Comput Syst Sci & Eng (2005) 4: 273–282

International Journal of
Computer Systems
Science & Engineering

A model driven component-based development framework for agents

Gaya Buddhinath Jayatilleke, Lin Padgham and Michael Winikoff

School of Computer Science and Information Technology, RMIT University, GPO Box 2476V, Melbourne, VIC 3001, Australia
Email: gjayatil, linpa, winikoff@cs.rmit.edu.au

Developing agent-oriented systems is still a difficult task. However, a component-based approach can help by supporting both modular modification of existing systems and construction of new systems from existing parts. In this paper we develop a foundation (conceptual model) for a component-based agent development framework by extending the concepts of the SMART framework. We apply our definitions to an existing agent application in order to both refine the definitions, and to evaluate the extent to which the model is able to support modification of existing systems. A prototype toolkit called Component Agent Framework for non-Experts (CAFnE) that implements the component framework is also explained briefly.

Keywords: agent-oriented systems, component-based approach, CAFnE

1. INTRODUCTION

Agents are a powerful technology with many significant applications, both demonstrated and potential [13, 14]. However, building and modifying agent systems currently requires substantial expertise in one or more agent development platforms. The complex domains where agents are used often have requirements that change as the understanding of the system grows. Agent systems are inherently modular and well suited to the gradual development of the system as specialised situations are recognised and appropriate behaviour specified. Typically domain experts (e.g. meteorologists, scientists, accountants) or users identify and require these changes once the system is deployed. The need for an agent software developer to make these modifications slows down the evolution of the system, as well as increasing the cost. Our vision is to develop a framework that facilitates domain experts themselves making modifications to a deployed system, in order for it to better fit needs which are identified as the system is used.

Our approach is to develop a framework based on agent systems made up of well defined components, where structured support is provided for modifying components, or adding new components in well defined ways. For example, assume that we have a definition for a goal, which is based on a description of the world in terms of a set of attributes. A domain expert wishing to add a new goal could then be presented with a menu of the attributes from which to choose a combination that specifies the goal state. In order to build plans to achieve this goal, the domain expert can be presented with the set of existing actions and (sub)goals which affect the relevant attributes.

The components within our framework must be of sufficiently fine granularity to support the domain experts in modifying an application to provide new or refined functionality. Consequently our view of components is somewhat different to how they are viewed in traditional Object Oriented Software Engineering, where components are viewed as binary units of composition with specified interfaces, which can be used in a suitable component infrastructure [10, 23].

Our components must be easily modifiable (and therefore not binary) building blocks. While our approach may well lead to greater reuse of small pieces of code (our components), supporting reuse is not the primary objective. Supporting a 'component market place' where components are bought and sold in binary form is not a goal of this research.
In order to support the modification of an existing application by domain experts, the application must be built and structured in terms of the relevant components. Many such components (e.g. plans, agents, goals) are an integral part of existing agent development applications, but others (e.g. actions, protocols, environment) are more implicit. Our framework will assist the agent software developer in building well structured applications that lend themselves to ongoing modification as the application develops in scope and complexity. This approach will make the application easier to understand and maintain, even when extensions beyond the scope of domain experts are required.

Our starting point for component definition is the SMART framework of Ashri and Luck [16]. We have modified and extended this framework based on a case study of a meteorological application, typical of the kind of system we aim to support. The framework has been implemented as a workbench which supports the domain expert in making well defined modifications to the application. In the following sections we motivate and explain the details of the components that we support, and provide examples from the application, regarding the kind of changes that can be made by domain experts. The meteorological application has in fact undergone a range of developments since the initial implementation. We will be analysing these in detail and evaluating the extent to which they can be supported within our framework. In future work the workbench will be used and evaluated by domain experts within this domain, as well as used within additional domains.

2. BACKGROUND

Component Based Software Engineering is a well-established technology within object-oriented software engineering [10], and has also been explored in relation to Agent Oriented Software Engineering. However none of the approaches that we have seen have provided the right combination of simplicity and expressivity to support non-experts in modification of evolving agent applications in the way that we envision.

Agent component systems such as PARADIGMA [1], DESIRE [3], JAF [24], Jadex [18] and others [8, 21] have focused on making agent *architectures* modular, rather than on making applications developed using these architectures modular. These approaches make it easier to change the core functionality of the agent architecture such as how the agent selects actions and how the agent perceives the world. However, they do not support domain-oriented structuring and changes of an agent application through components as in developing a weather event monitor agent from existing weather-related components. In other words they are not intended for non-experts. Another common problem faced in using these systems is the loosely defined nature of components such as *actions*. While most of these systems use *actions* as the primitive atomic behaviors used by agents (frequently contained in Plans), they do not clearly define what constitutes an *action* at the implementation level. Another aspect that needs highlighting is the use of XML in our framework and in Jadex. Jadex uses XML in defining the structure of the Java code that implements each component. This is significantly different to our use of XML where it defines the implementation independent (inherent) structure of the component. This allows for easy comprehension for the non-experts while keeping the implementation independent of the component definition.

The SMART framework [16] on which PARADIGMA [1] is based provides an extensive set of components for defining an agent. However we find that some of these are not relevant for our purpose, as they define the underlying infrastructure components such as *AgentPerception* and *AgentState*, rather than those required for a particular application. Other components are not defined in sufficient detail, or in a way that facilitates their use for building and modifying applications. These include components such as *Actions, Plans* and *Events*. We have however taken aspects of the SMART framework as a starting point and further developed or modified these in line with what we perceive as necessary for providing the concrete implementation support desired.

Another category of work has focused on providing agent toolkits with general purpose agent components for expert agent programmers. This category includes toolkits such as ABLE (Agent Building and Learning Environment) [2] and ZEUS [6]. Both these tools provide graphical interfaces with an extensive set of components mostly comprising core processing elements such as communication, learning and planning based on Java. ZEUS provides more support for multi agent communication and collaboration while ABLE specialises in agent learning. These prepackaged components help the rapid development of agent systems via component reuse. However they can only be used by expert programmers who are skilled in object-oriented languages such as Java and agent concepts. Therefore these tools do not provide an answer to our problem of supporting non-expert users.

