Developing A Software Testing Ontology in UML

for A Software Growth Environment of Web-Based Applications

Hong Zhu
Department of Computing
Oxford Brookes University
Wheatley Campus, Oxford OX33 1HX, UK
Tel: ++44 1865 484580
Fax: ++44 1865 484545
Email: hzhu@brookes.ac.uk

Qingning Huo
Lanware, Ltd.
68 South Lambeth Road, London SW8 1RL, UK
Tel: T: ++44 20 7735 1717
Email: Qingning.Huo@lanware.co.uk
Developing A Software Testing Ontology in UML

for A Software Growth Environment of Web-Based Applications

Abstract

This chapter introduces the concept of software growth environments to support sustainable long term evolution of web-based application systems. A multi-agent prototype system is designed and implemented with emphasis on software testing. In this environment, software tools are agents that cooperate effectively with each other and human testers through communications at a high level of abstraction. New tools can be integrated into the system with maximal flexibility. These are achieved through the design and utilisation of an ontology of software testing that represents the knowledge of software engineering and codifies the knowledge for computer processing as the contents of an agent communication language. The ontology is represented in UML at a high level of abstraction so that it can be validated by human experts. It is also codified in XML for computer processing to achieve the required flexibility and extendibility.

Keywords:

INTRODUCTION

The Internet and Web are becoming a distributed, heterogeneous and hypermedia computation platform, which stimulates many new progresses in software applications, cf. (Crowder, Wills & Hall, 1998). However, web-based applications are complex and difficult to develop and maintain. In (Zhu, et al. 2000), we argued that most web-based applications are by nature evolutionary and proposed a growth model of software process. To support the sustainable evolutionary development of web-based systems, we designed a multi-agent architecture of software development and maintenance environment and developed a prototype system for testing web-based applications. A key feature of the architecture and the prototype system is the use of an ontology of software testing to facilitate the communications between agents and between agents and human developers and testers. In this paper, we report the development of the ontology of software testing and its representation in UML.

The remainder of the chapter is organised as follows. Section 2 gives the motivation of our research and briefly outlines our approach to the development and maintenance of web-based applications. The structure and features of the multi-agent software environment is described. A prototype system for testing web-based applications is presented. Section 3 reports the ontology of software testing and its representation in UML. Section 4 discusses the uses of the ontology in the prototype systems. Section 5 concludes the chapter with a discussion of related works and directions for future research.

BACKGROUND AND MOTIVATIONS

Characteristics of Web-Based Applications

According to Lehman (2001), software systems can be classified into three types according to what ‘correctness’ means to the system. An S-type program is required to satisfy a pre-stated specification. For such a system, correctness is the absolute relationship between the specification and the program. A P-type program is required to form an acceptable solution to a stated problem in the real world. The correctness of a P-type program is determined by the acceptability of the solution to the stated problem. An E-type program is required to solve a problem or implement an application in a real-world domain which often has no clearly stated specification. Correctness here is determined by the program’s behaviour under operational conditions and judged by the users. Obviously, many kinds of
web applications such as e-commerce, enterprise portal, web-based CRM systems, e-government, e-science, etc., belong to the E-type, where problems are not clearly stated and the correctness of the system is judged by the users for its fitness to their purposes.

Different types of software systems tend to demonstrate different evolutionary behaviours, because their development processes are dominated by different types of uncertainties. Generally speaking, there are three types of uncertainties associated with software development (Lehman, 1990). Gödel-like uncertainties arise because software systems and their specifications are models of the real world. The representations of such models and their relationships are Gödel incomplete. Consequently, the properties of a program cannot be completely known from the representations. Heisenberg-type uncertainties result from the processes of using the system that may change the user’s perception and understanding of the application. A common phenomenon in the development of software systems is that the users are uncertain about the requirements, but they are often certain that ‘I’ll know it when I see it’ (Boehm, 2000). Uncertainties of this type exhibit themselves in the form of changing requirements either in the form of unsatisfactory implemented or to be implemented functional or non-functional requirements, or the emergent of new requirements. Pragmatic uncertainties are due to the problems in actually performing the development activities. Software development is still a process that relies on human performance. During this process, errors are made and faults are introduced. Many types of risks in software development are caused by this type of uncertainty. For example, the adaptation of a new development method, the use of a new software tool or programming language, the use of a new library of software code and so on may introduce uncertainties to the quality of the product and the development process.

Although these sources of uncertainties are associated with all software development projects, Gödel and Heisenberg types of uncertainties have strong impact on E-type software in general and web-based applications in particular. However, pragmatic uncertainty also plays a significant role in the development of web-based applications as web technology has been changing rapidly in the past few years. Consequently, web-based applications commonly demonstrate a clear evolutionary life-cycle. During the evolution process, uncertainties are clarified through developing and adjusting the model of the problem, revising the representation of the models, updating users’ requirements and correcting errors of development activities. In the meantime, new uncertainties may emerge and require further development and maintenance. Lehman characterised E-type systems’ evolution processes by 8 laws of evolution (Lehman, 2001), which are summarised in Table 1 below. These laws
should be equally applicable to web-based applications. In addition, in the investigation of web-based applications, we also observed a common phenomenon of web-based systems, that is, web-based systems commonly contain components developed using different technology, such as component codes written in different languages and executed on different platforms, data represented in different formats, interfaces designed to comply with different standards, interactions proceeded in different protocols, etc. We call this phenomenon the law of diversity, which is also listed in Table 1 together with Lehman’s laws.

