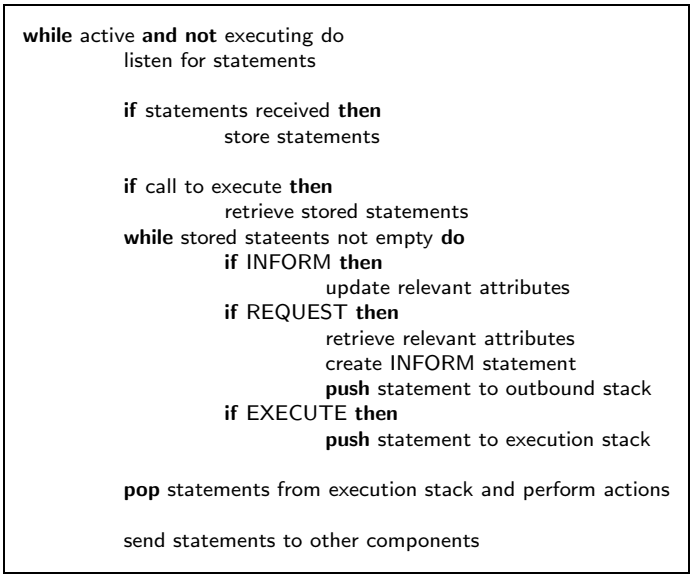

Fig. 5. Component Lifecycle

attributes define features that are relevant to the environment within which the agent is operating. In the case of an auction agent, for example, these attributes may include features such as *auction-house name*, *item for sale*, and so forth. Application-independent agent architectures, such as BDI-based architectures, typically make use of both types of attributes, including domain-specific attributes *within* the architecture-specific attributes. Thus, a *plan* may prescribe an action to contact a service, as identified by its *service name*. The components of an AgentSpeak(L) [10] architecture, for example, could then manipulate *plans* and *beliefs*, and have some generic way of manipulating the *domain-specific* attributes. However, a developer may also choose to develop an agent that has no architecture-specific attributes, creating components that can directly manipulate domain-specific attributes. Below, we describe a typical operation cycle for a component to explain how the different types of statements are handled.

3.3 Component Operation

An outline of the component operation is shown in Figure 5. Components begin their operation in an inactive state within the shell, where they do not receive or send statements. Once activated by the shell, components perform any relevant initialisation procedures and can then enter one of two possible types of operation. The default type is to receive statements until the shell calls them to enter their execution phase. An alternative behaviour is for the receipt of a statement to *trigger* their execution phase. Below, we consider the default operation first, before discussing the alternative.

When a statement is received, it is typically stored within the component until the component enters its component-execution phase, at which point all statements received by a component are processed. According to the type of statements received, the component performs one of three actions, as follows.

- An INFORM statement simply causes the component to update any relevant attributes, based on the information contained within the statement.
- An EXECUTE statement is placed on an *execution stack*. Once the processing of all received statements is completed, the EXECUTE statements are retrieved and the component performs the actions described within the statement.
- A REQUEST statement causes the component to attempt to retrieve the information requested and create an INFORM statement that contains that information. This INFORM statement is then placed in an *outbound* stack that stores all statements to be sent out. Outbound statements are sent once the processing of all received statements has finished and the actions prescribed by EXECUTE statements have been performed.

The entire process continues until a component is deactivated. Note that while a component is executing it cannot receive any statements. If statements are still arriving at the component, it is the task of the shell to manage those statements until the component is able to receive them.

The alternative behaviour for a component is to process every statement as it arrives, using the same method described above for the different types of statements. This *event-based* behaviour is especially useful for infostores that are typically queried with REQUEST statements for information, so that they can thus provide the response immediately.

At any given time, the state of a component, in terms of the information to be manipulated, is given by the set of statements that have not yet been processed, the set of statements in the execution stack, the set of statements in the outbound stack and any attributes that the component manipulates. Depending on the specific implementation of a component, it may be possible to interrogate components for their individual states.