The only work that we are aware of that develops a component framework aimed at applications, rather than agent architectures, is [9]. This work views each agent as consisting of a number of Activity components, where an activity is basically a tree of Decision components leading to a Behaviour component Although this work is promising, defining an agent with only Activity, Decision and Behavior components seem too limited in implementing complex agent systems. For example, it is not clear how agent beliefs (i.e. agent's view of the world) are maintained and also how proactive (goal-oriented) behavior can be implemented. As a whole, it is not clear how the three component types can be used to implement flexible behaviors as supported by architectures such as Belief Desire Intention (BDI) architecture [19].

The agent model is a natural and intuitive approach for building complex software systems, compared to some of the existing software engineering techniques [14]. However, despite the claimed benefits, agent systems are still regarded as being difficult to develop. By examining existing agent development environments we have identified a number of

key issues that make agent development difficult, especially for domain-experts.

Firstly, some of the detailed concepts behind agents are subtle and are difficult to understand and apply. For example the concept of *Intentions* in BDI theory is not a simple concept that can be grasped by a novice. Similarly advanced features in toolkits, such as meta-plans in JACK, are not easy to comprehend and use.

Secondly, the mapping of concepts to implementations is done in different ways by various platforms, even where the platforms are based on the same agent concepts. For example toolkits such as JACK, Jadex and JAM are all based on the BDI model and provide developers with a language to implement agents. However, each of the toolkits provides different constructs with different semantics. For example, in JACK a plan can have sub-goals, and if one of these fails, then the plan fails. In Jadex, on the other hand, the success or failure of a plan's sub-goals needs to be explicitly checked. One possible reason for this sort of inconsistency between agent platforms is the novelty of the technology and it not being mature enough to be standardized as in Object Oriented concepts. However, this inconsistency may confuse a novice agent developer in selecting and using a toolkit to implement an agent following a particular set of concepts.

Thirdly, an issue more relevant to our goal is the orientation of all existing agent development tools towards expert agent programmers. Most agent languages are based on an existing object oriented language (i.e. a host language) such as Java, requiring the agent developer to not only understand the agent specific constructs but also understand the host language.

Finally, the toolkits do not support the agent concepts in a clearly structured way. For example, in BDI agents, plans are typically thought of as including a sequence of steps, but languages such as JACK, Jadex and JAM do not structure a plan body as a sequence of steps, instead they simply view plan bodies as being code in a suitable programming language. Similarly, existing agent languages do not provide a structured way of modeling the agent's environment.

Therefore our aim is to first come up with a simplified agent model with clear definitions for the constructs within it, hence allowing non-experts to build and modify agent systems in a way closely related to agent concepts (i.e. BDI).

3. APPROACH

In this work we are interested in defining components with the right granularity to support non-expert users with limited programming knowledge in modifying agent behavior. It is essential that the component model support implementation, not just design. In particular, we are interested in implementation using a particular style of agent systems, namely Belief, Desire, Intention (BDI) [19] systems such as JACK [4], PRS [11] and dMARS [7]. In this paper we provide details of a conceptual component model which is a result of our initial work towards achieving the aforementioned goal.

As a starting point we used the SMART framework [16] to identify the potential building blocks of an agent system. The reasons for selecting SMART are two fold. Firstly SMART is precisely defined (using the Z formal specification notation). Secondly SMART has been used to formalise the operation of the dMARS agent platform [7] which suggests that the SMART model is compatible with the use of a BDI implementation platform.

Some of the component types defined in SMART correspond to core functions of an agent platform, for example the action selection function. These components are provided by the agent platform and are not usually modified by application developers. For example, the mechanism by which an agent selects which plan to use or which event to process next will not be changed by the designer of a particular agent application. These 'core platform components' are not the focus of our work since they are not the focus of an application developer, but of a platform developer.

Other component types *are* application-specific. For example, the entities that a particular application domain deals with, or the goals that an agent achieves. These component types are the focus of our investigation. Figure 1 shows an overview of how these descriptive components are structured within an agent.

In order to investigate the suitability of the descriptive components identified in SMART and to find what other component types are required, we applied these to an existing agent application. We used the weather alert system at the Bureau of Meteorology in Melbourne [17] for this purpose. We now briefly describe this application.

3.1 Meteorology alerting system

Currently, human forecasters receive a large range of information from many sources such as radar, automated weather stations, lightning, and volcanic ash advisories. The weather alert system[1] [17] aims to reduce the information overload experienced by human forecasters by filtering information and automatically generating a range of alerts. The system receives events from a range of sources including:

- Automated Weather Stations (AWS) which produce regular readings including temperature, pressure, and wind speed and direction.

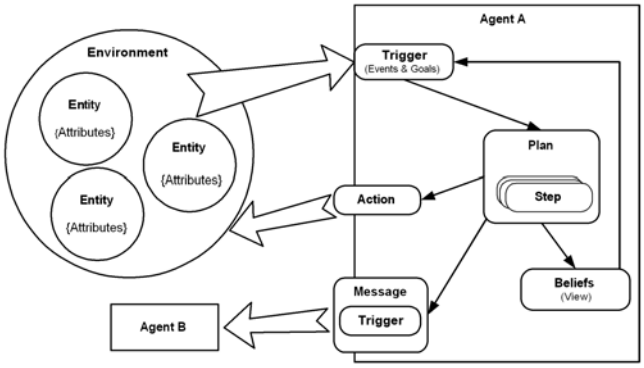

Figure 1 Simplified model of an agent in our framework

1. The weather alert system is part of the research project *Open Agent Architectures for Intelligent Distributed Decision Making* which is funded by the Australian Research Council (Linkage Grant LP0347025, 2003-2005) and is joint work with the Australian Bureau of Meteorology (Australia's national weather service, www.bom.gov.au) and Agent Oriented Software Pty. Ltd. (www.agent-software.com).

