Towards a Software Component Ontology

Alex Talevski
Digital Ecosystems and Business Intelligence Institute (DEBII), Curtin University of Technology
GPO Box U1987, Perth, WA, 6845, Australia
A.Talevski@cbs.curtin.edu.au

Pornpit Wongthongtham
Digital Ecosystems and Business Intelligence Institute (DEBII), Curtin University of Technology
GPO Box U1987, Perth, WA, 6845, Australia
P.Wongthongtham@cbs.curtin.edu.au

Surasak Komchaliaw
Digital Ecosystems and Business Intelligence Institute (DEBII), Curtin University of Technology
GPO Box U1987, Perth, WA, 6845, Australia
Maxtor_p@hotmail.com

ABSTRACT
Research has shown that component-based software engineering leads to software that exhibits higher quality, shorter time-to-market and therefore, lower development cost. However, the development of component-based systems has been widely plagued with problems surrounding the integration of third-party components. Currently, software developers are forced to rely on ambiguous definitions of a component's services. There is no easy to understand protocol for defining how third-party components and component compositions are described and integrated into systems. Most vendors specify their components' services in a proprietary or context dependant fashion. This makes it difficult to clearly understand a component's services, their use and their operational pre and post conditions. Software Engineering ontologies define common sharable software engineering knowledge. They explicitly define software engineering concepts, their relationships and their interactions. In this paper, we propose a Software Component Ontology that specifically defines a formal, explicit specification of a shared conceptualization in the domain of software component engineering. We propose the use of our software component ontology as the basis for the development of future component compositions and component based applications.

Categories and Subject Descriptors
D.2 [**Software Engineering**]: Software Component.

General Terms
Management, Standardization.

Keywords
Software Component, Ontology, Software Engineering.

1. INTRODUCTION
Large-scale systems account for the majority of all software development undertakings [1]. Growing enterprises demand rapid and frequent software development, maintenance, customization and evolution. However, the software development industry tends to develop large-scale systems that are monolithic, error-prone and expensive [2][3][4]. Such systems normally evolve from an uncoordinated build-and-fix pattern. A lack of a systematic approach to large-scale software development has resulted in many project failures [2][3][4]. Failures such as these can be attributed to many factors relating to software development complexity.

1.1 Software Complexity
Parnas [5] suggests that when a system is described by a continuous function, it can contain no hidden surprises. Small input changes should always cause correspondingly small changes in outputs [6].

Software programmers, analysts, architects and testers can only simultaneously comprehend seven chunks of information, plus or minus two [7]. Therefore, the distinguishing characteristic of large-scale software compared with the smaller variants is that it is much more difficult to grasp. A large-scale software system may have many variables that reside on multiple threads of control. The collection of variables and various processes represents the state of a software application. Discrete systems may have a vast number of possible combinations that potentially place a system in a new state. If an error is made as a result of improper development reasoning, the state of the system may change unpredictably.

1.2 Component based Software
Software complexity has plagued large-scale system development projects. In particular, we have noticed that rapidly changing requirements, diverse end-user bases, and the creation of extended enterprises (collaborative development models) play a major part in the necessity for simplified software development. Components extend the existing object principles by strengthening the role of an interface. A component interface separates the component implementation from its interaction. It contains a collection of operations and attributes that specify the services that a component provides. Component-based software engineering is a way of raising the level of abstraction for software development so that software development can be simplified through a more

ISBN: 978-1-60558- 349-5, iiWAS 2008. http://doi.acm.org/10.1145/1456492.1456507

coordinated divide and conquers approach to software problem solving. This results in benefits such as:

- Reusability – Reusability removes dependencies on development tools and languages, and allows the reuse of already tested and certified components that are of high quality and perform well.
- Productivity - When reusable artifacts are applied throughout the software process, less time is spent creating the plans, models, documents, code, and data that are required to deliver a system.
- Quality - Ideally, a software component that is developed for reuse would be verified to be correct, and would contain no defects. As a component is reused, defects are found and eliminated and its implementation is refined.
- Cost - A reduction in cost results from less effort being spent developing the software product. In addition, an increase in software quality lowers software maintenance costs.