Table 1. Laws of Evolution of E-type Systems

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuing Change</td>
<td>E-type systems must be continually adapted else they become progressively less satisfactory in use.</td>
</tr>
<tr>
<td>Increasing Complexity</td>
<td>As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it.</td>
</tr>
<tr>
<td>Self Regulation</td>
<td>Global E-type system evolution processes are self-regulating.</td>
</tr>
<tr>
<td>Conservation of Organisational Stability</td>
<td>Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving E-type system tends to remain constant over product lifetime.</td>
</tr>
<tr>
<td>Conservation of Familiarity</td>
<td>In general, the incremental growth and long term growth rate of E-type systems tend to decline.</td>
</tr>
<tr>
<td>Continuing Growth</td>
<td>The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime.</td>
</tr>
<tr>
<td>Declining Quality</td>
<td>The quality of E-type systems will appear to be declining unless they are rigorously adapted, as required, to take into account changes in the operational environment.</td>
</tr>
<tr>
<td>Feedback System</td>
<td>E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.</td>
</tr>
<tr>
<td>Diversity</td>
<td>An E-type system contains components that are developed using a diversity of techniques and integrated into the system at different times.</td>
</tr>
</tbody>
</table>

Lehman’s laws were proposed on the bases of his observations on E-type software systems that had survived after a long evolutionary process. They can be considered as ‘survival guidelines’ for the evolutionary development of E-type software systems. Violating these laws in the development of an E-type software system may mean a death penalty to the system. Here, the death of a software system should be understood in Peter Naur’s sense (1992) that the state of death become visible when demands for modifications of the program cannot be intelligently answered although the program may continue to be used for execution and provide useful results.
From Lehman’s theory of software evolution, we can see that clarifying uncertainties is the driving force of E-type software evolution. Therefore, the development of an E-type software system is best to be a process of growth in functionality. Tool supports must be provided to manage the complexity and quality of the product during its whole life time. Figure 1 below depicts a growth model of software lifecycles of web-based applications. As argued in (Zhu et al., 2000), this process is suitable for the development of web-based applications. It also has a number of advantages, which include reducing time pressure on the developers, minimizing development risks, offering learning opportunities to developers, improving communications between developers and users as well as various other stakeholders, etc.

Figure 1. The Growth Model of Software Lifecycle
To support sustainable long term evolutionary development of web-based applications with a growth strategy, we proposed a new type of software environments and designed an architectural structure for their implementations. Figure 2 depicts the architecture of software environments, which consists of a number of cooperative agents.

Figure 2. Architecture of Software Growth Environment

The architecture of software growth environment consists of the following types of agents. Development service agents provide developers with various supports to the evolution of software systems in the growth strategy. They fulfil the functions that support evolutionary development of web-based applications. Management agents are agents that manages agents and responsible for the following tasks.

(a) Registration. When a new agent is added into the system, information about its functionality, capability, execution environment, etc. are registered with a management agent. When an agent is deleted from the system, its registration information is updated.

(b) Task allocation. A management agent receives service requests as well as development and maintenance task requests. When such a task is requested, it searches for an appropriate agent and assigns the task to the agent through a task allocation protocol.

(c) Monitoring and recording agents’ and the system’s behaviours. The management agents will monitor the progresses of each task and record the state and outcomes of each task. They will also monitor and record the behaviour of each service agent for the optimisation of future task allocation.

These agents may also interact with the application system and its components to obtain data and knowledge of the application and their evolution histories in order to support their future evolutions. The interactions between human developers and the agents may also be
through a set of *interface agents* that provide assistant to each individual developer to communicate with the development tools and to access the data and knowledge of the application system at a high level of abstraction. Ideally, the application system consists of a number of *application service agents* that provide the services and functionality of the application system to its users.

This architecture significantly differs from existing software development environments such as CASE tools and run-time support environments such as middleware due to the following two features. Because of these features, it is called *software growth environment*.

First, tools that support the development and maintenance of a system run in the same environment of the software system. They coexist with the system monitoring the evolution process of the system and supporting the modifications of the system. Moreover, they grow with the system as new tools are integrated into the environment when new functional components of the application are developed using new technology and integrated into the system. The relationship between the tools and the system is similar to the relationship between a tree and its natural environment where it is growing, and between a human and his/her social environment that changes as the person is growing up.

Second, the tools (i.e. agents) in the environment collect, store and process the information about the system and the knowledge of software development, and present such knowledge to human beings or other software tools at a high level of abstraction when requested. Such information and knowledge include: (a) the structure of the system, the functionality, versions, evolution history and configurations of the system components, etc.; (b) the capability, performance, and operational conditions of each development and application service agent, as well as interrelationships between them; (c) the knowledge about software development processes, logical and temporal relations between development tasks and how tasks are decomposed into subtasks, etc.

Obviously, the key to the success of such a software growth environment is the mechanisms that enable software tools flexibly integrated into the system gradually and enables tools to cooperation with each other effectively. This can only be achieved by using agent technology and a well-developed ontology and representing the ontology in a highly flexible and extendable format to enable the collaboration between the agents.

*A Prototype System for Testing Web-Based Applications*

To demonstrate the feasibility and advantages of the above proposed approach, we
developed a prototype with emphasis on quality assurance and testing.

As shown in Figure 3, the environment consists of a number of agents to fulfill various testing tasks for web-based applications. These agents can be distributed to different computers, for example, as in Figure 4, on an application server, a test server, and a client. In fact, agents can be freely distributed according to any specific configuration. They can also be mobile and change their location at runtime. The following briefly describes the agents that have been implemented for testing web-based applications. More details can be found in (Huo, Zhu & Greenwood, 2003).

Figure 3. Structure of the Prototype System for Testing Web-based Applications

**GWP (Get Web Page)** agents retrieve web pages from a web site. Two agents of this type have been implemented. One is GWP-No-Cache, whose function is to fetch the web page of a given URL, and return the page’s contents. Another is GWP-Cache, which has the same functionality as GWP-No-Cache, but with cache ability. It uses a knowledge base to store downloaded web pages, and uses the last modification time to determine whether the web page is updated on the cached copy.

**WPI (Web Page Information)** agents analyze the source code of a web page and extract various useful information from the source code. The information includes the page title, meta-information, hyperlinks, etc. They also store the information about the web page’s structure in a knowledge base. When a web page’s structural information is requested, a message is sent to a GWP agent with a HTML source file as the content of the message. It runs a HTML parser on the file and extracts information of the structure of the file from the parser. If the input page is unmodified since last retrieval, the WPI agent just uses the cached data in the knowledge base.