- Terminal Aerodrome Forecasts (TAF) which are regularly-issued forecasts for airports, and contain pressure and temperature predictions.
- Thunderstorm alerts.

The system is structured as an open agent system implemented in JACK™ [4] where each agent subscribes to events. For example, one of the agent types subscribes to AWS and TAF events and checks for consistency between the forecasts (TAF) and the actual weather readings (AWS). If a significant inconsistency is detected it sends an alert event. Forecasters' GUIs subscribe to alert events. Currently there are agents that check for inconsistencies between forecasts (TAF) and data readings (AWS) and agents that check for extreme weather conditions (e.g. high wind speed, thunderstorms).

The key issues include ensuring that the system is resilient to various forms of failure (i.e. is robust); reducing 'alert pollution', that is trying to avoid overloading the human forecasters with too many alerts whilst ensuring that essential information is delivered; and ensuring that the system is extensible.

4. COMPONENT MODEL

Our exploration using the weather application yielded a basic set of component types which we believe are sufficient to describe an agent application. This set of components comprise: *attribute, entity, environment, goal, event, trigger, plan, step, belief* and *agent* (see Figure 1).

We follow a layered approach to define our model, similar to the levels M0, M1 and M2 defined in the Model Driven Architecture (MDA) of the Object Management Group (OMG). Figure 2 shows the levels in our model (left side) and examples of entities in each layer (right side). The name of the corresponding MDA layer is given in a circle. It is important to note that we have not defined our levels using the Meta Object Facility (MOF) which the MDA is based on [15]. A rigorous comparison of the MDA levels and our levels is out of the scope of this paper and we leave it as future work.

At the meta-meta level (M2 equivalent) we define the domain-independent generic component types found in our study. These generic types are then used in the meta level (M1 equivalent) to define domain dependent component types. For example, using the M2 level component type *Agent* we can define a *WeatherAnalyzer* type at the M1 level in the meteorology application domain. The M0 level defines

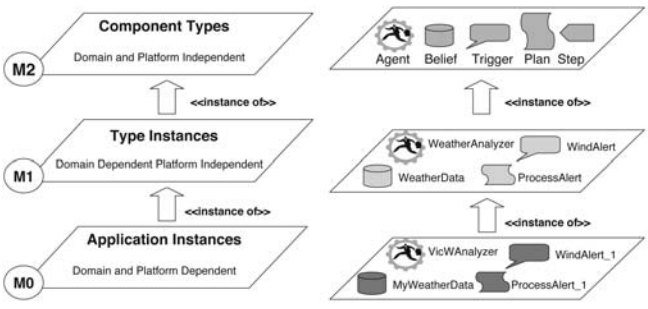

Figure 2 Layered Component Definitions

the runtime components of the system which are bound to a domain as well as a runtime platform. These components are defined based on the M1 level component types using instantiation. For example an instance of the *WeatherAnalyzer* type could be created at runtime as *VicWAnalyzer*.

We use XML as a language for defining the components and hence use an XML DTD[2] specification to provide the meta level definitions of component types. The reasons for selecting XML as the definition language for components are three fold. As stated earlier, our interest is in defining components at a descriptive level rather than at a platform level. For this XML provides an inherently structured language for specifying the anatomy of each component. This is also one of the reasons for deciding against a formal language such as Z ('Zed') [22] (used by SMART for defining its components) which is more suitable for defining process than structure. Secondly, using technologies such as XSLT (XSL-Transformations) it is easier to transform the XML definitions of the components to executable code. This process is shown in Figure 3. The XML component definitions generated by the top level development tool can be converted into executable code of an existing agent programming language by the *Transformation module*. This way we are able to leverage an existing agent platform for executing our component agents. For example, in our initial implementation, we are transforming the component definitions to JACK agent language [4] code. Thirdly, XML is heavily supported by tools and also well established in mainstream software engineering.

In the rest of section 4 we define the component types found in our study.

4.1 Attributes, entities and environments

A defining characteristic of agents is that they are situated in an environment (usually highly dynamic). For example, in the meteorological alerting domain the environment is dynamic in (at least) two ways

1. The Environment generates a continuous flow of sensory

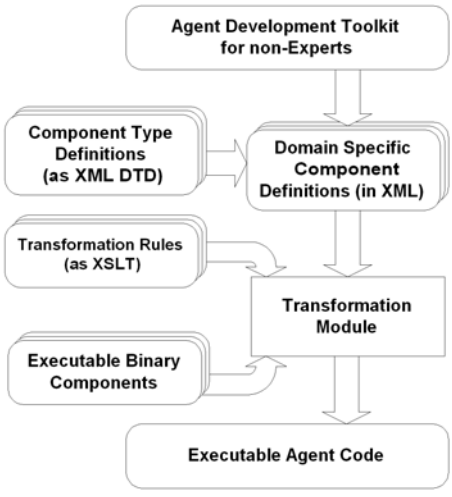

Figure 3 An Overview of the Component Framework

2 Refer to http://www.w3.org/TR/2004/REC-xml-20040204/ for a specification of XML and DTD

data (e.g. weather related readings from automated weather stations).
2. The Environment may add/remove sensory data sources with or without prior notice (e.g. an automated weather station might go down due to a technical fault).

Therefore we need an effective way to model and represent the Environment. The SMART model provides two ways of structuring the environment: as a set of Attributes, where an *Attribute* is defined as a 'perceivable feature'; or as a set of Entities, where an *Entity* is a set of Attributes. Using Entities rather than Attributes gives a more structured view of the environment. However SMART does not provide an explicit definition for an Attribute that can be used in an implementation. Therefore we provide a definition for an Attribute while retaining the model for the Environment as defined in SMART. We define an Attribute type as a tuple:

```
<!ELEMENT Attribute (%ID;)>
<!ATTLIST Attribute Type CDATA #REQUIRED>
<!ENTITY % ID "(#PCDATA)">
```

where *ID* refers to a *unique identifier* used to identify an Attribute. *Type* refers to the domain the Attribute value belongs to. As Attributes are used to hold values and also references to other components, *Type* can contain regular data types such as integer, real etc. and also identifiers of other components such as Entities, Events etc.

