Development of a Java-based Unified and Flexible Natural Language Discourse System

Philip Hanna Ian O'Neill Darryl Stewart Behrang Qasemizadeh

The Institute of Electronics, Communications and Information Technology
Queen's University Belfast
Belfast, BT71NN, UK
{ p.hanna, i.oneill, dw.stewart, b.qasemizadeh }@qub.ac.uk

ABSTRACT

This paper outlines the design and development of a Java-based, unified and flexible natural language dialogue system that enables users to interact using natural language, e.g. speech. A number of software development issues are considered with the aim of designing an architecture that enables different discourse components to be readily and flexibly combined in a manner that permits information to be easily shared. Use of XML schemas assists this component interaction. The paper describes how a range of Java language features were employed to support the development of the architecture, providing an illustration of how a modern programming language makes tractable the development of a complex dialogue system.

Categories and Subject Descriptors

I.2.7 [Artificial Intelligence]: Natural Language Processing—Discourse, Speech recognition and synthesis; D.2.11 [Software Architectures]: Domain-specific architectures

General Terms

Design; Human Factors

Keywords

Human Computer Interaction; Spoken Dialogue Systems; Dialogue Management

1. INTRODUCTION

The construction of advanced natural language dialogue systems that offer a more natural form of human-computer interaction represents a key strand of research into intelligent user interfaces.

Figure 1 illustrates the cyclic operation of a typical natural language dialogue system. Input is received aurally from the user and transformed into a corresponding word sequence using a speech recogniser (SR) before then being semantically 'understood' within a natural language parser (NLU). The semantic parse is fed into a dialogue manager (DM) that integrates the semantic parse within some defined task. The dialogue manager outputs an appropriate semantic reply which is then transformed into natural language using a generator (NLG) and finally synthesised (SS) as audio or displayed within some form of GUI. Depending upon the supported input modalities, the NLU component may also receive text that has been typed by the user. An overview of key research issues in this area can be found in [1].

Whilst advanced natural language dialogue systems offer the possibility of profound change with regard to how we use a multitude of different devices, such as mobile phones, or interact within environments such as living spaces, cars, etc., the construction of state of the art dialogue systems must tackle a number of significant development issues [2].

In particular, developers must often deal with a heterogeneous software environment consisting of a mixture of different software components, some of which will be knowledge-based with autonomous reasoning capabilities, developed using different languages and potentially running across a number of distributed platforms. Within this environment there is a need to ensure that components can be customised and new functionality integrated. The system must also be accessible to the different types of developer (speech technologist, linguist, dialogue modeller, etc.) and provide adequate monitoring and reporting capabilities.

Most current research towards creating an architecture for a natural language dialogue system has centred on developing a flexible framework that can be used to link together discourse components [3]-[6]. Typically such architectures provide a defined pipeline through which input is evolved, passing from one component to the next, towards an output. This provides loose component coupling, and hence flexibility and ease of integration.

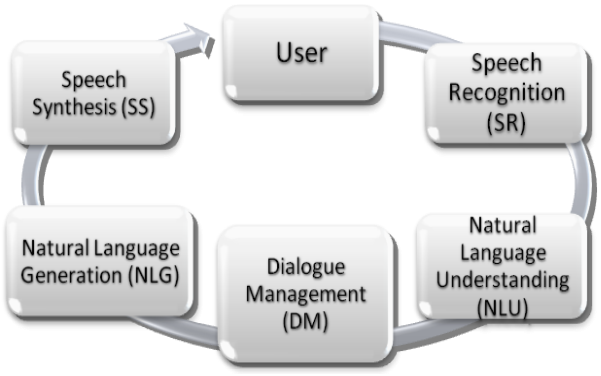

Figure 1. Common components within an NL dialogue system

PPPJ '09, August 27-28, 2009, Calgary, Alberta, Canada.

A challenge inherent in this architecture concerns how information can be shared between components. Whilst it is possible to regard the output from one component as the input to the next, effective operation is often dependent upon other components. For example, how a recognised word sequence should be semantically interpreted may depend upon the nature of the last question asked by the system or the acoustic word confidences within the speech recogniser. As such, dialogue effectiveness and performance can be improved if the minimal set of information passed between components as inputs and outputs is augmented with a wider pool of shared information.

This paper provides selected details of the development of one such approach, developed using Java, which retains the benefits of loose component coupling whilst enabling each component to easily query and understand the operation of other components. The architecture has been named QuADS (**Qu**een's **A**dvanced **D**ialogue **S**ystem). The paper shows how the language features and libraries of Java were employed to make possible an efficient, flexible and robust implementation of the proposed architecture.

The paper is structured as follows: Section 2 provides a brief summary of existing relevant research; Section 3 explores development issues concerning a unified and shared discourse model; Section 4 outlines the integration of the model into a flexible, agent-based architecture; Section 5 explores a number of other development areas in which Java's capabilities were usefully employed; Section 6 offers a number of conclusions.

2. CURRENT ARCHITECTURES

A number of notable dialogue architectures have been developed with a view to improving the flexibility, extensibility and reusability of natural language dialogue systems.

Examples of systems that define an architecture into which established dialogue components can be easily integrated include Olympus [3], derived from the CMU Communicator project [4], which provides a pipeline capturing the logical flow of information within the system and enables components to communicate through a centralised message-passing infrastructure. Another example is TRINDIKIT [5] which offers developers a set of interconnected tools for building information state dialogues. Other approaches adopted by researchers include the development of dialogue architectures motivated towards ease of prototyping or development. For example, the Dialogue Prototyping Equipment & Resources (DIPPER) system [6] offers a number of interfaces to established natural language dialogue components, thereby permitting rapid prototyping.