**WSS (Web Site Structure)** agents analyze the hyperlink structure of a web site, and
generate a directed graph to describe the structure. This structure is also stored in a knowledge base to share with other agents.

TCG (Test Case Generator) agents generate test cases to test a web site according to certain testing criteria. Currently, three agents are implemented for node coverage, link coverage and linear independent path coverage criteria, respectively. Details of these test criteria for hypertext applications can be found in (Jin, Zhu & Hall, 1997).

TCE (Test Case Executors) agents execute test cases, and generate execution results. Two TCE agents are implemented. One is to run the test cases interactively in front of the human tester with the aid of a testing assistant agent. The other is to playback a recorded test sequence. This is often used in regression testing.

TO (Test Oracles) agents verify whether a test result matches a specification. Different types of test results require different kinds of oracles. For each type of result data, one agent is design and implemented. Some simply compare the test output with the results from previously recorded tests. Some examine if the test output satisfies a certain condition, such as if the structure matches a certain pattern. These patterns can be predefined or generated automatically from previous tests or defined by software engineers.

TA (Testing Assistants) agents are user interface agents that assist human testers in the process of testing. They communicate at a high level of abstraction and in a language that are understandable by human testers based on the ontology. They provide helps to human testers on various testing tasks. For example, they get test requirements from the human users, send correctly formatted messages to TCG to generate test cases, present the generated test cases to the user, guide the user to walk through the links in a web site to test each web page on the test cases, collect human tester’s feedback on the validity of tested pages, record testing history and generate testing reports.

WSM (Web Site Monitor) agents monitor the changes on web sites and generate new testing tasks according to these changes.

An ontology of software testing is developed and codified in XML for the communications between agents. The following section gives details of the ontology and its uses in the prototype system.

**ONTOLOGY OF SOFTWARE TESTING**

Generally speaking, ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define
extensions to the vocabulary (Uschold & Gruninger, 1996). It is widely recognised that ontology can be used where domain knowledge specification is useful (Staab & Maedche, 2001). For example, ontology can be used in the communications between people and information systems. It can also be used to improve inter-operability between systems, such as translation of modelling methods, paradigms, languages and software tools. It can also be used in systems engineering, e.g. to achieve reusability, shareability, search, reliability, specification and knowledge acquisition (Neches et al., 1991; Uschold & Gruninger, 1996; Staab & Maedche, 2001). Ontology can be used in a multi-agent system as a means for agents to share knowledge, to transfer information and to negotiate their actions. For example, Fox and Gruninger (1994) proposed using ontology to represent agent activities in a cooperative information system. The advantage of using ontology in such a system is that ontology provides a standard specification of concepts in the specific domain. All agents that understand the ontology can participate in the system. Although ontology has been an active research area in the past decade, there is no ontology reported in the literature for software engineering purpose. In this section, we report our work on designing an ontology of software testing (Huo, Zhu & Greenwood, 2002).

A number of ontology modelling methods have been proposed in the literature. The most widely used traditional approaches include the Knowledge Interchange Format (KIF) (National Committee for Information Technology Standards), description logic, and object oriented modelling, such as in UML (Cranefield, Haustein & Purvis, 2001). In recent years, XML is more and more used as the format to represent ontology and as a format of agent communication languages. XML has a very simple syntax. It is customisable, extensible, and most importantly, suitable for web-based applications. The users can define the tags and formats to represent both simple concepts and complex structures. These tags and formats form a formal knowledge representation language. For these reasons, XML is used in our system to codify the ontology for computer processing. However, an XML representation of ontology is at a rather low level of abstraction. It does not support the validation of the ontology by domain experts. Therefore, we need a representation of ontology at a higher level of abstraction. As a powerful modelling language, UML has the advantage of representing the concepts and relationships at a high level of abstraction that are readable and understandable to human beings so that the knowledge represented in the ontology can be validated by domain experts. Therefore, in addition to the representation of the ontology in XML at machine processing level, we also represent the structure and relationships of the concepts and relations of the ontology in UML. In this chapter, we focus on the UML.
representation. The XML Schema (XSD) definition of the XML representation is given in the appendix.

**Taxonomy of Testing Concepts**

Taxonomy is a way to specify and organize domain concepts. We divide the concepts related to software testing into two groups: the basic concepts and compound concepts. As shown in Figure 4, there are six types of basic concepts related to software testing, which include *testers, context, activities, methods, artefacts,* and *environment.*

![Figure 4. Basic Concepts of Software Testing](image)

For each basic concept, there may be a number of sub-concepts. For example, a testing activity can be the generation of test cases, the verification of test results, the measurement of test adequacy, etc. A basic concept may also be characterized by a number of properties, which are the parameters of the concept. For example, a software artefact is determined by (a) its format, such as HTML file, JavaScript, etc., (b) its type, such as a program, or a test suite, etc., (c) its creation and revision history, such as who and when created the artefact, and who and when revised it, and the version number of the artefact, etc. (d) the location that the artefact is stored, and (e) the data, i.e. the contents, of the artefact. The following briefly discusses each type of the basic concepts.

**(A) Tester.** A tester refers to a particular party who carries out a testing activity. A tester can be a *human being,* a *software tool* (including software agents), or a *team,* which consists of one or more testers. This structure represented in UML as follows in Figure 5.

![Figure 5. The Concept of Testers](image)
A tester team contains a number of other testers, which can be individuals or sub-teams, and has a leader, which is an attribute that gives the name of the leader of the team. An important attribute of tester is capability that describes what a tester can do. The concept of capability is a compound concept that must be defined on the bases of other basic concepts of software testing. It is discussed in the next subsection.