4 Behavioural Specification

As discussed above, communication between components takes place through the exchange of statements. Individual components are not aware of the origin of received statements nor the destination of statements they produce, ensuring that components are independent of each other. Third-party coordination is achieved by placing components within a *shell*, which acts as the

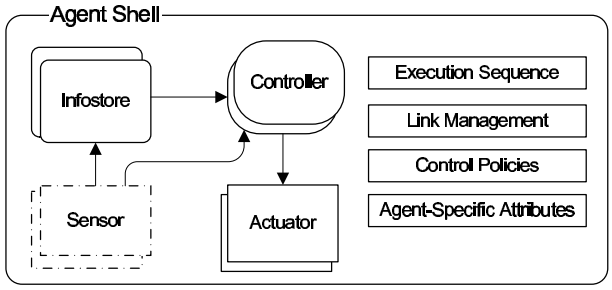

Fig. 6. Agent shell

third-party that manages the sequence in which components execute and the flow of information between components. These are the aspects that make up the behavioural specification of the agent.

4.1 Shell

The shell manages components, firstly, by defining *links* between components and, secondly, through the *execution sequence* of components. The basic aspects of a shell are illustrated in Figure 6. Components are placed within a shell, links are created between components to enable the flow of statements, and an execution sequence is defined. In addition, the shell can be used to maintain *descriptions* of agents in terms of attributes, capabilities and goals. We consider each of these aspects in more detail below.

4.2 Links

Information flows through *links* that the shell establishes between components. Each link contains *paths* from a *statement-producing* component to the *statement-receiving* components. Each component that produces statements has a link associated with it that defines the components that should receive those statements. Links also ensure that, in the case of a REQUEST statement, the reply is sent to the component that produced the request. Thus, links manage paths, which are one-to-one relationships between components. They are usually unidirectional, except in the case of a REQUEST statement, for which an INFORM may be returned in the opposite direction.

The shell then uses the information within links to coordinate the flow of statements between components. Ultimately, this coordination depends on the choices that a developer makes, since it requires knowledge of each component and how they can be composed.

By decoupling the handling of statements between components from the com-

ponents themselves, we gain considerable flexibility. We can manage the composition of components and the flow of information without the components themselves needing to be aware of each other. It is the architecture developer's task to ensure that the appropriate links are in place. At the same time, there is flexibility in altering links, and it becomes easier to introduce new components. Furthermore, basic transformations can be performed on a statement from one component to the other to ensure compatibility if the output of one component does not exactly match the required input for another. For example, if a sensor component provides information from a thermometer based on the Celsius scale, while a controller that uses that information uses Fahrenheit, the link can be programmed to perform the necessary transformation. These features thus enable the reconfiguration of architectures.

4.3 Execution Sequence

Apart from the management of the flow of information, we also need to consider the execution of components for a complete view of agent behaviour. This is defined via an *execution sequence* that is managed by the shell. Execution of a component includes the processing of statements received, the dispatch of statements, and the performance of any other actions that are required. The execution sequence is an essential part of most agent architectures and, by placing the responsibility of managing the sequence within the shell, we can easily reconfigure it at any point during the operation of the agent. For many architectures this may be purely sequential, but there are cases in which concurrent execution of components is desired (e.g., the DECAF architecture is based on a fully concurrent execution of *all* components [11]). In general, the issue of supporting complex execution sequence constructs, such as conditional paths and loops, is considered to be an issue that goes beyond the scope of this research, and there is a wealth of existing research that can be accessed to address this need. For example, recent developments within the field of Semantic Web Services provide a process model language for describing the operation of a web service [12]. Nevertheless, through our proposed mechanisms, we facilitate the necessary separation of concerns to enable the integration of such work within the scope of agent architecture development.

4.4 Agent Description

The description of the agent as a whole can be maintained by the shell or explicitly within the agent architecture, with components dedicated to the task, depending on the capabilities and needs of the architecture. In the former case, the shell can store a number of attributes that describe the agent

owner, its location, user preferences, etc. The level of detail covered by this description is mostly an application-specific issue, and this information can either be provided directly to the shell by the developer, or collected from the various components. The shell could query a component that is able to provide information about the current location, for example, and add it to the description of the whole agent. Likewise, it may keep a record of the current goal an agent is trying to satisfy, or the plan it is pursuing. The capability to collect and provide attributes describing the agent within the shell may be particularly useful in a situation in which a developer wants to export a view of the agent for debugging purposes, or when some information needs to be advertised, to facilitate discovery by other agents.

5 Applying *act*SMART to AgentSpeak(L)

In order to test the effectiveness of *act*SMART for constructing agent architectures we developed a BDI-type architecture based on Rao's AgentSpeak(L) [13] language. Luck and d'Inverno have previously formalised AgentSpeak(L) within the framework of SMART [14], and this formalisation is used as the basis of our implementation. The presentation is divided into the descriptive, structural and behavioural aspects of the architecture, and an overview of the three perspectives is illustrated in Figure 7.