Of particular relevance to this paper is the JASPIS [7] architecture which utilises an agent-manager-evaluator model of interaction. A central manager holds a shared information store and is connected to other managers through a star-based topology. As such, components are effectively stateless, with the discourse state held centrally. The information store makes use of blackboards and databases as a means of holding information. The precise information structure is domain dependent and consequently is defined and constructed for each developed application.

Finally, a platform that has benefitted from wide commercial uptake alongside offering a vehicle for dialogue research is VoiceXML [8] which defines the W3C standard for interactive voice dialogues between a user and system. However, VoiceXML does not readily permit certain forms of flexible dialogue to be modelled.

3. DEVELOPMENT OF A UNIFIED AND SHARED DISCOURSE MODEL

3.1 Goal of a unified and shared model

Information flow within a spoken dialogue system can be largely modelled as transformational (i.e. SR→NLU→DM→NLG→SS). Each process is typically dependent upon the established discourse context and recent discourse history. For example, both speech recognition and natural language understanding can be improved if the system's last response is used to predict what the user might say next. However, whilst there are clear advantages in sharing information, the standalone nature of most components within a typical dialogue system means that information flow is mostly one-way, with limited information sharing.

One of the key goals of the QuADS architecture was to construct a single discourse model that could be shared between all dialogue components, as well as providing a number of complete/partial views into the discourse product that components may employ in order to share and access information.

3.2 Developing a unified discourse model

In order to develop a unified discourse model that could be shared, the inputs and outputs of each component were defined, alongside the key internal data structures typically maintained by each component. All identified data was decomposed until expressed as structured primitive data. Following this, the various data structures were fused together into a single model. An overview of the developed unified discourse model can be seen in Figure 2.

The identified *discourse elements* are as follows:

- **Item:** An `Item` element encapsulates some quantum of information within the dialogue system (e.g. holding details of a train journey, a recognised word sequence, or an objective or target to be realised). As such, items provide the fundamental modelling element by which the goals, processes, procedures and knowledge associated with a discourse task can be modelled within the system.
- **State:** This element encapsulates one or more `Item` elements into a single collection, thereby modelling the collective state of some form of interaction or process.
- **Act:** An `Act` element encapsulates any action that operates upon `Item` elements. As such, acts may introduce, modify or remove items. The actions embodied within an `Act` element may be inter-object or intra-object.
- **Step:** This element provides a means of encapsulating one or more `Act` elements into a larger, more meaningful, quantum of defined interaction.
- **Product:** This element encapsulates the total evolution of the discourse between the user and system. A product can be defined as consisting of a series of `State` elements, representing the state of the dialogue at discrete points in time, alongside a corresponding series of `Step` elements representing the user and system generated actions from which the discourse is evolved. A combined state and associated set of actions is known as a *turn*.

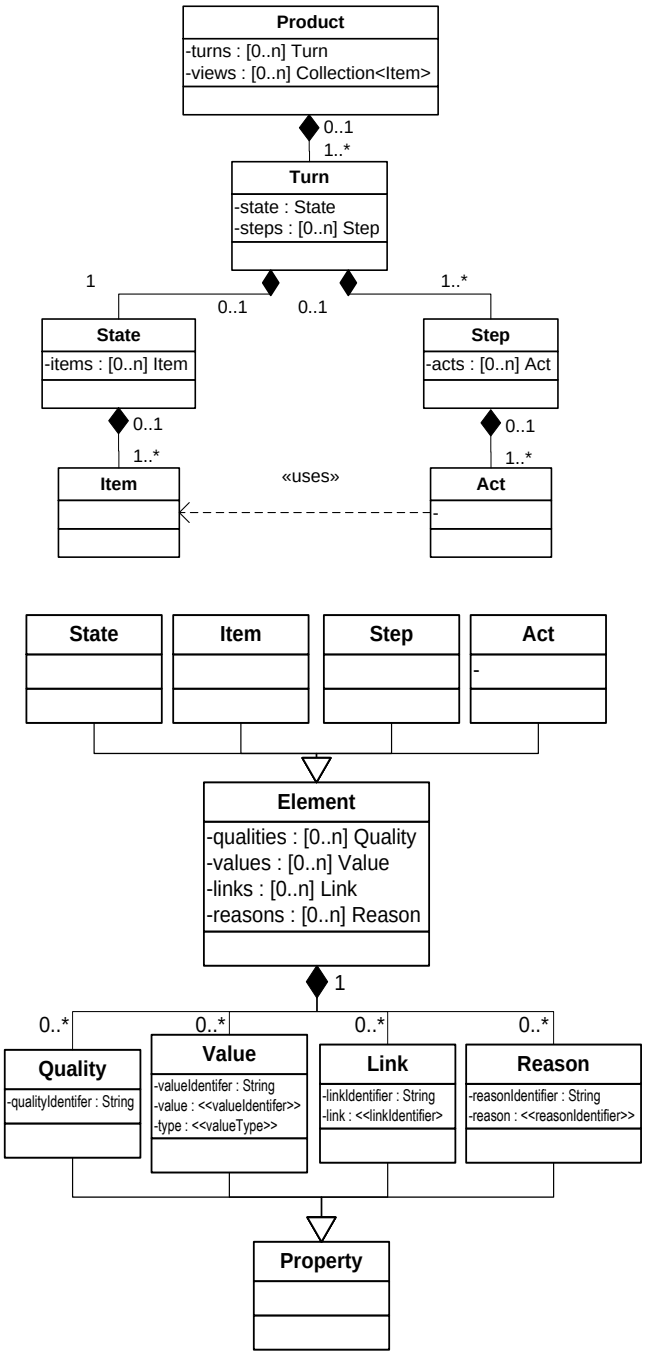

Figure 2. Core class relationships within the developed discourse model

In order to promote model flexibility and extensibility, `Item`, `Act`, `State` and `Step` instances are considered to be refinements of a more fundamental `Element` type. Each element (and hence each act, item, etc.) is defined as consisting of four different property type bundles, namely:

- **Qualities:** A quality expresses a particular feature or characteristic that is assumed to be inherent to the element (i.e. providing a richer definition of the element type).
- **Values:** A value holds a quantity or value associated with the element. It differs from a quality in that value changes do not change the nature, or conceptual type, of the element.
- **Links:** A link provides a means of connecting two elements together in some defined manner.
- **Reasons:** A reason records a justification for the existence of the given element. It provides a means of engaging in meta-discourse reasoning about items, acts, etc.

The structure outlined above provides a flexible discourse model that can be potentially shared across different discourse components. The manner in which the product is shared is outlined in Section 3.3.

3.2.1 Use of Java in developing a unified model

The creation of the model outlined in Figure 2 relied heavily upon the use of Java's object-oriented capabilities through the construction of classes and the inheritance/encapsulation of behaviour. Additionally, the formation and manipulation of element qualities, values, links and reasons extensively employed Java's generic collections. The key software engineering issue that was encountered concerned how *discourse* elements could be best represented and managed.

A discourse element is similar to a software object in that it can encapsulate a number of properties and define forms of behaviour. Additionally, discourse elements can be refined into more specialised types of element. For example, a discourse element representing the notion of agreement can be refined into separate `Agree` and `Disagree` acts. However, discourse elements must model the conventions and usage assumptions adopted within person-to-person conversations. This introduces a number of challenges with regard to the way in which discourse elements are evolved and linked to other elements, as explored next.

3.2.1.1 Element Identification

Discourse elements will typically evolve and change over time. In addition, the types of link between elements may also be dependent upon the evolution of the elements, i.e. in some cases linkage will be temporally dependent and in other cases independent. For example, a hotel `RoomBooking` discourse item will likely evolve over the duration of a conversation as details of the booking, such as arrival date, etc., are provided. A discourse act to `Confirm` the properties of the `RoomBooking` item will refer to the state of the item at a defined instance in time. If, at some later point in the dialogue, it becomes necessary to query what has been confirmed with the user, the `RoomBooking` state at the point of confirmation must be queried. In contrast, a `HolidayBooking` item which links to the `RoomBooking` item will likely wish to refer to the most current state of the room booking.

In order to realise the above structure within Java, each element was assigned two identifiers: a numeric `uniqueElementID`, which uniquely identifies a particular type of element at a particular instance in time, and a string `identifier` which uniquely identifies an element but includes all instances of that element as evolved over time.

3.2.1.2 Element Evolution

The discourse product is evolved on a turn-by-turn basis as information is received from either the user or discourse components that add to, modify, or negate existing information.

As noted, previous discourse turns are stored as part of a discourse history, thereby permitting the evolving nature of the dialogue to be modelled and queried. In order to evolve the discourse product the most recent turn *n* is initially cloned and then evolved using input and processes occurring during turn *n+1* to form the completed turn.

The evolution process provides an example of a tension, frequently encountered during the design of the dialogue system, concerning how a base set of behaviour might be developed that would offer the type of functionality needed by most dialogue developers whilst permitting customised behaviour to be easily introduced. In particular, the evolve process should take into account how discourse elements 'age' over successive turns. For example, an `Offer` act may become less relevant if, over time, it remains unaddressed by the user. In order to provide flexibility, the reflection capabilities of Java were employed within the `Element` class to provide a generic *inheritable* approach that enables extending classes to be flexibly cloned and evolved, i.e.:

```
public Element clone() {
   try {
      // Create new refined element type
      Constructor elementConstructor =
         this.getClass().getConstructor(
            Class.forName( "java.lang.String" ) );

      Element clone =
         (Element)elementConstructor.newInstance(
            this.getIdentifier() );

      // Copy common element bundles
      clone.clonePropertiesFrom(this);

   } [[Catch block omitted]]

   return clone;
}
```

The `clonePropertiesFrom` method was structured to provide a default cloning process with suitable attachment points whereby properties can be evolved in a customised manner if desired.

```
protected void clonePropertiesFrom(Element target)
{
   // Evolve all defined values
   for(String valueId : target.values.keySet()) {

      // Retrieve and add the value type
      String valueType =
         target.valueTypes.get(valueId);
      valueTypes.put(valueId, valueType);

      // Retrieve the value object
      Object value = target.values.get(valueId);

      try {
         // Value specific turn evolve
         Object evolvedValue =
            valueEvolve( valueType, value);

         // Ensure namespace update
         addValue(
            valueId, valueType, evolvedValue);

      } [[Catch block omitted]]
   }

   [[Similar code for evolving qualities, etc.
     omitted]]
}
```

3.2.1.3 Element Reference

As noted in section 3.2.1.1, every discourse element is assigned a unique identifying name that is persistent over the evolution of the dialogue. The turn-by-turn evolution of discourse elements entails that components must be provided with a straightforward means of mapping a particular element identifier onto the most recent evolution of that particular element.