Since the environment is dynamic, Attributes are not simply mappings of names to values, but fluents [20] – the value of a given Attribute depends on the situation. Hence, in some situations it may be useful to explicitly attach state references (for example time stamps) to Attributes. However in our initial implementation we have ignored this fact for simplicity.

Using the Attribute definition we can define an Entity as being a collection of Attributes:

```
<!ELEMENT Entity (Attribute)+>
<!ATTLIST Entity ID CDATA #REQUIRED>
```

where *ID* refers to a *unique identifier* used to identify an Entity.

While Environment is not an implementable component in our model, the above definitions show that Attributes and Entities can be used to define the environment an agent is situated in. This helps in defining environment related constructs such as percepts and beliefs.

The meteorology domain has Entities such as Automated Weather Station (AWS) readings which contain Attributes such as temperature, pressure, wind direction and wind speed. Following is an Attribute definition for a temperature reading.

```
<Attribute Type="int">temperature</Attribute>
```

Formally, an AWS reading is a Type Instance of the Entity Component Type (referred to as *awsreading*). The definition for the *awsreading* entity is given below:

```
<Entity ID="awsreading">
  <Attribute Type="String">location</Attribute>
  <Attribute Type="int">temperature</Attribute>
</Entity>
```

At runtime an instance of the *awsreading* entity will have its *location* and *temperature* attributes bound to values (e.g. *location='Melbourne'* and *temperature=18*). The nature of the runtime structure of Attributes depends of the execution platform used to run the application. For example, in a platform based on Java, an Entity may be represented as an Object.

4.2 Goals

Being proactive is an important property of agents and consequently, Goals are a crucial concept for agents [25]. Simply, a Goal can be seen as a set of attributes that describe a desired state of the world.

In addition to the desired state of the world, [26] argues that Goals should include a failure condition which describes when the Goal should be abandoned; as well as an indication of how the Goal could be achieved (the *procedural* aspect of the Goal). However, from an implementation point of view linking Goals to the process of achieving them (called a *Plan*) can lead to problems when a priori knowledge of the relevant plans is not available and the agent has to formulate a plan using a planner. Hence in our model we bind a Plan to a Goal rather than a Goal to a Plan. Based on the above we define a Goal as:

```
<!ELEMENT Goal (Attributes, Success?, Failure?)>
<!ELEMENT Attributes (Attribute+)>
<!ELEMENT Success (#PCDATA)>
<!ELEMENT Failure (#PCDATA)>
<!ATTLIST Goal ID CDATA #REQUIRED>
```

where *ID* is a 'Goal Identifier' used to identify a Goal and *Success* and *Failure* are (optional) boolean expressions formulated with Attributes that state the success and failure conditions of the Goal respectively.

A distinction not made in this definition is between different types of Goals such as achievement and maintenance. A formal definition for a Goal that takes this aspect into consideration is given in [7] in formalizing the dMARS agent architecture. However, we regard the distinction between achievement and maintenance Goals to be a part of the Goal processing rather than to be a part of the Goal definition itself. In other words we are able to define both types of Goals with the definition given above while the processing of the two types would be different. For example the same Goal *g* could be fired as *achieve(g)* and *maintain(g)* where the definition of *g* would be the same (using our definition) while the resulting process would be different.

An example of a Goal from the meteorology application domain is given below. The purpose of the Goal in this case is to find a weather station which is responsible for monitoring a given region. We assume that this mapping is not available in the agent's internal beliefs and that the agent has to consult an external agent or a directory service to obtain this information. Therefore due to the absence of a provider service, the agent might fail in achieving this Goal. The Goal also assumes the existence of an attribute called *GoalSeekTime* and a Beliefset query (see section 4.7) *queryws*. *GoalSeekTime* is a time counter that keeps track of the time since the generation of the Goal and *queryws* queries the agents beliefs for a weather station for a given region.

```
<Goal ID="FindEffectiveWS">
  <Attributes>
    <Attribute Type="STRING">region</Attribute>
  </Attributes>
  <Success> queryws(region) </Success>
  <Failure> goalSeekTime >= 5 </Failure>
</Goal>
```

Based on our definition of a Goal, the above states that the Goal with an identifier *FindEffectiveWS* has a success condition (*Success*) which says that once the Goal is achieved the Belief set query *queryws* should return *true* for the given *region* and a failure condition (*Failure*) that says the Goal needs to be achieved within five seconds.

4.3 Events

An Event is a notification of a certain state of the internal or external environment of the agent. Based on our definitions, an Event type can be defined as:

```
<!ELEMENT Event((Attributes|Entities)+, Step?)>
<!ELEMENT Attributes (Attribute+)>
<!ELEMENT Entities (Entity+)>
<!ELEMENT Step> (See section 4.6 for Step definition)
<!ATTLIST Event ID CDATA #REQUIRED>
```

where the *Attributes* and *Entities* describe the state being notified by the Event and the optional Step (see section 4.6) provides a way to specify a reflexive action. A reflexive action is when an action is executed directly as a result of an Event occurrence without invoking a plan.

An Event is similar to a Goal in that once an Event is generated, it needs to be handled by an Event Handler (similar to a Plan). However an Event describes a current actual situation whereas a Goal describes a desired situation that needs to be brought about. Hence an Event does not state success or failure conditions. Further, by including a single Step, the Event provides a way to execute a mandatory action before any plans are processed. This helps in implementing reflexive behavior and also in simplifying the specification of Event handling when a single action is sufficient to handle the Event. A good example is 'percept events' that are generated as a result of sensing actions of the agent. Most of the percept data are written to the agent's Beliefs before they are handled by any plans. By including a write action in the Event, an agent is able to easily achieve this, without executing any plans.