Component-based approaches achieve loose coupling among interacting software components where disparate components interact using a common interaction protocol and architectural constraints. By abstracting a component's internals through an interface, components become well isolated and standardized. Component services are well-defined, self-contained, and do not depend on the context or state of other services [8]. Therefore, composite component architectures can be formed without knowing the specific implementation details of a particular service. This allows specialized component developers to focus on their areas of expertise while ignoring other complexities and continually refining component quality and performance. These components are then guaranteed through their interface definition and acquired as building blocks.

1.3 Component Tailorability

Tailorability is a formal concept which defines how generic software can satisfy the specialized, rapidly changing, unclear and / or evolving changes in third-party system requirements. It provides a means for the dynamic creation and modification of software based on multiple levels of detail and complexity. Morch [9] identified three levels of tailoring activity that typically exist within a software development project.

- Customization – Customization refers to user interface modifications.
- Integration – The integration level of tailoring refers to changes in application components, compositions, interconnections and their configuration.
- Extension – The extension level of tailoring refers to the addition of new application components and configurations.

Each tailoring activity requires formal, explicit specification and a shared conceptualisation for it to be eased within a distributed development environment where third-party components are reused. Using a framework as a basis for the creation and modification of software, it possible to construct, customize, integrate and evolve software in a straightforward way.

With the evolution of the web and web services it is now more and more common for software developers to acquire third party components online and reuse them within their own application contexts [10] in a distributed development environment. Unfortunately, even with the broad benefits of component development, component-based systems have been widely plagued with problems surrounding system composition and integration. Currently, software developers are forced to rely on lacking or ambiguous definitions and specifications of a component's services. There is no easy to understand protocol for defining how a third-party component is described, configured, integrated and modified to fit within third-party system requirements or within distributed development environments. Most vendors specify their components' services in a proprietary or context dependant fashion. This makes it difficult to clearly understand a component's operational properties and requirements.

2. ONTOLOGIES

Changes are inevitable during software development projects; such projects are continuously confronted with an evolving specification problem. If such changes are not properly tracked and traced or maintained, this would impede the development and third party integration of components. In component projects, the project data needs to be modified periodically to reflect

- project development progress
- changes in the software requirements
- changes in design
- additional functionality
- incremental improvements
- reconfiguration

Ontologies are a widely accepted state-of-the-art knowledge representation. Software engineering concepts, ideas and knowledge, software development methodologies, tools and techniques are organized into a software engineering ontology that is used as the basis for classifying the communication concepts by enabling specification, reasoning, problem solving and other intelligence aspects within software development projects.

We have merged Gruber's [11], Borst's [12], and Studer's [13] definitions of an ontology as a basis for the software engineering ontology definition. Ontologies are formal, explicit specifications of a shared conceptualization in the domain of software engineering with the following properties;

- machine-process-able semantics
- explicitly defined
- consensual knowledge
- abstract model

When coupled with multi-agent systems, ontologies allow greater ease of communication by aggregating the agreed project knowledge with domain knowledge, and other concepts of software engineering into a shared information resource platform that is distributed amongst the development team and others that reuse the projects artifacts. The first Software Engineering Ontology is available online at www.seontology.org.

In this paper we present the model of our software component ontology using the notations proposed in [14] to represent the ontology and communication architecture. The ontology will be transformed to a software development resource using the web ontology language, OWL, and can be accessed by multi-site, multi-team and multi-development groups. The development of

the software component ontology basically consists of two processes i.e. (*i*) creating concepts or ontology classes and the relationships that hold among them; (*ii*) defining constraints of the relationships or ontology restrictions.

3. SOFTWARE COMPONENT ONTOLOGY

A software component ontology is well defined, both human and machine understandable, common, standardized and sharable software engineering knowledge within the component ware domain. It is concerned with all processes of component production from the stages of component requirements through verification and validation. It defines component engineering concepts, abstractions, relationships and interactions as domain concepts and instantiations for manual or automated reasoning surrounding component compositions and interaction. Software component ontologies signify standardized project information which evolves to reflect component development. It fosters a seamless and virtual intra project environment of project data across sites and third party vendors / buyers. In order to define components and allow them to be discovered, identified, queried and reused, the ontology defines the use of components within a software development environment. Using the software component ontology, it is possible to define components through uniquely identifying and querying their, description, interfaces, operations, attributes, pre and post conditions, performance characteristics and extra functional properties. The ontology enables effective ways of distributing such knowledge for software engineers, software developers and automated components systems. Reaching such a consensus of understanding is of benefit in a distributed and/or third-party component development environment.