In addition, element identifiers are also used by discourse components to access information maintained by other components. For example, if a natural language component wishes to access the most recent dialogue act output by a dialogue manager component, then, it is reasonable to assume that the NL component can understand the types of act output from the DM. However, it is not reasonable to assume that the NL component will have an extensive understanding of the internal structure within the DM. As such, it is desirable that components can refer to named discourse elements without requiring precise knowledge of where that element is located.

The indirect mapping from identifier name to `Element` instance was made possible through the construction of namespaces combined with a number of resolve (or mapping) algorithms. Each `Element` instance maintains a namespace of all contained elements (e.g. the namespace for a `State` contains all named items stored within that state, whilst the namespace for an `Act` contains all the named values, links, etc. contained within that act). Retrieval of a named element is accomplished using one of a number of defined resolve methods. Each resolve method embodies a particular search strategy that attempts to map provided identifiers onto an appropriate `Element` instance or named element quality, value, link or reason. Consider:

```
public Element resolveElementId(
   Element context, String elementId ) {
   Element element = null;

   do {
      // Extract fragment from element descriptor
      String fragmentId = [[Code omitted]]

      // Product resolve
      if( assembly.getProducts()
            .containsKey(fragmentId) ) {
      element = assembly.getProducts()
          .get(fragmentId).getCurrentTurnState();

      // Link resolve
      else if( context.hasLink(fragmentId) ) {
         element = resolveElementIdWithinProduct(
           context, context.getLinks(fragmentId));
         context = element;
      }

      // Registry resolve
      else {
        element=resolveElementIdWithinRegistries(
            context, fragmentId );
        context = element;
      }

    // Repeat whilst more fragments remain
    } while( [[More fragments remain]] )

    return element;
}
```

For example, a request of `resolveElementId( null, "HotelBooking.ArrivalPeriod.Date" )` might result in 'HotelBooking' being mapped onto an `HotelBooking` item within the current discourse product (assumed default context) with 'ArrivalPeriod' next mapped onto a correspondingly named linked item within the `Hotelbooking` item, before finally mapping "Date" onto a `Date` value within the linked `ArrivalPeriod` item.

The use of namespaces and the resolve process entails that a dialogue component need not be concerned with the underlying structure of the discourse product, nor the organisation and evolution of elements managed by other components. It does, however, assume that discourse elements share a common set of descriptive labels across discourse components, thereby enabling one component to search for an element of interest that is managed by another component.

3.3 Developing a shared discourse model

In order to be useable, each component needed a means of viewing, modifying and extending the discourse product using conceptual constructs and notions appropriate to that component. For example, whilst both speech recognition and natural language generation components can share the same dialogue product, each component will view and change the dialogue product in terms of the acts and items that make 'sense' to that component.

In order to permit components to impose different views upon the unified discourse model, a number of high-level, component-specific views of the basic underlying discourse model were defined. The high-level views were realised by introducing classes that extend the base `Act` and `Item` classes and that are further defined within a number of XML schemas. In particular, each XML schema provides details of how the basic underlying structure can be mapped to a component-specific interpretation.

For example, the `Item` element is a core object within the unified model and is defined in terms of property bundles of qualities, values, links and reasons. Discourse components dealing with natural language can make use of a `Word` element, which is defined as a refinement of the `Item` element. In particular, it is defined to hold a 'Manner' element quality (holding any recognised prosodic information) alongside 'Word' and 'Confidence' element values (holding the word string and associated recognition confidence). As such, the `Word` element is a type of `Item` with pre-defined qualities, values, etc.

A total of four base schemas were defined. At the lowest level, a `CoreSchema` provides a schema-based definition of the proposed unified discourse model. The core schema is extended by two schemas, namely an input/output schema (`IOSchema`) and a problem-solving schema (`PSSchema`). The `IOSchema` provides the basic input/output interface between the different IO modalities that surround the system and the principle recognition and generation managers. The `IOSchema` refines the core schema, making available items such as a `WordSequence`, `WordLattice`, `ImageURI`, etc. and acts such as `TypedInputAct`, `SpokenOutputAct`, etc.

In turn, the `PSSchema` defines the core semantic constructions that inputs are mapped onto and outputs are generated from. The problem-solving schema is derived from that detailed by Blaylock [9] as a means of modelling and evolving discourse problems. In particular, an `Item` is refined to encompass an `Objective` (things to accomplish), a `Recipe` (planned sequences of action) and a `Resource` (utilised items) alongside corresponding acts.