**Example 1.** The following is an example of a human tester named Howard represented in XML.

```xml
<TESTER TESTER_TYPE="HUMAN" TESTER_NAME="Howard" />
```

The following is an example of a test team that consists of Joe as the leader and a software agent as a member.

```xml
<TESTER TESTER_TYPE="TEAM" TESTER_NAME="ATEAM" TESTER_LEADER="JOE">
  <TESTER TESTER_TYPE="HUMAN" TESTER_NAME="JOE" />
  <TESTER TESTER_TYPE="SOFTWARE" TESTER_NAME="ANAGENT" />
</TESTER>
```

(B) Context. Software testing activities occur in various software development stages and have different testing purposes. For example, unit testing is to test the correctness of software units at implementation stage. Integration testing is to verify the interface between software units at integration stage. The context of testing in the development process determines the appropriate testing methods as well as the input and output of the testing activity. Typical testing contexts include *unit testing*, *integration testing*, *system testing*, *regression testing*, and so on.

![Figure 6. The Concept of Test Context](image)

(C) Activity. There are various kinds of testing activities, including *test planning*, *test case generation*, *test execution*, *test result validation and verification*, *test coverage measurement*, *test report generation*, and so on.
(D) Method. For each testing activity, there may be a number of testing methods applicable. For instance, applicable unit testing methods include structural testing, fault-based testing and error-based testing. Each test method can also be classified into program-based and specification-based. There are two main groups of program-based structural testing methods: control-flow methods and data-flow methods. The control-flow methods include statement coverage, branch coverage and various path coverage criteria, etc.; see (Zhu, Hall & May, 1997) for a survey of research on software testing methods. These concrete testing methods are instances of various subclasses of testing methods. The structure of the concept of testing methods is shown in UML as follows.

(E) Artefact. Each testing activity may involve a number of software artefacts as the object under test, intermediate data, testing result, test plans, test suites, test scripts, and so on. There are different types of objects under test, such as source code in programming languages, HTML files, XML files, embedded images, sound, video, Java applets, JavaScript, documents, etc. Testing results include error reports, test coverage measurements, etc. Each artefact is also associated with a location that the artefact is stored, the data, i.e. the contents, of the artefact, and a history of creation and revision, which include the creator, update-time,
version numbers, etc.

**Figure 9. The Concept of Artefact**

**F) Environment.** The environment in which testing is performed is also an important issue in software testing. Information about the environment includes hardware and software configurations. For each hardware device, there are three essential fields, including the device category, the manufacturer and the model. For the software component, there are also three essential fields: the type, product name and version.

**Figure 10. The Concept of Test Environment**

**Compound Concepts**

Compound concepts are those defined on the bases of basic concepts, for example,
testing tasks and agent's capability. They are defined as follows.

**(A) Capability.** The capability of a tester is determined by the activities that a tester can perform together with the context for the agent to perform the activity, the testing method used, the environment to perform the testing, the required resources (i.e. the input) and the output that the tester can generate.

![Figure 11. The Compound Concept of Capability](image)

**Example 2.** The following is an example of capability description in XML. The agent is capable of doing node coverage test case generation in the context of system testing of hypertext applications represented in HTML.

```xml
<CAPABILITY>
    <CONTEXT CONTEXT_TYPE="SYSTEM_TEST" />
    <ACTIVITY ACTIVITY_TYPE="TEST_CASE_GENERATION" />
    <METHOD METHOD_NAME="NODE_COVERAGE_TESTING" />
    <CAPABILITY_DATA CAPABILITY_DATA_TYPE="INPUT">
        <ARTEFACT ARTEFACT_TYPE="OBJECT_UNDER_TEST" ARTEFACT_FORMAT="HTML" />
    </CAPABILITY_DATA>
    <CAPABILITY_DATA CAPABILITY_DATA_TYPE="OUTPUT">
        <ARTEFACT ARTEFACT_TYPE="TEST_SUITE" ARTEFACT_FORMAT="NODE_SEQUENCES" />
    </CAPABILITY_DATA>
</CAPABILITY>
```

**(B) Task.** A testing task consists of a testing activity and related information about how the activity is required to be performed, such as the context to give the purpose of the testing activity, the specific testing method to use, the environment in which the activity must be
carried out, the available resources and the requirements on the test results. It can be represented by the following UML class diagram.

![UML class diagram](image)

Figure 12. The Compound Concept of Task

Notice that, the class diagram for the concept of task is very similar to the diagram for the concept of capability. However, the semantics of the concepts are different.

**Example 3.** The following is an example of testing task that requires to generate test cases according to the node coverage criterion for the HTML pages at the URL http://www.brookes.ac.uk.

```xml
<TASK>
  <CONTEXT CONTEXT_TYPE="SYSTEM_TEST" />
  <ACTIVITY ACTIVITY_TYPE="TEST_CASE_GENERATION" />
  <METHOD METHOD_NAME="NODE_COVERAGE_TESTING" />
  <TASK_DATA TASK_DATA_TYPE="INPUT">
    <ARTEFACT ARTEFACT_TYPE="OBJECT_UNDER_TEST" ARTEFACT_FORMAT="HTML">
      <ARTEFACT_LOCATION>http://www.brookes.ac.uk</ARTEFACT_LOCATION>
    </ARTEFACT>
  </TASK_DATA>
</TASK>
```

Notice that, not all combinations of basic concepts make sense. For example, the node coverage method cannot be applied to a media file, such as images, sound or videos. A weakness of XML is that it provides very limited power to restrict such illegal combinations.

**Basic Relations**

Relationships between concepts play a significant role in the management of testing
activities in our multi-agent system. We identified a number of relationships between basic
concepts as well as compound concepts.

Basic relations between basic concepts form a very important part of the knowledge of
software testing. They must be stored in a knowledge-base as basic facts. This type of
knowledge is listed below.

(A) Subsumption relation between testing methods. A testing method $A$ subsumes
method $B$, if the application of method $A$ always achieves a test adequacy that is adequate
according to method $B$. The subsumption relation has been intensively investigated in
software testing literature; see (Zhu, Hall & May, 1997) for a survey.

(B) Compatibility relation between artefacts’ formats. An artefact format $A$ is
compatible with artefact format $B$, if they are of the same type and the format of $A$ is
compatible with $B$ in the sense that if a tester understands the format of $A$ implies that the
tester also understands the format of $B$.