In this section, we focus on component specifications and compositions. Using the Software Component Ontology, components, their interfaces and their interconnections are defined as an ontology specification. This explicit model supports software tailoring by defining application constructs, their composition and interaction. External entities (other applications and constructs) may access this ontology data in order to evaluate the composition of a tailored system or use the services that it provides. A composition is typically formed from many interconnected components that are constructed in a layered and hierarchical manner. Interconnected compositions are coupled using operation and attribute connectors and adaptors.

3.1 Component

The Software Component Ontology model illustrated in Figure 1 represents an explicit specification of a single software component and its interface.

A component may expose a number of interfaces. Each interface either provides or requires services in the form of attributes and operations. An attribute is a named property value that defines the characteristics and state of a component. An operation is the implementation of a specific service that represents the dynamic behaviors of a component. Operations are specified using their input and output parameters and pre and post conditions.

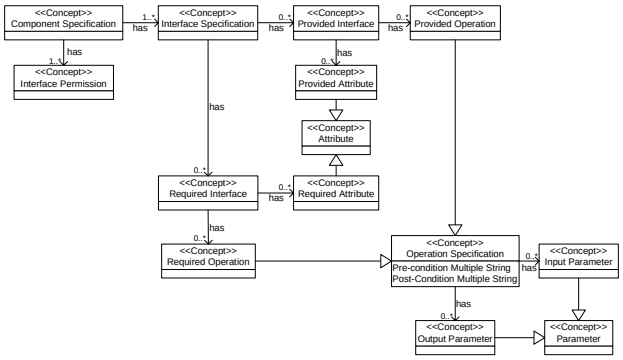

Figure 1. Software component ontology model

3.2 Customization

Graphical User Interfaces (GUI) attempt to reduce the complexity involved in using the computer by utilizing the great pattern recognition and graphic processing power of the human mind. A typical user interface contains a number of windows in the shape of a form. Forms can contain a number of controls which in turn can embed controls within them. User interface and control behavior is typically specialized through the modification of a control's attributes, operations and event mappings.

The Software Component Ontology User Interface model illustrated in Figure 2 describes a component's user interface and its configuration, composition and interaction with underlying components.

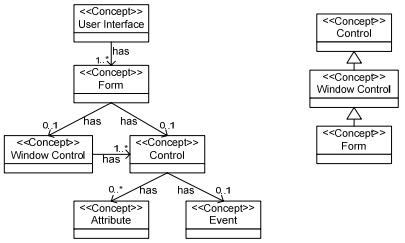

Figure 2. Software component ontology user interface model

3.3 Integration

Connectors and adaptors provide a level of indirection that reduces dependencies among components. Interconnected compositions are coupled using operation and attribute connectors and adaptors.

Attribute Connectors

Figure 3 illustrates a Software Component Ontology Attribute Connector. Attribute connectors are used to connect required and provided attributes. Once connected, a required attribute always requests the value it requires from the provided attribute that it is connected to. Two attribute connector types exist. Attribute connectors can be used to connect required component attributes to provided attributes or operation returns, or to a static value provided by the user. Attribute adaptors are connectors that translate component interaction.

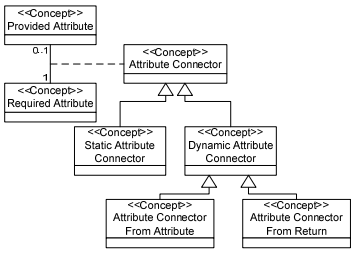

Figure 3. Software component ontology attribute connector model

Operation Connectors

Figure 4 illustrates a Software Component Ontology Operation Connector model. Like attribute connectors, operation connectors are used to connect required and provided operations. Once connected, a required operation always calls the provided operation that it is connected to. Two operation connector types exist. Operation connectors can be used to connect required component operations to provided operations or attributes, or to a static return value as provided by the user. Attribute connectors may be used when required to connect operation parameters. Operation adaptors may be used to connect an incompatible provided operation.