For example, an Event type instance *awsdatareceipt* is generated when an agent receives data from an Automated Weather Station. This Event is defined below:

```
<Event ID="awsdatareceipt">
  <Attributes>
       <Attribute Type="String">location</Attribute>
       <Attribute Type="int">temperature</Attribute>
       <Attribute Type="int">pressure</Attribute>
  </Attributes>
  <Step ID="add" Type="belief">
    <BeliefSet>awsData</<BeliefSet>
    <Input ID="location" Type="String">thisEvent.
      location</Attribute>
    <Input ID="temperature" Type="int">thisEvent.
      temperature</Attribute>
    <Input ID="pressure" Type="int">thisEvent.
     pressure</Attribute>
  </Step>
</Event>
```

Attributes *location, temperature* and *pressure* define the data being passed on by the *awsdatareceipt* Event. As it is a percept Event that is generated by the environment, we include a step that writes the Event data to a Belief set called *awsData*.

4.4 Triggers

A Trigger is an Event or a Goal, which invokes a plan (see section 4.5 for the definition of a plan).

```
<!ELEMENT Trigger (Goal|Event)>
```

Defining a Trigger allows us to give a common definition to a Plan as a process that handles Triggers (i.e. Events or Goals). However a Trigger is only an addition to our terminology and not a component in its own right. By defining a Trigger we provide flexibility and expressiveness in the modeling process when it is not clear if an Event or a Goal is to invoke a Plan. This can be expressed as a Trigger and later can be implemented as an Event or a Goal.

4.5 Plans

A Plan responds to a predefined Trigger (i.e. achieves a Goal or handles an Event) by sequentially executing a set of 'steps' (a step is a generalized form of an action defined in section 4.6), after checking whether a predefined state, known as the Context is true. The Context is defined with respect to the agent's beliefset about the Environment and the properties of the Trigger being handled by the Plan. This is similar to the notion of a Plan definition given in [25]. Based on this definition we represent a Plan as:

```
<!ELEMENT Plan (Context, Steps)>
<!ELEMENT Context (#PCDATA)>
<!ELEMENT Steps (Step+)>
<!ATTLIST Plan ID CDATA #REQUIRED>
<!ATTLIST Plan Trigger_ID CDATA #REQUIRED>
```

Where *Context* is a boolean expression that specifies the state in which this plan is applicable and *Steps* specify the sequence of steps to be executed by the plan. The parameters *ID* and *Trigger_ID* refer to the identifier of the plan and the identifier of the Trigger (i.e. Goal or Event) being handled by the plan respectively.

Example: The plan type instance *FindAWS* handles the *FindEffectiveWS* Goal defined in section 4.2.

```
<Plan ID="FindAWS" Trigger_ID="FindEffectiveWS">
  <Context> true </Context>
  <Steps>
    <Step ID="QueryAWSServer" Type="action">
        <Input ID="Region" Type="String">thisEvent.
         region</Input>
        <Output ID="WSName" Type="String">
         wsName</Output>
```

```
      </Step>
      <Step ID="condition" Type="logical">
            <Input ID="ConditionString" Type=
             "String">wsName != NULL </Input>
      </Step>
      <Step ID="update" Type="belief">
            <BeliefSet>WSData</BeliefSet>
            <Input ID="Region" Type="String">
             ?thisEvent.region</Input>
            <Input ID="WSName" Type="String">
             wsName</Input>
      </Step>
  </Steps>
</Plan>
```

The action *QueryAWSServer* queries from an external server the Weather Station name (WSName) for the given Region. The second step checks whether an AWS server was found. The third step is an internal beliefset update which updates the agent's belief set with the new data.

4.6 Steps

Actions are defined as primitive elements in many agent platforms and frameworks. In SMART, actions are defined as operations that change the state of the external Environment. However, from an implementation point of view where we consider actions to be the steps of a plan, it is useful to also allow for 'internal actions' that affect the agent's internals rather than its environment. In order to avoid confusion, we retain the term 'action' for an operation that affects the environment, and view actions as being one type of *step*. Some of the other types of steps (apart from actions) include:

- *sense*: changes attributes in the agent's beliefs that reflect the external Environment.
- *trigger*: generates a *Trigger*. Note that the step type *trigger* is not a *Trigger* itself but a process that instantiates a Trigger (i.e. 'trigger' is a verb and 'Trigger' is a noun).
- *message*: this is similar to generating a *Trigger*, however in this case the generated *Trigger* is sent to another agent. This is a very important step type for an agent in a multi agent system.
- *belief*: reads or updates Attribute values in the agent's beliefs. In other words *sense* steps are a special type of belief steps.
- *logical*: These are logical expressions that evaluate to *true* or *false*. This is a special type of step defined to make the plan formulation simpler. This type of step does not change the internal or external environments. By including logical steps an agent can test various conditions within the plan execution and abandon the plan as being failed if a logical step evaluates to *false*.

Based on the above, we attempt to provide an implementation-oriented specification for a *step*. We believe that with the right tool support a non-expert will be able to create plans by combining steps, and modify existing plans that are formulated in terms of collections of steps. A step can be formalized as below:

```
<!ELEMENT Step (Input*, Output*, Instructions,
  Outcome)>
<!ELEMENT Input (Attribute+)>
<!ELEMENT Output (Attribute+)>
<!ELEMENT Instructions (#PCDATA)>
<!ELEMENT Outcome (#PCDATA)>
<!ATTLIST Step ID CDATA #REQUIRED>
<!ATTLIST Step Type (action | sense | trigger | belief
 | logical) #REQUIRED>
```

where *ID* refers to the 'Step Identifier'. *Input* specifies which Attributes are required to carry out the step. *Output* specifies a set of attributes that will be bound to values as a result of the step execution. *Outcome* is a boolean expression based on Attribute/Value pairs that specifies the state of the Environment/View after the execution of the step. In other words *Outcome* can be used to verify whether the step has succeeded or failed. *Instructions* specify the execution process of the step. From an implementation point of view *Instructions* will include the code that implements the step