(C) Enhancement relation between environments. An environment $A$ is an
enhancement of environment $B$, if a testing task can be performed in environment $B$ implies
that it can also be performed in environment $A$. Assume that an enhancement relation is
defined on software and hardware components. The enhancement relation between
environments can be defined formally as follows. Let environments $A$ and $B$ consist of sets
$\{a_1, a_2, ..., a_n\}$ and $\{b_1, b_2, ..., b_m\}$ of hardware and software components, respectively. $A$ is an
enhancement of $B$, if and only if for all $b_i$, $i=1, 2, ..., m$, there is one component $a_j \in \{a_1, a_2, ..., a_n\}$ such that $a_j$ is an enhancement of $b_i$.

(D) Inclusion relation between test activities. A test activity $A$ may include a number
of more basic activities. For example, the test execution activity may include the derivation of
test driver and/or test stubs, the installation of test tools, etc. A test activity can be completed
only if all the sub-activities are completed.

(E) Temporal ordering between test activities. To fulfill a test task, a number of test
activities must be carried out in certain temporal order. For example, the generation of test
cases must be carried out before test execution.

These relations are all partial orderings. That is, they are transitive and reflexive. Figure
13 summarises these basic relations.
Based on these basic facts and knowledge, more complicated relations can be defined and used through knowledge inferences. The following are definitions of the most important ones.

**The MorePowerful Relation on Capability**

The relation $\text{MorePowerful}$ is defined between two capabilities. Let $C$ represent the set of all capabilities. For all $c_1, c_2 \in C$, we say that $c_1$ is more powerful than $c_2$, write $\text{MorePowerful}(c_1, c_2)$, if and only if all of the following statements are true.

1. $c_1$ and $c_2$ have the same context, and
2. $c_1$ and $c_2$ have the same activity, and
3. The method of $c_1$ subsumes the method of $c_2$, and
4. The environment of $c_2$ is an enhancement of the environment of $c_1$, and
5. For all input artefacts $a_1$ of $c_1$, there is at least one input artefact $a_2$ of $c_2$ that $a_1$ is compatible with $a_2$, and
6. For all output artefacts $a_2$ of $c_2$, there is at least one output artefact $a_1$ of $c_1$ that $a_2$ is compatible with $a_1$.

Informally, $\text{MorePowerful}(c_1, c_2)$ means that a tester has capability $c_1$ implies that the tester can do all the tasks that can be done by a tester who has capability $c_2$. In UML, the $\text{MorePowerful}$ relation is an association class; see Figure 14 for its structure.

It is easy to prove that the $\text{MorePowerful}$ relation is also a partial ordering.

**Theorem 1** (Reflexiveness): $\forall c \in C. \text{MorePowerful}(c, c)$.

**Theorem 2** (Transitiveness):

$\forall c_1, c_2, c_3 \in C. \text{MorePowerful}(c_1, c_2) \land \text{MorePowerful}(c_2, c_3) \Rightarrow \text{MorePowerful}(c_1, c_3)$. 
The Contains Relation on Test Tasks

The relation Contain is defined between two tasks. Let \( T \) represent the set of all tasks. For all \( t_1 \) and \( t_2 \in T \), we say that task \( t_1 \) contains \( t_2 \), write \( \text{Contain}(t_1, t_2) \), if and only if all of the following conditions are true.

1. Task \( t_1 \) and \( t_2 \) have the same context;
2. The activity of \( t_1 \) includes the activity of \( t_2 \);
3. The method of \( t_1 \) subsumes the method of \( t_2 \);
4. The environment of \( t_1 \) is an enhancement of the environment of \( t_2 \);
5. For all input artefacts \( a_2 \) of \( t_2 \), there is at least one input artefact \( a_1 \) that \( a_2 \) is compatible with \( a_1 \);
6. For all output artefact \( a_2 \) of \( t_2 \), there is at least one output artefact \( a_1 \) of \( t_2 \) that \( a_2 \) is compatible \( a_1 \).

Informally, \( \text{Contain}(t_1, t_2) \) means that accomplishing task \( t_1 \) implies accomplishing task \( t_2 \).

Similar to the relation MorePowerful on capabilities, the Contains relation is also an association class and can be similarly represented in UML; see Figure 14.

The Contain relation is also a partial ordering. That is, we have the following property of the relation.

**Theorem 3** (Reflexiveness): \( \forall t \in T, \text{Contain}(t, t) \).

**Theorem 4** (Transitiveness): \( \forall t_1, t_2, t_3 \in T, \text{Contain}(t_1, t_2) \land \text{Contain}(t_2, t_3) \Rightarrow \text{Contain}(t_1, t_3) \)

The Matches Relation Between Tasks And Capabilities

In the assignment of a testing task to a tester, a broker agent must answer the question whether the task matches the capability of the tester. For example, assume that an agent is registered as capable of generating statement coverage test cases for Java Applets and a test task is requested for structural testing a Java Applet. The broker agent need to infer that the agent is capable of fulfil the task. Therefore, it is necessary to define the following Matches relation between a capability and a task.

For any \( c \in C \) and \( t \in T \), we say that capability \( c \) matches task \( t \), write \( \text{Match}(c, t) \), if and only if all of the following conditions are true.

1. Capability \( c \) and task \( t \) have the same context;
2. The activity of \( c \) includes the activity of \( t \);
3. The method of \( c \) subsumes the method of \( t \), or the method of \( t \) is an instance or a subclass of the method of \( c \);
4. The environment of \( t \) is an enhancement of environment of \( c \);
5. For all artefacts \( a_c \) in the input artefact set of \( C \), there exists at least one artefact \( a_t \) in the input artefact of \( t \), such that \( a_t \) is compatible with \( a_c \);
6. For all artefact \( a_t \) in the output artefact set of \( t \), there exists at least one artefact \( a_c \) in the output artefact of \( c \), such that \( a_c \) is compatible with \( a_t \).
Match(c, t) means that a tester with capability c can fulfil the task t. The following properties of the relations form the foundation of the inferences that the broker agent requires in the assignment of testing tasks.