The basic operation of agents in AgentSpeak(L) is based around their beliefs, desires and intentions. An agent has beliefs (about itself, others and the environment), desires (in terms of the states it wants to achieve in response) and intentions as adopted plans. In addition, agents also maintain a repository of available plans, known as the plan library. Agents respond to changes in their goals and beliefs, which result from perception, and which are packaged into data structures called events, representing either new beliefs or new goals. They respond to these changes by selecting plans from the plan repository for each change and then instantiating one of these plans as an intention. These intentions comprise actions and goals or plans to be achieved, with the latter possibly giving rise to the addition of new plans to that intention. More details of the architecture can be found in [13,14], but this brief description will suffice for the purpose of elaborating the *act*SMART model.

5.1 Descriptive Specification

As discussed in Section 3.2, the attributes used within an architecture can be divided into domain-specific and architecture-specific attributes. This distinction is adopted in our implementation of AgentSpeak(L) since it uses its

Descriptive Specification	Behavioural Specification	Structural Specification
Attributes Beliefs Intentions Plans Events Goals Triggers **Capabilities** Actions **Goals** Achieve Goals Query Goals	Receive sensor information Update Beliefs Trigger Plans Choose Plans Create new intentions Select intention to pursue Create internal events Perfom actions	Sensors BDIEventProcessor (Controller) Actuators BDIActionProcessor (Controller) BeliefBase (Infostore) PlanBase (Infostore) PlanSelector (Controller) IntentionBase (Infostore) IntentionSelection (Controller)

Fig. 7. Agentspeak descriptive, behavioural and structural specification

own *terminology* for describing both the external and internal environment of the agent. The descriptive specification is in essence a specification of this terminology, which can then be connected to the concepts of SMART.

In AgentSpeak(L), beliefs represent facts about the world. *Actions* are the plan steps that make up the *body* of plans and intentions. *Plans* are made up of *triggers* that define the invocation conditions (either a belief or a goal), the *context* that is the set of beliefs under which the plan is applicable, and the *body* that defines what actions are to be taken or what goals are activated. *Triggers* are divided into *external* and *internal* triggers where the former are determined externally and the latter are determined through components within the agent. *Goals* are simply collection of beliefs to be brought about and can either be *achieve* goals (making some belief true) or *query* goals (testing whether a belief is true). Finally, *intentions* are sets of plans that are currently activated.

In the formalisation of AgentSpeak(L), beliefs, events, intentions, triggers, goals and plans are all defined in terms of attributes. Thus, the descriptive specification of AgentSpeak(L) is a mapping between the SMART concepts of attributes, capabilities and goals and their respective concepts within the AgentSpeak(L) architecture. In essence, AgentSpeak(L) goals are the same as SMART goals, AgentSpeak(L) actions map to SMART capabilities, and descriptions of AgentSpeak(L) beliefs, intentions, plans, events, goals and triggers map to SMART attributes.

Note that we differentiate between a *goal* as an attribute used to describe the information stored within an agent (and as a result a describable feature of the agent's internal environment) and a goal as a desirable state of affairs in SMART. The former refers to a data structure stored within the architecture, while the latter refers to an intrinsic value that may not be explicitly stored within the architecture.

5.1.1 Structural Specification

The structural specification of an agent is based on its individual components and the kinds of statements that they produce or receive. The statements are constructed with attributes of the types described above. A description of each component and its main functionality follows.

BeliefBase The *BeliefBase* is an *infostore* that stores the beliefs of an agent. It can accept INFORM statements that carry beliefs to be added to, or update, the set of existing beliefs. It can also accept REQUEST statements about existing beliefs from linked components at each cycle of operation.

PlanBase The plans that the agent has at its disposal are stored in the *PlanBase*, which can accept INFORM statements to update the existing set of plans, and INFORM statements relating to triggers in plans. Based on these triggers, the *PlanBase* fires INFORM statements containing the plans activated by the triggers.

PlanSelector The *PlanSelector* is a controller that selects plans based on current beliefs and relevance. It can then send EXECUTE statements to other components about plans that should be activated.

IntentionBase The *IntentionBase* is an infostore that stores active and inactive intentions. It administers the status of stored intentions based on information received from other components.

IntentionSelection The *IntentionSelection* controller analyses intentions and decides what action to take based on the *body* of the plans. It sends EXECUTE statements to components for actions to be performed, or INFORM statements to other parts of the architecture when internal triggers are matched or when intentions are to be suspended or cancelled.