The final developed base schema, namely the discourse management schema (`DMSchema`), refines the `PSSchema` to introduce the discourse specific acts that underpin most current models of dialogue. The schema was, in part, derived from the DIT dialogue act hierarchy developed by Bunt [10]. Figure 3 provides a snapshot of top-level acts defined within the hierarchy,

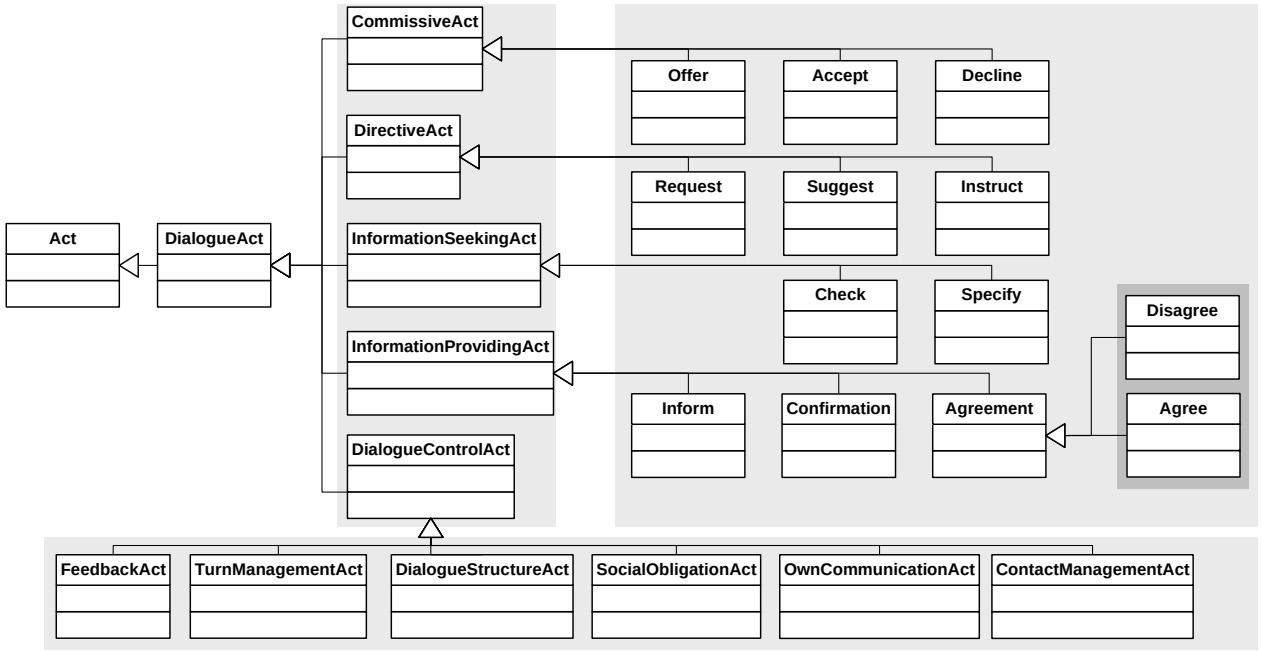

Figure 3. Top level discourse acts and a selection of refined acts defined within the DMSchema

alongside some of the more refined discourse acts. In brief: commissive acts deal with the commission of action such as offering and agreeing to some plan of action; directive acts deal with the direction of action, such as suggesting a certain form of action; information-seeking and -providing acts deal with the identification and supply of information in response to directives. Finally, dialogue control acts help manage the often uncertain, turn-by-turn, changing nature of the dialogue.

Importantly, any two components which 'understand' a particular defined schema can interact with one another using the acts and items defined within that schema. In effect, each schema provides an interpretational window onto the underlying unified discourse model. Hence, different discourse components can readily access and query the workings of other components, including selectively using a defined schema to retrieve certain key items of information. For example, a natural language understanding component implementing the `IOSchema` can query the acoustic and word likelihoods assigned to a recognised input, as stored by a speech recogniser, or, through implementing the `DMSchema`, query the dialogue acts output from the dialogue manager. This is in contrast to a traditional dialogue management system where each component typically has little or no access to other components other than through normal IO progression.

3.3.1 Use of Java in developing a shared model

Java's extensive XML support offered a means of parsing input XML and generated output XML in agreement with the defined schemas. The core problem encountered in realising the shared model concerned how best to enable dialogue developers to easily and straightforwardly extend the provided schemas to include refined items, acts, etc. In particular, the provided functionality should enable dialogue developers to easily construct and deconstruct refined Element instances from/to a corresponding XML description. Additionally, conformity checking should be embedded to ensure that both extending schemas and XML input conform to the requirements of the defined base schemas.

Through the use of Java's `Validator` and `Schema` instances it was readily possible to verify that the source XML met the specification defined within the relevant schema, thereby significantly reducing the amount of code that was needed to validate the process of construction and deconstruction.

The introduction of straightforward element-to-XML construction and deconstruction relied upon the design of the `Element` class. In particular, all classes extending `Element` are required to be defined using the quality, value, link and reason property bundles which are inherited from the base `Element` class. This dependency meant that it was possible for the `Element` class to offer the following two methods as a generic means of handling the process of construction and deconstruction.

```
public static Element buildFromXML(
   Node sourceNode ) throws AMDSException

public abstract org.w3c.dom.Element buildAsXML(
   Document document ) throws AMDSException
```

Provided discourse elements are defined in terms of the property bundles defined within the `Element` class, then the inherited XML functionality will enable refined `Element` instances to be correctly constructed and deconstructed. However, in order to ensure that objects constructed from an XML description are correctly instanced (i.e. an XML description of a `SomeItem` should be realised as an instance of `SomeItem`) it was necessary to ensure that the `Element` XML methods were structured to make use of Java's reflection capabilities, e.g.:

```
public void addValue(
   Node sourceNode, Element targetElement )
{
   try {
      // Extract added object id and type
      NamedNodeMap attributes =
         sourceNode.getAttributes();
      String id = attributes.
         getNamedItem("id").getNodeValue();
      String type = attributes.
         getNamedItem("type").getNodeValue();

      // Build and add object
      Object value = buildObject(type, attributes.
         getNamedItem("value").getNodeValue());
      targetElement.addValue(id, type, value);
   } [[Catch block omitted]]
}

public Object buildObject(
   String className, String constructorParameter )
{
   Object targetObject = null;
   try {
     // Build requested object using parameter
     Class targetClass = Class.forName(className);
     Constructor targetClassConstructor =
        targetClass.getConstructor(
        Class.forName("java.lang.String"));
     targetObject =
        targetClassConstructor.newInstance(
           constructorParameter );
   } [[Catch block omitted]]

   return targetObject;
}
```