**Theorem 5:** \( \forall c_1, c_2 \in C, \forall t \in T, \text{MorePowerful}(c_1, c_2) \land \text{Match}(c_2, t) \Rightarrow \text{Match}(c_1, t) \).  

**Theorem 6:** \( \forall c \in C, \forall t_1, t_2 \in T, \text{Contain}(t_1, t_2) \land \text{Match}(c, t_1) \Rightarrow \text{Match}(c, t_2) \).

Figure 14 below shows the structures of compound relations.

![Compound Relations Diagram](image)

**USES OF THE SOFTWARE TESTING ONTOLOGY**

In this section, we discuss how the ontology is used in our multi-agent system.

**Communication Protocols And Task Scheduling**

In a multi-agent system, many agents can play a similar role but with different specialities. As discussed in the previous section, in our system, agents that play the same role may have different capabilities, are implemented with different algorithms, execute on different platforms, and are specialised in dealing with different formats of information. The agent society is dynamically changing; new agents can be added into the system and old agents can be replaced by a newer version. This makes task scheduling and assignment more important and more difficult as well. Therefore, management agents are implemented as brokers to negotiate with testing service agents to assign and schedule testing activities to testing service agents. Each broker manages a registry of agents and keeps a record of their capabilities and performances. Each service agent registers its capability to a broker when joining the system. Tests tasks are also submitted to the brokers. For each task, the broker will send it to the most suitable agent use the Match relation as a means of inferences. The following describe the communication protocols and mechanisms for capability registration and testing task submission.
Combining Ontology with Speech-Act

In a multi-agent society, a clearly defined semantics is necessary for agents to express their intentions and commitments to tasks. For example, when an agent sends a message to a broker, it must make the intension of the message clear as to register their capabilities or to submit a test job request, or to report a test result, etc. Such intensions can be represented as illocutionary forces of the message. As in (Singh, 1993; 1998), illocutionary forces can be classified into 7 types: assertive, directive, commissive, permissive, prohibitive, declarative, and expressive.

To incorporate illocutionary forces in our agent communications, we associate each message with a speech-act parameter. Hence, messages have the following structure in UML.

![Message Structure Diagram]

Figure 15. Message Structure

The following example is a typical scenario of using the ontology in agent communication.

**Example 4.**

The following is a sequence of messages between agents $A_1$ and $A_2$ and a broker $B$.

1. Agent $A_1$ sends an ASSERTIVE message with a <$capability$> parameter to the broker $B$.
   This means the agent $A_1$ wants to register to the broker $B$ and claims its capability.

2. Agent $A_2$ sends an EXPRESSIVE message to the broker $B$, with a <$task$> parameter describing a testing task. This means that the agent wants to find some agent to perform the testing task.

3. The broker $B$ searches its knowledge about registered agents, and finds that agent $A_1$ is the best match for the task. It then sends a DIRECTIVE message with the <$task$> parameter to agent $A_1$.

4. When agent $A_1$ finishes the task, it sends an ASSERTIVE message with an <$answer$> parameter to the broker. The <$answer$> parameter describes the status of the task and
output of the task if it is successful, or the reason of failure or error messages if it is not successful.

(5) The broker $B$ may forward the message to agent $A_2$, or try to find another agent to carry out the testing task in case the output of agent $A_1$ is not successful. □

**CONCLUSION**

This paper presents an ontology of software testing and discusses it uses in a multi-agent software environment to support the evolutionary development and maintenance of web-based applications. The prototype system consists of a number of software testing and quality assurance agents. Each testing task is assigned to an agent in the system, which either performs the requested task or decomposes it into smaller tasks for other agents to perform. In this way, agents cooperate with each other to carry out complicated testing tasks.

As Jennings and Wooldridge (1998) pointed out, agent techniques benefit in application areas that involve diverse platforms and information formats and in dynamic environments. Our experiment supports this claim. In particular, for the following reasons, agent technology is suitable for testing web-based systems.

The dynamic and evolutionary nature of web-based applications requires constantly monitoring the changes of the system and its environment. Sometimes, the changes in the system and its environment may require changes in testing strategy and method accordingly. Agents are adaptive, and they can adjust their behaviours based on environment changes. These changes can be integrated to the system dynamically.

Web-based information systems often operate on a diversity of platforms and use various different formats of media and data. This demands a wide variety of test methods and tools to be used in testing a single system. Multi-agent systems can provide a promising solution to this problem. Different agents are built to handle different types of information, to implement different testing methods and to integrate different tools. Thus, each individual agent can be relatively simple while the whole system is powerful and extendible.

The distribution of large volume of information and functionality over a large geographic area requires testing tasks carried out at different geographic locations and to transfer information between computer systems. The agents can be distributed among many computers to reduce the communication traffic.

Agents also provide a nice way that human testers interact with testing tools. The relationship between human testers and agents are no longer a commander/slave relation.
Instead, the human tester and the agents form a testing team and cooperate with each other. In particular, when a part of a complicated testing task cannot be performed by the tools, the testing will not fail completely. Instead, the agent can pass the unsolvable task to the human tester and ask for help. This feature is of particular importance for testing web-based applications, because they are often extended by integrating into the system new components developed with new technology that may have no ready made testing tools at the beginning. The collaborative relationship not only release human testers from routine work, which are performed by the agents, but also enables the human testers to participate in the testing process by taking the most intellectually challenging tasks so that the whole testing job can be performed more efficiently and effectively.

In the design and implementation of the prototype system, we realised that the key issue is the mechanism that enables the flexible integration of agents into the environment and the effective communications between agents and between the human testers and agents. Our solution is the ontology of software testing. It is used as the content language for software agents to register into the system with a capability description, for human testers and agents to make testing requests and report testing results, for management agents to allocate tasks to agents. This paper reports how the concepts of the ontology and the relations between them are defined in UML. Their properties are also analysed. Speech-act theory is incorporated in the system and combined with the ontology to define communication protocols and to facilitate collaborations between agents. Our experience in the development of the ontology further confirmed the advantages and benefits of using ontology in tool integration that have already been observed in other application domains such as those mentioned in section 0, but have not been explored in software engineering research as far as we know.