4.7 Beliefs

An agent's Beliefs are responsible for storing the agent's view of the internal state and the external environments. In their simplest form Beliefs can be represented as a set of Attribute tuples. However in our model, Beliefs are extended further by introducing Keys which make them closer to a *relation* in a relational database model. The relational model is sufficient for most data (belief) descriptions and it is also simple to use. Extending this model to support more complex structures such as storing objects instead of atomic values would be part of our future extensions. The current definition of a Belief set is given below:

```
<!ELEMENT Belief (Attribute+, Keys)>
<!ELEMENT Keys (Key+)>
<!ELEMENT Key (#PCDATA)>
<!ATTLIST Belief ID CDATA #REQUIRED>
```

where the set of *Attributes* (denoted by *Attribute+*) defines a tuple in the Belief set and the *Keys* define a subset of the attributes that acts as a unique identifier for each tuple instance.

Example. The Belief set used in section 4.3 called *awsData* is defined below.

```
<Belief ID="awsData">
   <Attribute Type="DATETIME">timestamp</Attribute>
   <Attribute Type="STRING">location</Attribute>
   <Attribute Type="READING">temperature</Attribute>
   <Attribute Type="READING">pressure</Attribute>
   <Keys>
      <Key> timestamp </Key>
   </Keys>
</Belief>
```

An implementation of a belief set would need to provide a means to *update*, *query* and raise triggers on its beliefs. The *update* and *query* operations can be seen as *belief steps* defined in section 4.6. An *update* step changes the state of the Belief set. A *query* step returns a set of Attribute instances with the values set to values taken from the Belief set. By raising triggers we refer to the ability of the Belief set to generate Events or Goals when predefined conditions are met by the current beliefs. While existing database

concepts can be used to implement these functions, tuple spaces [5] are perhaps more suitable due to the associative nature of data storage and logical querying.

4.8 Agent

An agent in our framework is a collection of Triggers (Goals and Events), Plans and beliefs. In other words, an agent type in our model can be defined as below:

```
<!ELEMENT Agent (Triggers+, Plans+, Beliefs+)>

<!ELEMENT Triggers (Handles*, Posts*)>
<!ELEMENT Handles (Event_ID | Goal_ID)+>
<!ELEMENT Posts (Event_ID | Goal_ID)+>
<!ELEMENT Event_ID (#PCDATA)>
<!ELEMENT Goal_ID (#PCDATA)>
<!ELEMENT Plans (Plan_ID)+>
<!ELEMENT Plan_ID (#PCDATA)>
<!ELEMENT Beliefs (BeliefSet_ID)+>
<!ELEMENT BeliefSet_ID (#PCDATA)>

<!ATTLIST Agent ID CDATA #REQUIRED>
```

The set of triggers includes all the Goals and Events used by the agent. The distinction between the Triggers *handled* and *posted* by the agent could be derived from the Plans used by the agent as only a Plan can handle or post a trigger. However specifying them explicitly helps in detecting errors by cross checking between Triggers specified in the agent definition (intended triggers) and triggers used by the Plans (used triggers).

5. USING THE COMPONENT MODEL

The components identified have been used to model the weather alert system described in section 3.1, in order to evaluate the effectiveness of the framework. We found that structuring the agents as consisting of the defined components made it easy to modify and add new behaviors. These included operations such as adding new plans and creating new triggers which were identified as frequent changes by the meteorologists who use the system.

The component definitions support a well defined structuring of the application. However in order for domain experts to be able to make modifications using the framework it is necessary to provide an interface and toolkit to easily manipulate the components, and to see the available fillers for building or modifying particular parts of the application.

Figure 4 shows the prototype toolkit we have developed, named CAFnE (Component Agent Framework for non-Experts). It supports the user in defining components and integrating them into the application. The toolkit produces the relevant XML definitions, based on input from the GUI. It also includes a Transformer module to transform the platform independent components (defined in XML) to JACK agent language constructs in order to provide an executable application. For more details on the CAFnE toolkit, see [12].

The prototype CAFnE IDE shown in Figure 4 has data from the sample Weather Alert Application we are working

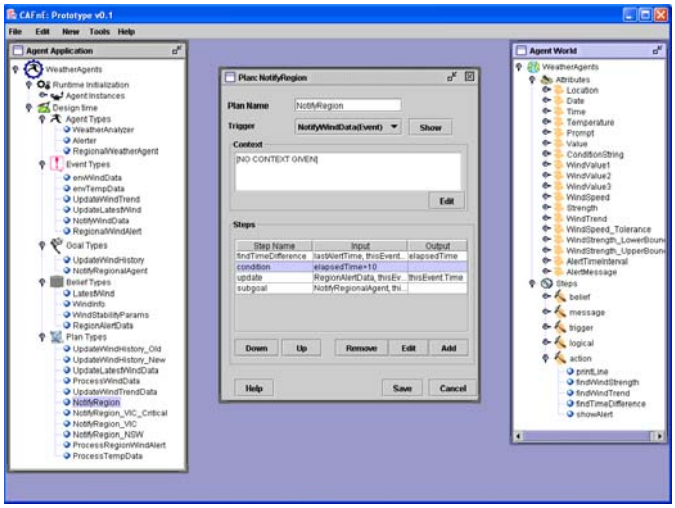

Figure 4 CAFnE Toolkit

with. The tree view window on the left side lists the available component types while the right side window contains the attributes and the steps used in the application domain. It is essential that Attributes and Steps be carefully defined as the other component types are based on one or both of these base component types.

In the rest of the section we briefly explain how the sample weather application is encoded using the component types defined in the previous section and how the CAFnE tool can be used to support the modification process.

A high level view of the sample weather alert system is shown in Figure 5[3] with respect to percepts, agent types and messages passed between the agents. (More detailed views of the internal structure including plans, events and belief sets of each agent are available in separate diagrams of the PDT tool which we plan to fully integrate with CAFnE.) Figure 5 shows that the Agent of type WeatherAnalyzer receives wind speed readings as Event types called envWindData.