BDIEventProcessor The *BDIEventProcessor* component is not an intrinsic part of the AgentSpeak(L) architecture. Its task is to translate information received from sensor components into attribute types that are consistent with the rest of the architecture. It converts any information received into the appropriate belief data types so that they can be manipulated by the rest of the architecture. Such a component is required when sensors are not designed to take into consideration the attribute structures of AgentSpeak(L), something that is likely in highly heterogeneous environments with legacy applications.

BDIActionProcessor Similar to the *BDIEventProcessor*, the *BDIActionProcessor* controller translates architecture specific commands for action to a form understandable by the actuators. Once more, this is only necessary if the actuators have not been designed with AgentSpeak(L) in mind.

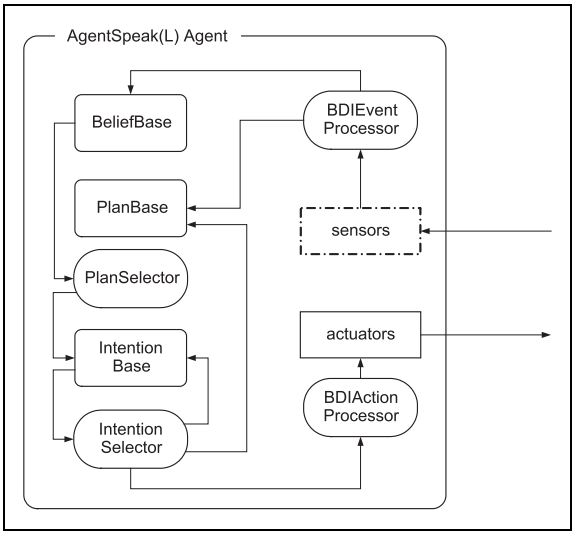

Fig. 8. Architecture components

5.1.2 Behavioural Specification

To a large extent, the structural specification indicates how components should be linked in order for the architecture as a whole to operate in a sensible manner. The behavioural specification, however, gives precise definitions of these links. By separating the two aspects, we gain flexibility in altering the behaviour to the extent that changing links between components allows. In addition, components can be replaced without affecting the rest of the architecture as long as the statements they produce are consistent with what other components can process.

The links between components are illustrated in Figure 8, in which the execution sequence begins with statements arriving at the *BDIEventProcessor* from sensors. These events are translated to triggers and are passed in statements to the *BeliefBase* and the *PlanBase*. The *BeliefBase* is informed of new beliefs, or changes its existing beliefs, while the *PlanBase* retrieves plans that have a trigger equal to the triggers produced by the *BDIEventProcessor*. The *PlanBase* then informs the *PlanSelector* about the plans that have appropriate triggers, and the *BeliefBase* informs the *PlanSelector* about the current beliefs of the agent. Based on this information, and plans that are already activated, the *PlanSelector* chooses the applicable plans and informs the *IntentionBase* of these plans. The *IntentionBase* creates a new intention and sends this intention along with any other pending intentions to the *IntentionSelection* component. In turn, this analyses the intentions and produces the appropriate internal events, which propagate to the *PlanBase* and *IntentionBase*, and sends EXECUTE statements to *BDIActionProcessor* about the actions that need to be performed.

This execution cycle is, of course, only one of several possibilities. The key point to note is that it can be dynamically altered to suit specific situations

as long as the inputs and outputs to components are statements they can process.

6 Discussion

Initial experience with *act*SMART is positive. The BDI architecture described above is readily implementable, and a working prototype system has been developed in exactly the way specified here. In particular, the implementation of the architecture in *act*SMART has provided useful experience as to the suitability of the model for agent construction in an application development setting. The fine-grained control over every aspect of the agent aids significantly in testing and debugging, since components can be tested individually and, more importantly, they can be tested in connection with other components without requiring an instantiation of the entire architecture. Moreover, the state of each component, and the agent as a whole, is clearly defined at each moment, and changes to individual components and to the overall architecture are easy to achieve. Finally, constraining exchanged inter-component information to specific attribute types provides a tight link to the ontology the agent uses to describe the environment. This minimises confusion between different data types, and makes integration of different components simpler.

While our case study of AgentSpeak(L) demonstrates the validity of the approach and highlights its value, more empirical analysis is required, especially on limited devices. In particular, the benefits that are offered come at an increased computational cost in relation to the use of a specific means of exchanging information between components, and throughout an agent. This is clearly not an optimal, nor most cost-effective, means of information exchange between components. Although the consequences of this are still to be completely understood, and although more work is needed, both analytical and experimental (or developmental), initial tests on PCs and high-end PDAs indicate that the difference is not significant.