The ontology is designed based on the domain knowledge of software testing to mediate the communications between the agents. It was represented in XML to codify the knowledge of software testing for agents’ processing of messages. The representation in XML for run-time communications between agents achieved a flexibility of modification and extendibility very well. However, during the testing and validation of the prototype system, we realised that XML representation is at a rather low level of abstraction. It is not very readable for domain experts to validate the ontology. Our first attempt to represent the ontology at a higher level of abstraction was the uses of BNF to describe the syntax structure of XML expressions (Huo, Zhu and Greenwood, 2003). For example, the following is the BNF definition of tester.

\[
<\text{tester}> ::= \text{"<TESTER" } \{\text{<tester parameter>}\} \text{"">" } \{\text{<tester>}\} \text{"</TESTER>"}
\]
DEVELOPING A SOFTWARE TESTING ONTOLOGY

<tester parameter> ::= <tester type> | <name> | <capability> | <leader>

<name> ::= "NAME =" <string>

<leader> ::= "LEADER =" <name>

<tester type> ::= "TYPE =" ( "HUMAN" | "SOFTWARE" | "TEAM" )

BNF descriptions of the XML syntax are significantly shorter than the corresponding XML Schema definitions. For example, the XML Schema definition of XML representation of the concept tester below is 3 times longer than the BNF expressions. BNF is more suitable to human readers. Moreover, software engineers and computer scientists, who are the domain experts of software testing, are more familiar with BNF than XML Schema.

```xml
<xs:element name="TESTER">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="TESTER"/>
    </xs:sequence>
    <xs:attribute name="TESTER_TYPE" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="HUMAN"/>
          <xs:enumeration value="SOFTWARE"/>
          <xs:enumeration value="TEAM"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="TESTER_NAME" use="required"/>
    <xs:attribute name="TESTER_LEADER"/>
  </xs:complexType>
</xs:element>
```

However, the BNF representation is still not good enough for end users of the prototype system, who communicate with the agents in the ontology of software testing to request testing tasks and receive testing results. BNF is still at the syntax level. It does not properly represent some important concepts of ontology, such as the concept of sub-class, etc. Therefore, we also developed the representation of the ontology in UML. It is at a suitable level of abstraction for both validation by human experts and communication with the end users.

Representing the ontology in two notations at different abstraction levels raised the question how to validate the consistency between the UML and XML representations. Recently, standard XML representations of UML models and tools though XMI have emerged to enable the automatic translation of UML models into XML representations. Using such techniques will result in completely re-writing the whole prototype system. It is unclear and worth further investigation that whether the automatic technique of translation
from UML to XML can be applied to our ontology. It seems that our ontology is significantly more complicated than the examples and case studied conducted in the development of such techniques and reported in the literature.

We are also further investigating the methodology of developing ontology at a wider context of software engineering and further developing the prototype of software growth environment. The automatic translation technique will be beneficial to our further research. A difficulty problem is the development of a model of the whole system, including definitions of the organizational structure, functionality and dynamic behaviours of the agents. It seems that an agent-oriented modelling language such as the CAMLE (Shan and Zhu, 2003a; 2003b) or AUML (FIPA Modelling TC) is necessary to catch the agents’ autonomous and social behaviours. In our design and implementation of the ontology in UML and XML, we noticed that UML does not provide adequate support to the formal specification and analysis of the relations between concepts although OCL can be partially helpful. For example, we have to use first order logic formula for the definitions and proofs of the properties of compound relations.
REFERENCES


APPENDIX. XML SCHEMA (XSD) DEFINITION OF XML REPRESENTATION OF THE ONTOLOGY OF SOFTWARE TESTING

The following is the complete XML Schema (XSD) definition of the XML representation of the ontology of software testing.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified">
  <!-- TESTER -->
  <xs:element name="TESTER">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="TESTER"/>
      </xs:sequence>
      <xs:attribute name="TESTER_TYPE" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:token">
            <xs:enumeration value="HUMAN"/>
            <xs:enumeration value="SOFTWARE"/>
            <xs:enumeration value="TEAM"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="TESTER_NAME" use="required"/>
      <xs:attribute name="TESTER_LEADER"/>
    </xs:complexType>
  </xs:element>
  <!-- CONTEXT -->
  <xs:element name="CONTEXT">
    <xs:complexType>
      <xs:attribute name="CONTEXT_TYPE" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:token">
            <xs:enumeration value="UNIT_TEST"/>
            <xs:enumeration value="INTEGRATION_TEST"/>
            <xs:enumeration value="SYSTEM_TEST"/>
            <xs:enumeration value="REGRESSION_TEST"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <!-- ACTIVITY -->
  <xs:element name="ACTIVITY">
    <xs:complexType>
      <xs:attribute name="ACTIVITY_TYPE" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:token">
            <xs:enumeration value="TEST_PLANNING"/>
            <xs:enumeration value="TEST_CASE_GENERATION"/>
            <xs:enumeration value="TEST_CASE_EXECUTION"/>
            <xs:enumeration value="TEST_RESULT_VERIFICATION"/>
            <xs:enumeration value="TEST_COVERAGE_MEASUREMENT"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:enumeration value="TEST_REPORT_GENERATION"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:element name="METHOD">
<xs:complexType>
<xs:attribute name="METHOD_NAME" use="required">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="CONTROL_FLOW_TESTING"/>
<xs:enumeration value="DATA_FLOW_TESTING"/>
<xs:enumeration value="STATEMENT_COVERAGE_TESTING"/>
<xs:enumeration value="BRANCH_COVERAGE_TESTING"/>
<xs:enumeration value="PATH_COVERAGE_TESTING"/>
<xs:enumeration value="NODE_COVERAGE_TESTING"/>
<xs:enumeration value="LINK_COVERAGE_TESTING"/>
<xs:enumeration value="LIP_COVERAGE_TESTING"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="METHOD_TECHNIQUE">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="STRUCTURAL_TESTING"/>
<xs:enumeration value="FAULT_BASED_TESTING"/>
<xs:enumeration value="ERROR_BASED_TESTING"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="METHOD_APPROACH">
<xs:simpleType>
<xs:restriction base="xs:token">
<xs:enumeration value="PROGRAM_BASED_TESTING"/>
<xs:enumeration value="SPECIFICATION_BASED_APPROACH"/>
<xs:enumeration value="RANDOM_TESTING"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<!-- ARTEFACT -->
<xs:complexType name="ARTEFACT">
<xs:sequence>
<xs:element ref="ARTEFACT"/>
</xs:sequence>
</xs:complexType>
<xs:element name="ARTEFACT">
<xs:complexType>
<xs:sequence>
<xs:element minOccurs="0" ref="TESTER"/>
<xs:element minOccurs="0" ref="ARTEFACT_DATA"/>
<xs:element minOccurs="0" ref="ARTEFACT_LOCATION"/>
</xs:sequence>
<xs:attribute name="ARTEFACT_TYPE" use="required">
...
<xs:simpleType>
  <xs:restriction base="xs:token">
    <xs:enumeration value="OBJECT_UNDER_TEST"/>
    <xs:enumeration value="TEST_PLAN"/>
    <xs:enumeration value="TEST_SCRIPT"/>
    <xs:enumeration value="TEST_RESULT"/>
    <xs:enumeration value="TEST_SUITE"/>
    <xs:enumeration value="TEST_COVERAGE"/>
    <xs:enumeration value="ERROR_REPORT"/>
    <xs:enumeration value="SPECIFICATION"/>
  </xs:restriction>
</xs:simpleType>