4. EXTENSIBLE AND CONFIGURABLE ARCHITECTURE

As noted in Section 1, one of the objectives within spoken dialogue system research is to develop models of discourse that can be easily and flexibly adapted to different problem domains. In order to accomplish this, it is necessary to develop a system that encapsulates and shares domain-independent discourse behaviour whilst permitting domain-specific behaviour to be easily integrated. For example, asking for a check-in date to a hotel is domain-specific behaviour, whilst a confirmation strategy to verify newly supplied information would likely be shared across all domains.

Whilst the discourse model outlined in Section 2 provides a flexible and extensible basis from which to drive discourse behaviour across a number of different components, it is neither prescriptive nor suggestive with regard to the form of architectural design to use when building discourse components. As a baseline, it was assumed that a component architecture [11] would be used to modularise the different stages within the dialogue system. A component in this sense can encompass a simple process offering services to other components through to embodying complex processes which are responsible for driving key tasks within the dialogue. It was further assumed that information flow can be modelled as a transformational pipeline at the highest level between key components.

In order to develop a suitable architectural view *across* discourse components the often unpredictable and adaptive nature of dialogue was taken into consideration. In particular, the developed architecture would need to permit developers to easily customise existing functionality and also introduce new behaviour. It was decided to model discourse components using a partially managed multi-agent approach [12], whereby decisions are determined through agent interaction as supervised and directed by a manager. Figure 4 provides an overview of the developed architecture, where:

- The discourse product provides a single, shared, evolving record of dialogue progression, holding the current discourse state alongside previous states.
- A manager has responsibility for directing the evolution of some aspect of the dialogue task, e.g. natural language output, dialogue strategy selection, etc. One or more managers will be responsible for updating the product.
- An agent contains task expertise on offer to managers and used to determine how the dialogue product will evolve. Underlying support for inter-agent communication is performed in accordance with a subset of the defined FIPA agent communication language (ACL) [13].
- A forum provides a central point of communication between a manager and its associated agents. The forum provides a manager component with a means of directing task evolution enquiries towards one or more registered agents, as well as providing a means of enabling multiple agents to put forward an agreed response (i.e. supporting arbitration).
- A model is associated with a product and provides some form of product-related assessment (i.e. holding meta-information about the project). Models provide product assessments that can be of use to agents in terms of better informing their decision-making process.
- A registry holds common and reusable discourse elements (e.g. Objectives, Recipes, Date items, etc.). In effect, each library provides a set of reusable elements which can be used, as needed, by dialogue developers.

Structured as such, domain-independent processes are defined within managers, with 'plug-and-play' agents used to define and drive domain-specific behaviour in response to manager requests.