Through this example, and through others reported elsewhere [15,16] we have shown how the combination of SMART and *act*SMART can be used to support the development process of agent systems by offering three distinct but complementary views. Descriptive specification provides an abstract (and typically formal) means for systems specification, but largely ignores issues of system development. By introducing *act*SMART, however, we complement this with a specification of both the system structure, and its behaviour, at a level that enables ready implementation.

This is a first iteration towards a methodology. We have identified notations and models for use in agent specification and development, and have illustrated

their use in application to the development of a sophisticated agent architecture. Similarly, we have provided a component-based agent system that lends itself to reconfiguration for different platforms, but have only briefly considered how this suggests the use of a library of components, and have not at all considered whether and how agent patterns [17] may be used in this context, which seems almost an ideal combination. However, we have not fully elaborated the *process* of development, which is essential for a mature and industrial-strength contribution to support agent-oriented software engineering. In that respect, much more remains to be done, but the work described here provides a platform on which to base it.

References

[1] N. R. Jennings, On agent-based softare engineering, Artificial Intelligence 117 (2) (2000) 277–296.

[2] J. Bryson, L. A. Stein, Architectures and Idioms: Making Progress in Agent Design, in: C. Castelfranchi, Y. Lespérance (Eds.), Intelligent Agents VII. Agent Theories Architectures and Languages, Vol. 1986, Springer, 2001, pp. 73–88.

[3] M. d'Inverno, M. Luck, Understanding Agent Systems, 2nd Edition, Springer, 2004.

[4] J. Spivey, The Z Notation, 2nd Edition, Prentice Hall, 1992.

[5] J. Cheesman, J. Daniels, UML Components: A Simple Process for Specifying Component-Based Software, Addison-Wesley, 2000.

[6] D. D'Souza, A. Wills, Objects Components and Frameworks with UML, Addison-Wesley, 1998.

[7] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[8] A. S. Rao, M. P. Georgeff, BDI-agents: from theory to practice, in: Proceedings of the First International Conference on Multiagent Systems, AAAI Press/ The MIT Press, 1995, pp. 312–319.

[9] T. Finin, Y. Labrou, J. Mayfield, KQML as an agent communication language, in: J. Bradshaw (Ed.), Software Agents, MIT Press, 1997.

[10] M. d'Inverno, M. Luck, Engineering AgentSpeak(L): A Formal Computational Model, Journal of Logic and Computation 8 (3) (1998) 233–260.

[11] J. Graham, K. Decker, Towards a Distributed Environment-Centered Agent Framework, in: N. Jennings, Y. Lesperance (Eds.), Intelligent Agents VI Agent Theories, Architectures, and Languages, Vol. 1757 of LNCS, Springer, 1999.

[12] A. Ankolenkar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, K. Sycara, DAML-S: Web Service Description for the Semantic Web, in: I. F. Cruz, S. Decker, J. Euzenat, D. L. McGuinness (Eds.), The First Semantic Web Working Symposium, Stanford University, California, 2001, pp. 411–430.

[13] A. S. Rao, AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, in: W. V. de Velde, J. W. Perram (Eds.), Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Vol. 1038 of LNCS, Spinger, 1996, pp. 42–55.

[14] M. Luck, M. d'Inverno, Motivated Behaviour for Goal Adoption, in: C. Zhang, D. Lukose (Eds.), Multi-Agent Systems: Theories, Languages, and Applications, 4th Australian Workshop on Distributed Artificial Intelligence, Vol. 1544 of LNCS, Springer, 1998, pp. 58–73.

[15] R. Ashri, I. Rahwan, M. Luck, Architectures for Negotiating Agents, in: V. Marik, J. Muller, M. Pechoucek (Eds.), Mutli-Agent Systems and Applications III, Vol. 2691 of LNAI, Springer, 2003, pp. 136–146.

[16] R. Ashri, M. Luck, An Agent Construction Model for Ubiquitous Computing Devices, in: P. Giorgini, J. Muller, J. Odell (Eds.), Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering, 2004.

[17] J. Gonzalez-Palacios, M. Luck, A framework for patterns in gaia: A case-study with organisations, in: P. Giorgini, J. Muller, J. Odell (Eds.), Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering, 2004.