The WeatherAnalyzer agent type processes this data by firstly classifying the wind reading as HIGH, MID or LOW based on a set of threshold values stored in a beliefset. Secondly by looking at past wind readings stored in a belief-

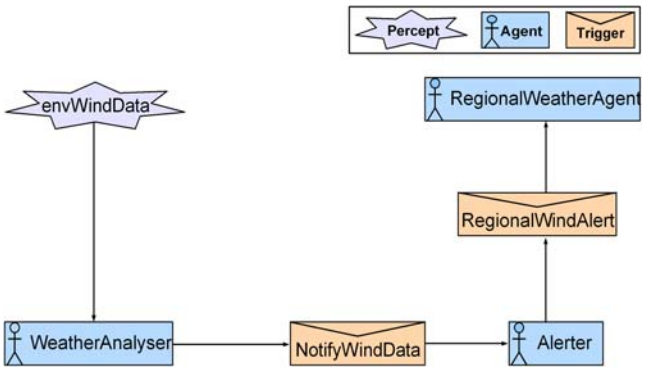

Figure 5 Application Overview

3 Designed using the Prometheus Design Tool (PDT), http://www.cs.rmit.edu.au/agents/pdt

set, it also determines if the wind speed is increasing or decreasing. Finally, it sends these details to Alerter agent type using an Event type NotifyWindData (shown in Figure 5). Based on a notification criteria for each region[4], an Alerter agent potentially notifies agents of type RegionalWeatherAgent of the wind level. For example, an alert is sent to the RegionalWeatherAgent for the Victoria region if there have been no alerts sent in the past 10 minutes and if the wind speed is both HIGH and increasing. Once a regional agent receives an alert it displays the alert for the meteorologist, formatted based on the severity of the alert.

All the component types (i.e. steps, plans, beliefsets, events, goals and agent) required to achieve the above behavior are defined using the appropriate component definitions. The clear structure of the definitions not only guides the developer in structuring the application, but also makes it easier to provide an interface so that the meteorologist can modify these components as needed. For example, the middle window in Figure 4 shows the edit window for the NotifyRegion plan, used by the Alerter agent to respond to the NotifyWindData event. The definition of the NotifyRegion plan is given below:

```
<Plan ID="NotifyRegion" Trigger_ID="NotifyWindData">
  <Context>true</Context>
  <Steps>
    <Step ID="findTimeDifference" Type="action">
       <Input ID="Time1" Type="String">
        lastAlertTime</Input>
       <Input ID="Time2" Type="String">
        thisEvent.Time</Input>
       <Output ID="TimeDiff" Type="int">
        elapsedTime</Output>
    </Step>
    <Step ID="condition" Type="logical">
       <Input ID="ConditionString" Type=
        "String">elapsedTime>10</Input>
    </Step>
    <Step ID="update" Type="belief">
       <BeliefSet>RegionAlertData</BeliefSet>
       <Input ID="Location" Type="String">
        thisEvent.Location</Input>
       <Input ID="Date" Type="String">
        thisEvent.Date</Input>
       <Output ID="Time" Type="String">
        thisEvent.Time</Output>
    </Step>
    <Step ID="subgoal" Type="trigger">
       <Trigger>NotifyRegionalAgent</Trigger>
       <Input ID="Location" Type="String">
        thisEvent.Location</Input>
       <Input ID="Strength" Type="String">
        thisEvent.Strength</Input>
       <Input ID="WindTrend" Type="String">
        thisEvent.WindTrend</Input>
       <Input ID="Date" Type="String">
        thisEvent.Date</Input>
       <Input ID="Time" Type="String">
        thisEvent.Time</Input>
     </Step>
  </Steps>
</Plan>
```

While some of the steps in the plan might not make sense without having a deeper knowledge of the application, the point to note here is that the structure of the plan has allowed us to provide a simple interface to create and edit any plan. This applies to all the component types defined in our framework. For example looking at the second step shown in the plan window of Figure 4, we see that the condition required is that elapsed time is greater than 10 (minutes). If the meteorologist wishes to increase or decrease this, it can be done very simply by changing the value within the tool.

To illustrate a slightly more complex scenario, imagine that the meteorologists decide that they want a more fine grained set of regions than the existing states. All they will need to do is create the required new agent instances, of type RegionalWeatherAgent, by using the AgentInstances choice at the top of the left hand menu. They can then edit these to differ from the current state based RegionalWeatherAgent agents, by editing the relevant information such as Location. If a slightly different notification behaviour is required, compared to the agent used as a template, it is simply necessary to create a new plan type (say NotifyRegion_Melbourne) to replace the existing one (e.g. NotifyRegion_VIC) that was inherited from the template, and to change the necessary details such as alert frequency, situations to alert on, and so on.

The key to being able to provide an interface to do this, that shows the application specific details, but does not need to be developed for each application, is the component structure based on application specific Attributes and Steps, together with generic relationships between components (e.g. a plan consists of a Context, which is a boolean condition based on values of Attributes, and a sequence of Steps).

Our initial explorations indicate that the framework is quite successful for both building and representing a typical application, and for enabling the type of modifications needed by domain experts. When we have further refined the interface of the toolkit, and added some additional component types, we will be testing usability directly with meteorologists as a typical target group.

6. CONCLUSIONS AND FUTURE WORK

This paper described a conceptual framework of domain independent component types that can be used to formulate and modify an agent system. We used SMART concepts as a starting point in analyzing an existing agent application for generating weather alerts. This process yielded a set of component types, namely *attribute, entity, environment, trigger (goal, event), plan, step, belief* and *agent* that can be used to build an agent system. While some of these are identified as building blocks in SMART, the definitions and usage are different in our framework. For example a *plan* is not a basic construct in SMART. Another difference is the definition of an *action* in specifying a *step*. We have attempted to define our components to make the description of an agent simpler from an implementation perspective. This is one of the key differences between our work and some of the other work on component based agents. Some of the key issues yet to be developed in our work include a structured representation for condition statements (as in Plan Context), a richer plan language (set of operators for steps) and a representation for

4 The regions are the various states of Australia.

the conversation protocols, or sets of messages that are sent between agents for a particular purpose.