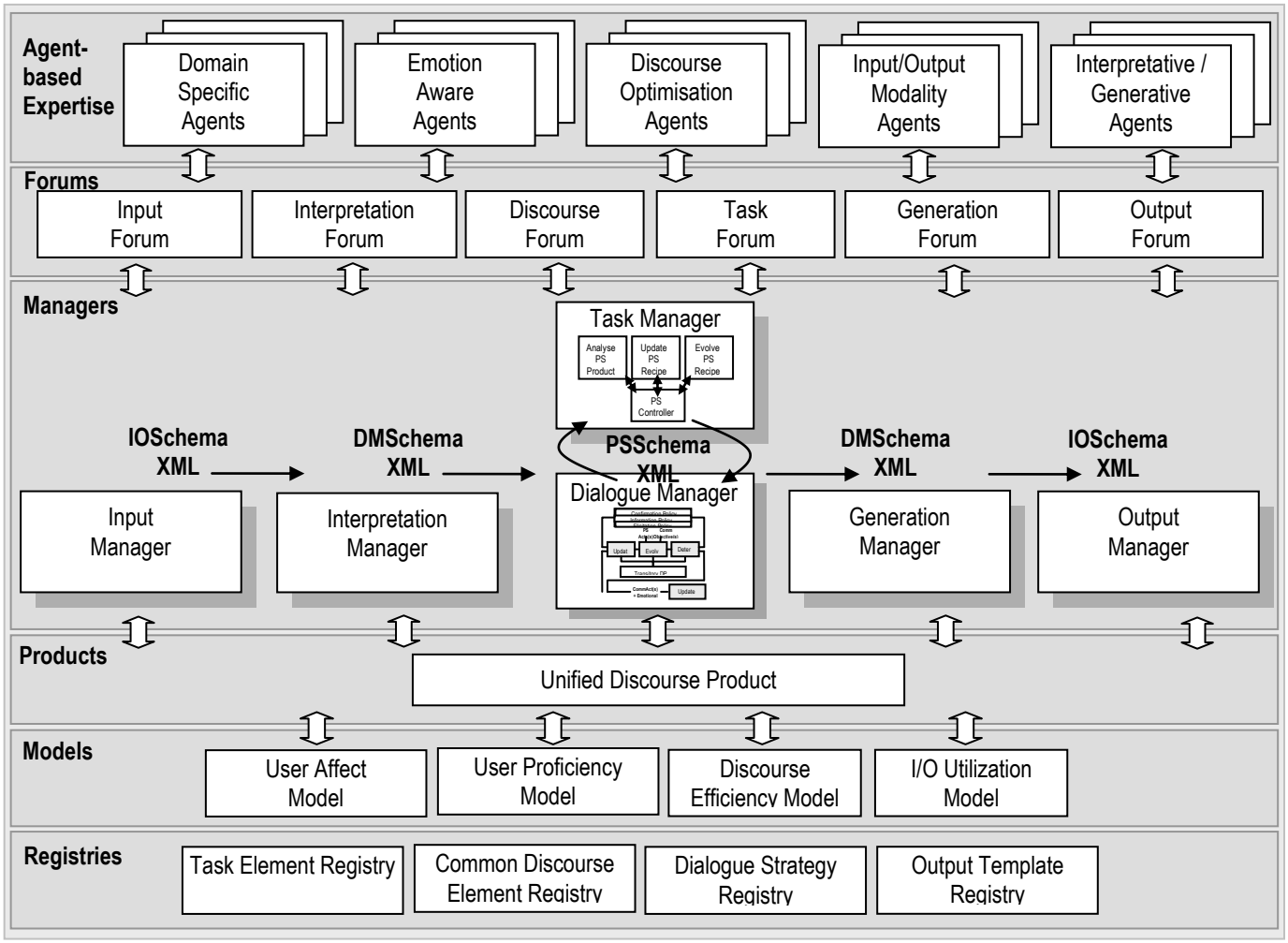

Figure 4. Overview of the developed multi-agent based dialogue system incorporating the unified discourse product

4.1.1 Use of Java in developing an extensible and configurable architecture

The developed architecture is intended to provide a framework upon which both domain-dependent and domain-independent natural language discourse behaviour can be readily incorporated and executed. Java interfaces were deployed as a means of defining the expected functionality of the manager, forum, agent, model, product and registry components.

Default implementations of each component were created. For example, the base agent class provides a default implementation of methods defined within the `Agent` interface, written to support the operation of the agent communication language (ACL). In turn, the base manager class implements the corresponding `Manager` interface to provide support for an input → integrate → evolve → output sequence. The use of inheritance means that domain-specific behaviour can be readily added on top of the inherited functionality offered through the base classes.

In order to support a distributed deployment of managers and agents, Java's object streams were used to enable inter-component communication between Java-based components. In addition, an extendable base agent was developed to support TCP-IP communication between non-Java or non-integrated systems using the defined XML schemas.

Combined together, the employed Java features enable new expertise to be readily incorporated within the dialogue system through the extension of one of the available base classes, with extending classes needing to have little awareness of the overall underlying communication structure employed within the system, though they must support the relevant defined XML schema.

5. OTHER JAVA ASPECTS

5.1 Assertion testing and exception handling

In developing the unified model, consideration was given to how other developers might extend the base classes through the introduction of domain-specific extensions. In particular, it was considered desirable that any usage assumptions associated with a class should, if invalidated, be communicated to the developer.

This was felt to be important given the overall complexity of the unified model and the need to ensure consistency of usage between different developers. For example, the element resolve process outlined in section 3.2.1.3 requires developers to adopt a common set of descriptive labels across discourse components. In order to provide the necessary safeguards, the base `Element` class embeds extensive generic error-handling alongside a range of sanity tests which ensure (to a certain degree) that usage assumptions concerning elements are not invalidated. Any invalided assumptions result in a generated report.

Error-checking and sanity tests were largely handled through the use of assertions and exceptions. Assertion-checking was intended to be of use to developers when building/modifying agents or introducing/changing discourse behaviour. The assertions can be disabled within a release build in order to reduce the cost of expensive run-time error/usage checks. Other forms of error were handled through the use of exceptions and a balanced combination of local and global exception handlers. An example of the error/sanity-checking can be seen below:

```
public void setValue( String id, Object value ) {
   try {
      // Check for parameter errors
      assert values != null;
      assert id != null &&
         values.containsKey(id);
      [[Other invocation error checks omitted]]

      // Sanity test for a user named identifier
      assert !isGeneratedElementName(id);
   }
   catch( java.lang.AssertionError error ) {
      if(isGeneratedElementName(id))
         Assembly.reportWarningMessage(
            "Element.setValue(): Adding element"
            + "with default (generated) name.");
      else
         Assembly.reportErrorMessage(
          "Element.setValue(): Invalid parameters"
          +  [[Error message omitted]]
      error.printStackTrace();
   }

   // Add value (+consider namespace addition)
   considerAddValueToNameSpace(value);
   values.put( id, value );
}
```

5.2 Performance monitoring and threading

Execution performance is of vital importance within a spoken dialogue system as the process of input recognition through to output synthesis should, on average, take less than approximately one second to complete. An average completion time of a couple of seconds will likely be noticeable to the user and lessen the acceptability and usability of the dialogue system.

Whilst some unmanaged languages, such as C++, provide developers with the forms of low-level control needed to optimise execution performance on a given target platform, the same cannot be easily argued of managed languages. However, whilst Java may not be suited to an application that must provide *optimal* performance, the rich language and library features within Java means that it offers the right language choice when developing complex architectures that will be subject to various forms of extension and refinement and which will need to run across a distributed platform, interacting with different external processes and data sources.