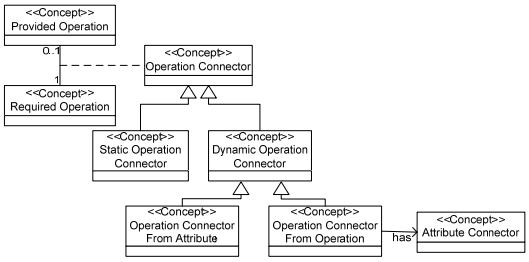

Figure 4. Software component ontology operation connector model

3.4 Extension

A component implemented by combining the functionality provided by others is referred to as a composite component. The process of extending applications through component composites is referred to as component composition. Component composition is an iterative process where we define new and increasingly complex component compositions in a recursive and hierarchical fashion. Therefore, component-based applications represent groups of interconnected component plug-ins in a composite architecture. Whenever a component passes a message to another, the two components are said to be synchronized. The interaction between client and server is predefined as an interaction protocol. The ultimate goal of this approach is the use of a plug and play strategy where components are used as building blocks and the interconnection of such components is performed through the use of models and automated tools. Our approach extends applications from four key constructs; Application Solutions (Figure 5), Service Modules, Compositions, and Components (Figure 6).

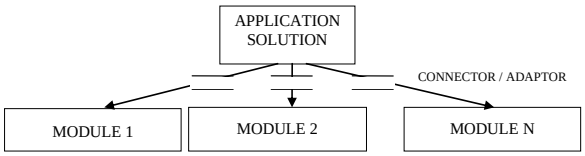

Figure 5. Application solution

Components' compositions and interconnections must be specified using the software component ontology in order to assist solution developers and automated systems with specification and reasoning surrounding the construction of component based applications. The whole set of software component concepts are transformed into the generic software component ontology as domain knowledge in the area of software engineering. The software component ontology is divided into generic sub-ontology and the specific sub-ontology. The generic software component ontology represents all software component concepts while specific software component ontology represents some concepts of software components for the particular project need.

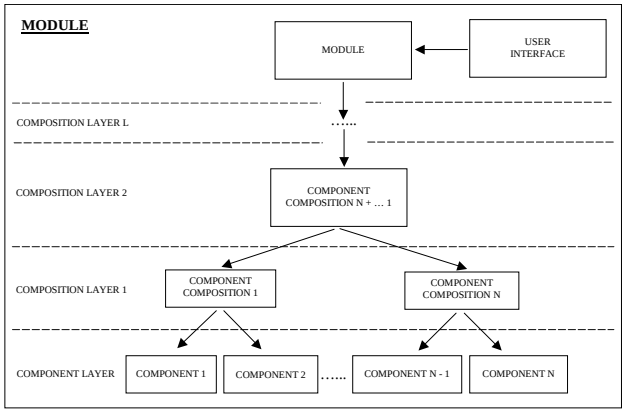

Figure 6. Service module

4. SOFTWARE COMPONENT ONTOLOGY MANAGEMENT

The ontology system is built on top of Jena [15]. Developed by Hewlett-Packard, Jena 2.1 is a framework with the capacity of manipulating ontologies [16]. The ontology system provides navigating, querying, and manipulating functions. The design philosophy of the ontology system is to use the in-memory storage model and serialize it into a physical document stored in the ontology repository. The ontology system provides three services: navigating, querying and manipulating services. To navigate the software component ontology, the ontology system reads the OWL software component ontology into a model and then accesses the individual elements. The software component ontology can be navigated for clarification or classification of certain concepts. Software component ontology queries serve as a searching tool to help narrow down the vast number of concepts and instances in the ontology. The software component ontology could be queried on interface definitions, component composition, component specification, etc. By consulting the software component ontology, automatic selection and composition of software components based on interface definition can be achievable by a software agent.