As a continuation of this work we intend to provide a 'Transformation Module' for at least one additional platform (thus demonstrating platform independence), to further refine some of the components, such as Context, and to add components to capture the interactions between agents. This would include ways to design and implement agent protocols. Another area we are interested in is looking at ways to visualise and assess the effects of a change made to the system. This will help the non-Experts in evaluating their changes and minimise errors. We also plan to improve the user interface and evaluate the approach with a user group of meteorologists by asking them to make the changes which we know other meteorologists actually requested within this application as it has developed over time. If they are successfully able to make such changes, without any intervention from a programmer, then we will consider the approach to be highly successful. Initial indications are that this seems very possible.

ACKNOWLEDGEMENTS

This work was supported by the Australian Research Council (Linkage Grants LP0347025 and LP0453486) in collaboration with the Australian Bureau of Meteorology and Agent Oriented Software Pty Ltd.

REFERENCES

1 **R. Ashri, M. Luck and M. d'Inverno.** Infrastructure support for agent-based development.*Foundations and Applications of Multi-Agent Systems*, LNAI2403, pages 73–88, 2002.

2 **J. Bigus, D. Schlosnagle, J. Pilgrim, W. Mills and Y. Diao.** ABLE: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.

3 **F. M. Brazier, C. M. Jonker and J. Treur.** Principles of component-based design of intelligent agents. *Data Knowledge Engineering*, 41(1):1–27, 2002.

4 **P. Busetta, R. Ronnquist, A. Hodgson and A. Lucas.** JACK Intelligent Agents – Components for Intelligent Agents in Java. *Technical report*, Agent Oriented Software Pty. Ltd, Melbourne, Australia, 1998. Available from http://www.agent-software.com.

5 **N. Carriero and D. Gelernter.** Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

6 **J. Collis and D. Ndumu.** The zeus agent building toolkit: Zeus technical manual (release 1.0). *Technical report*, British Telecommunications PLC, 1999.

7 **M. d'Inverno, D. Kinny, M. Luck and M. Wooldridge.** A formal specification of dMARS. In M. Singh, A. Rao, and M. Wooldridge, editors, Intelligent Agents IV: *Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages*, pages 155–176. Springer-Verlag LNAI 1365, 1998.

8 **K. Erol, J. Lang and R. Levy.** Designing agents from reusable components. In *Proceedings of the Fourth International Conference on Autonomous Agents*. Barcelona, Spain, 2000.

9 **H. J. Goradia and J.M. Vidal.** Building blocks for agent design. In *Fourth International Workshop on AOSE*, pages 17–30. AAMAS03, July 2003.

10 **G. T. Heineman and W. T. Council.** *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Publishing Company, ISBN: 0-201-70485-4, 2001.

11 **F. F. Ingrand, M. P. Georgeff and A. S. Rao.** An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.

12 **G. Jayatilleke, L. Padgham, and M. Winikoff.** Component Agent Framework for non-Experts (CAFnE) Toolkit. In M. Calisti, M. Klusch, and R. Unland, editors, (to appear in) *Agent Prototypes and Applications, Whitestein Series in Software Agent Technologies*. Whitestein, 2005.

13 **N. Jennings and M. Wooldridge.** Applications of intelligent agents. In N. R. Jennings and M. J. Wooldridge, editors, *Agent Technology: Foundations, Applications, and Markets*, chapter 1, pages 3–28. Springer, 1998.

14 **N. R. Jennings.** An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, April 2001.

15 **A. Kleppe, J. Warmer and W. Bast.** MDA Explained, *The Model Driven Architecture: Practice and Promise*. Addison-Wesley Publishing Company, ISBN: 0-321-19442-X, 2003.

16 **M. Luck and M. d'Inverno.** *Understanding Agent Systems*. Springer, ISBN 3540419756, 2001.

17 **I. Mathieson, S. Dance, L. Padgham, M. Gorman, and M. Winikoff.** An open meteorological alerting system: Issues and solutions. In *Proceedings of the 27th Australasian Computer Scienc Conference*, Dunedin, New Zealand, Jan. 2004.

18 **A. Pokahr and L. Braubach.** Jadex: User guide (release 0.92).*Technical report*, Distributed Systems Group, University of Hamburg, Germany, 05 2004.

19 **A. S. Rao and M. P. Georgeff.** BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*. San Francisco, 1995.

20 **S. Russell and P. Norvig.** *Artificial Intelligence A Modern Approach*. Prentice Hall, ISBN 0 13 080302 2, 2003.

21 **N. Skarmeas and K. L. Clark.** Component based agent construction. *International Journal on Artificial Intelligence Tools*, 11(1):139–163, 2002.

22 **J. M. Spivey.** *The Z Notation: A Z Reference Manual*. Prentice Hall International, 1989.

23 **C. Szyperski.** *Component Software: Beyond Object Oriented Programming*. Addison-Wesley Publishing Company, ISBN: 0-201-17888-5, 1998.

24 **T. Wagner, B. Horling, V. Lesser, J. Phelps and V. Guralnik.** The Struggle for Reuse: Pros and Cons of Generalization in TÆMS and its Impact on Technology Transition. *Proceedings of the ISCA 12th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2003)*, July 2003.

25 **M. Winikoff, L. Padgham and J. Harland.** Simplifying the development of intelligent agents. In *proceedings of the 14th Australian Joint Conference on Artificial Intelligence (AI'01)*, pages 557–568. Adelaide, 2001.

26 **M. Winikoff, L. Padgham, J. Harland and J. Thangarajah.** Declarative & procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, Apr. 2002.