Two broad approaches were employed with a view to maximising the performance of the developed system:

- Exploiting the nature of turn-based dialogue: Execution load within a dialogue manager can be characterised by short periods of intensive activity (when constructing the reply to the user) interspersed by long periods of relative inactivity (waiting for the user's next input). This pattern was exploited within the design, whereby tasks such as cloning the last discourse turn in preparation for the next input, logging accumulated reports to disk, updating GUI elements, etc., are only triggered once the reply has been sent to the user.
- Implementation approach: Whilst code clarity and safety remained the most important implementation aspect, care was taken to ensure that garbage collection churn was minimised (e.g. autoboxing was avoided, discarded `Element` instances were cached and reused at key points, etc.). This limited the overall memory management cost.

To provide accurate timing, Java's `System.nanoTime` function was used to permit accurate ms time measurement for agent request times, IO costs, etc. The collected times were used to provide overall timing statistics for use within logging as well as offering a mechanism whereby maximum agent response times can be imposed and monitored.

5.3 Logging and Reporting

Given the wide range of different agents, managers, models, etc. involved within the dialogue system it is important that a developer can view component interaction. Equally, it is important that the dialogue models and strategies can be monitored to measure their suitability and effectiveness. In order to accomplish this, level-based logging (ranging from full-debug logging to summary logging) was used within classes to report inputs, processes and outputs. In order to permit more directed investigation, each component (agent, manager, etc.) implemented a reporting interface; whereby interested listeners (e.g. GUI components, etc.) could be attached to the discourse component and selectively receive reports from that component, e.g.:

```
if( Assembly.reportMessages )
   report( ReportType.Normal, 3, "PSManager ["
      + getIdentifier() + "] Objective release" );
```

Where `report` is defined as:

```
protected void report(
   ReportType type, int level, String message )
{
   if( Assembly.reportLevel < level ) return;

   if( reportingObject != null )
      reportingObject.reportMessage(message);

   switch( type ) {
      case Normal :
         Assembly.reportMessage(message);
         break;
      case Warning :
         Assembly.reportWarningMessage(message);
         break;
      case Error :
         Assembly.reportErrorMessage(message);
         break;
      default:
         Assembly.reportErrorMessage(
            "Unknown report type ["+type+"]");
   }
}
```

`Assembly.reportMessages` is defined outside of `report` to avoid needless report method calls when reporting is disabled.

6. CONCLUSIONS

This paper outlines the design of a unified discourse model which can be shared and accessed across different discourse components through the use of XML schemas. The paper also introduces a flexible and extensible agent-based architecture within which to structure and employ discourse expertise.

The development of the system was made possible through the use and exploitation of features available within Java. In particular, the paper illustrates how Java's object-oriented capabilities, interfaces, XML support, object reflection and distributed functionality can be employed to support the design and development of the outlined architecture. Additionally, the paper also indicates how Java language features such as assertions, exception-handling, generics and collections were used to ease the development process and produce robust software.

In conclusion, this paper provides an illustration of how the development of a complex software artefact is made tractable through the use of features and capabilities provided within the Java programming language.

7. ACKNOWLEDGMENTS

This work is supported by the EPRSC under project number EP/E028640/1 (ISIS).

8. REFERENCES

[1] McTear, M.F. 2004. Spoken Dialogue Technology: Towards the Conversational User Interface. Springer. 1852336722

[2] Glass, J., 1999. Challenges for spoken dialogue systems. *Proc. IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, Colorado, USA.

[3] Bohus, D., Raux, A., Harris, T., Eskenazi, M., Rudnicky, A., 2007b. Olympus: an open-source framework for conversational spoken language interface research, *Proc. Bridging the Gap: Academic and Industrial Research in Dialog Technology Workshop*, Rochester, NY.

[4] Rudnicky, A., Xu, W. 1999. An agenda-based dialog management architecture for spoken language systems, *Proc. Workshop on Automatic Speech Recognition and Understanding*, (Keystone, Colorado).

[5] Larsson, S. and Traum, D. R. 2000. Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural Language Engineering*, 6, 3-4 (Sep. 2000), 323-340.

[6] Bos, J. Klein, E. Lemon, O. and Oka, T. 2003. Dipper: Description and formalisation of an information-state update dialogue system architecture. *4th SIGdial Workshop on Discourse and Dialogue*, Sapporo.

[7] Turunen, M., Hakulinen, J., Räihä, K., Salonen, E., Kainulainen, A., and Prusi, P. 2005. An architecture and applications for speech-based accessibility systems. *IBM Systems Journal* 44, 3 (Aug. 2005), 485-504.

[8] http://www.voicexml.org/.

[9] Blaylock, N. Allen, J. 2005. A collaborative problem-solving model of dialogue. *Proc. 6th SIGdial Workshop on Discourse and Dialogue*, 200-211.

[10] Bunt, H. C., Girard,Y. M. 2005. Designing an Open, Multidimensional Dialogue Act Taxonomy. Claire Gardent and Bertrand Gaiffe (eds). *Proc. of the ninth workshop on the semantics and pragmatics of dialogue (SEMDIAL)*. 37-44.

[11] Hopkins, J. 2000. Component primer. *Communications of the ACM* 43, 10 (Oct. 2000), 27-30.

[12] Shoham, Y., Leyton-Brown, K. 2008. Multiagent Systems: Algorithmic, Game Theoretic, and Logical Foundations. Cambridge University Press.

[13] http://www.fipa.org/repository/aclspecs.html