Specific component ontology instances can be added, deleted and updated. There will be changes over a period of time made to the instantiations of the ontology. The changes will be recorded by a logger object. Basically, instantiations can be updated by three basic operations: add, delete and modify. The add operation extends the existing instantiations of the ontology with new instantiations. The delete operation removes some instantiations from the ontology. The modify operation modifies some instantiations of the ontology but it still keeps its original construct. Generally, any update to the instantiations of ontology can be described by a sequence of the three operations. For example, a delete operation followed by an add operation can be considered as a replacement operation. Notice that the replacement operation loses its original construct while the modify operation still maintains its construct.

5. CONCLUSION AND FUTURE WORK

Component-based software engineering leads to software that exhibits higher quality, shorter time-to-market and therefore, lower development cost. However, the development of component-based systems has been widely plagued with problems surrounding the integration and composition of third-party components. Currently, software developers are forced to rely on ambiguous definitions of a component's services. There is no easy to understand protocol for defining how third-party components and interconnected component compositions are described and integrated into systems that are developed in multi-site development environments. The lack of a framework for expressing component collaboration makes component-oriented programs more complicated to maintain, expand and widely reuse. The Software Component Ontology outlined in this paper defines common sharable software component knowledge. Software component concepts, their relationships and their interactions are explicitly defined. Our model provides an approach to transforming explicit semantic component knowledge to conceptual component knowledge representations.

6. REFERENCES

[1] "ATP FOCUSED PROGRAM: Component-Based Software", On-line at: http://www.atp.nist.gov/atp/focus/cbs.htm (2003).

[2] "Salvaging a Failed CRM Initiative", Gartner Inc., On-line at: http://www3.gartner.com/DisplayDocument?ref=g_search&id=352804 (2002).

[3] M. Doane, "The Overwhelming Failure of Go-It-Alone CRM", Meta Group, On-line at: http://www.metagroup.com/us/displayArticle.do?oid=35932 (2002).

[4] I. Sommerville, G. Dewsbury, K. Clarke, M. Rouncefield, "Dependability and Trust in Organisational and Domestic Computer Systems", In Trust in Technology: A Socio-technical Perspective, Kluwer 2004.

[5] D. Parnas, The Influence of Software Structure on Reliability, in Current Trends in Programming Methodology: Software Specification and Design. Englewood Cliffs, NJ: Prentice-Hall. 1977.

[6] D. Parnas, "Software Aspects of Strategic Defence Systems", Communications of the ACM, 28(12): 1326--1335, 1985.

[7] G. Miller, "The Magical Number Seven, Plus or minus Two: Some Limits on Our Capacity for Processing Information", The Psychological Review 63(2): 81--97, March 1956.

[8] "Service-oriented architecture (SOA)", On-line at: http://www.service-architecture.com/webservices/articles/service-oriented_architecture_soa_definition.html (2003).

[9] A. Morch, "Three Levels of End-User Tailoring: Customisation, Integration, and Extension", Computers and Design in Context, The MIT Press, pp: 51--76, Cambridge, Massachusetts, 1997.

[10] C. Jackson and H. J. Wang, "Subspace: Secure Cross-Domain Communication for Web Mashups," presented at WWW 2007, Banff, Alberta, Canada, 2007.

[11] T.R. Gruber, A translation approach to portable ontology specification, in Knowledge Acquisition, 1993.

[12] W. Borst, Construction of Engineering Ontologies, Centre of Telematica and Information Technology, University of Tweenty, Enschede, The Netherlands, 1997.

[13] R. Studer, B. VR, D. Fensel, Knowledge Engineering: Principles and Methods, in: IEEE Transactions on Data and Knowledge Engineering, 1998.

[14] P. Wongthongtham, E. Chang, T.S. Dillon, "Ontology Modelling Notations for Software Engineering Knowledge Representation", 2007 Inaugural IEEE International Conference on Digital Ecosystems and Technologies, Cairns, Australia, February 2007.

[15] Carroll, J.J., et al., Jena: Implementing the Semantic Web Recommendations, Digital Media Systems Laboratory, HP Laboratories Bristol, 2004.

[16] McBride, B. Jena: Implementing the RDF Model and Syntax Specification. in Semantic Web Workshop, WWW2001. 2001.