<xs:attribute name="ARTEFACT_FORMAT" use="required">
  <xs:simpleType>
    <xs:restriction base="xs:token">
      <xs:enumeration value="PROGRAM"/>
      <xs:enumeration value="HTML"/>
      <xs:enumeration value="XML"/>
      <xs:enumeration value="TEXT_FILE"/>
      <xs:enumeration value="WORD_FILE"/>
      <xs:enumeration value="PDF_FILE"/>
      <xs:enumeration value="POSTSCRIPT_FILE"/>
      <xs:enumeration value="BMP_IMAGE"/>
      <xs:enumeration value="JPEG_IMAGE"/>
      <xs:enumeration value="CD_SOUND"/>
      <xs:enumeration value="MPEG_VIDEO"/>
      <xs:enumeration value="VCD_VIDEO"/>
      <xs:enumeration value="DVD_VIDEO"/>
      <xs:enumeration value="JAVA_APPLET"/>
      <xs:enumeration value="JAVA_SCRIPT"/>
      <xs:enumeration value="NODE_SEQUENCES"/>
      <xs:enumeration value="LINK_SEQUENCES"/>
      <xs:enumeration value="LIP_SEQUENCES"/>
      <xs:enumeration value="DATA"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:element name="ARTEFACT_DATA" type="any"/>
<xs:element name="ARTEFACT_LOCATION" type="any"/>

<xs:element name="ENVIRONMENT">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="HARDWARE"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="SOFTWARE"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="HARDWARE">
  <xs:complexType>
    <xs:attribute name="HARDWARE_DEVICE" use="required"/>
    <xs:attribute name="HARDWARE_MANUFATURER" use="required"/>
    <xs:attribute name="HARDWARE_MODEL" use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="SOFTWARE">
  <xs:complexType>
    <xs:attribute name="SOFTWARE_TYPE" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="OS"/>
          <xs:enumeration value="DATABASE"/>
          <xs:enumeration value="COMPILER"/>
          <xs:enumeration value="WEB_SERVER"/>
          <xs:enumeration value="WEB_BROWSER"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="SOFTWARE_PRODUCT" use="required"/>
    <xs:attribute name="SOFTWARE_VERSION" use="required"/>
  </xs:complexType>
</xs:element>

<!-- CAPABILITY -->
<xs:element name="CAPABILITY">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1" ref="CONTEXT"/>
      <xs:element minOccurs="1" maxOccurs="1" ref="ACTIVITY"/>
      <xs:element minOccurs="1" maxOccurs="1" ref="METHOD"/>
      <xs:element minOccurs="0" maxOccurs="1" ref="ENVIRONMENT"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="CAPABILITY_DATA"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- TASK -->
<xs:element name="TASK">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1" ref="CONTEXT"/>
      <xs:element minOccurs="1" maxOccurs="1" ref="ACTIVITY"/>
      <xs:element minOccurs="1" maxOccurs="1" ref="METHOD"/>
      <xs:element minOccurs="0" maxOccurs="1" ref="ENVIRONMENT"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="TASK_DATA"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType>
  <xs:element name="TASK_DATA">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="ARTEFACT">
          <xs:attribute name="TASK_DATA_TYPE" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:token">
                <xs:enumeration value="INPUT"/>
                <xs:enumeration value="OUTPUT"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
</xs:complexType>

<!-- REPLY -->
<xs:element name="REPLY">
  <xs:complexType>
    <xs:attribute name="REPLY_STATUS" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="SUCCESSFUL"/>
          <xs:enumeration value="FAILED"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="REPLY_REASON"/>
  </xs:complexType>
</xs:element>

<!-- MESSAGE -->
<xs:element name="MESSAGE">
  <xs:complexType>
    <xs:choice>
      <xs:element ref="CAPABILITY"/>
      <xs:element ref="TASK"/>
      <xs:element ref="REPLY"/>
    </xs:choice>
    <xs:attribute name="MESSAGE_ACT" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="ASSERTIVE"/>
          <xs:enumeration value="DIRECTIVE"/>
          <xs:enumeration value="COMMISIVE"/>
          <xs:enumeration value="PERMISSIVE"/>
          <xs:enumeration value="PROHIBITIVE"/>
          <xs:enumeration value="DECLARATIVE"/>
          <xs:enumeration value="EXPRESSIONIVE"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
</xs:complexType>
<xs:complexType>
  <xs:sequence>
    <xs:any min_occurs="0" max_occurs="unbounded" processContents="strict"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>