Software Component Specification: A Study in Perspective of Component Selection and Reuse

CJ Michael Geisterfer, Sudipto Ghosh
Computer Science Department
Colorado State University
Email: {mgeister,ghosh}@cs.colostate.edu

Abstract

In component-based software engineering research, much effort has gone into developing specification techniques for software components. There exist many software component specification techniques, from Interface Description Language (IDL), to design-by-contract based, to formal methods. However, much of the focus of the research literature is aimed at component specification for the development of components, not their use. The current best practices for component specification ignore information that is vital in determining if an available, ready to use component contains precisely the functional and extra-functional properties required and if that component can be used in the target environment. These specification techniques do not sufficiently support selection and reuse of software components.

This paper evaluates some of the current component specification techniques with respect to the needs of component selection and reuse. From this evaluation, some recommendations made as to advancing the development of component specifications to include the purposes of component selection and reuse.

Keywords: *component specification, component selection, component reuse, component based development*

1. Introduction

In component-based software engineering (CBSE) research, much effort has gone into developing specification techniques for software components. In the majority of the research literature, the solutions and approaches have concentrated on the aspects of the *development* of components. That is, specification has concentrated on communicating the implementation specifics of components, so that component implementation work can be divided within an organization. However, one of the primary benefits of components is the ability to decrease time-to-market for new systems and applications by leveraging existing known solutions by *reusing* available, ready-to-use components [5, 7, 12, 23]. In order to achieve this decrease in time-to-market, application designers and developers must be able to select the appropriate software components from some repository of available components based on a definition of needs, and then assemble them into their component systems.

To have a reasonably high expectation that selected components meet the actual needs, engineers must be able to specify their needs precisely and components must contain enough information at the correct level of precision. For reuse, the implication is that the specifications of the available, ready-to-use components contain enough information to facilitate the successful assembly of the component into the target environment, including configuration and dependency information. However, this is hardly ever the case.

Although there exist many methods of software component specification, from Interface Description Language (IDL), to design-by-contract based, to formal methods, much of the focus of the research literature is aimed at the development of components. The current best practices for component specification ignore information that is vital in determining if an available, ready to use component contains precisely the functionality required and if that component can be used in the target environment. These specification approaches do not sufficiently support selection and reuse of software components.

Significant progress in the advancement of the component-based paradigm will not occur without successfully addressing selection and reuse. The issues related to *replacement* of software components (stated as primary driver for CBSE in [5]) are also closely related to the issues involved in selection and reuse. In order to successfully replace a given component, it should be specified (understood) to same extent that one would need to specify the requirements for the component that is to replace it.

The goals of this paper are: (1) as a prelude to further

Proceedings of the Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS 2006)
0-7695-2515-6/06 $20.00 © 2006 IEEE

studies, to gain an appreciation for the software component specification space; (2) to evaluate some of the current research literature on component specification with respect to component selection and reuse; (3) as a result of that evaluation, determine the set of information about a component that is required to determine selection and reuse; and (4) to make some initial recommendations on the needs for specification of software components in light of the needs of selection and reuse.

This paper is organized as follows. Section 2 provides an overview of the type of information that is needed to enable automated component selection and to facilitate reuse. Section 3 summarizes and evaluates several specification approaches found in the literature. Recommendations based on that evaluation are offered in Section 4. Section 5 concludes our paper.

2. Specification Needs of Component Selection and Reuse

To perform an evaluation of existing methods and frameworks, we must first have an idea as to the aspects of the specification of software components that are important to the selection and reuse of these components. Historically, specification of components initially concentrated on syntactic description of interfaces (for example, the early COM and CORBA models [11, 25, 32]). The next step was adding some semantic information, namely as pre- and post-conditions on the operations defined in the interfaces. To date, the current "best practices[1]" have extended the semantic information to include a little more operational descriptions (see Section 3.2 below).

The problem with these methods of specification is that they only consider the functionality that is *provided* by the component. Such information is usually included in a *Provided Interface*, which is essentially the *signature* of the component, providing the syntactic information about the functionality provided by the component. It includes the externally observable elements of the component, which may include operations, events, and properties.

There is little to no consideration as to how or where these components are to execute, or what the component itself requires from other components or its environment. As Han [12] and Szyperski [23] point out, the following information is required to make informed decisions about selecting and reusing software components:

- **Operational Specification:** This is the semantic interaction of the component. At a minimum, this should include an operational specification of the functionality of each operation included in the component's signature. Not only should this describe constraints on the elements of interfaces of the component, it should also describe the constraints on the relationships between these elements. Degree of support in the component for polymorphism and re-entrant conditions are other examples of information that may be useful.
- **Operation Context:** Here we need to describe where and how the component will be used. At a minimum, the 'where' information should include any assumptions made about the environment in which the component is expected to execute. The 'how' information should include expectations on component usage, such as the roles it is expected to play in a component system.
- **Non-functional Properties:** This would describe the properties such as performance, security, and reliability. These properties pose problems in specification.
- **Required Interfaces and Resources:** This describes the functionality and resources (e.g., memory and processing) required by the specified component to perform its provided functionality. One of the most neglected areas of component specification has been identifying the needs of the component itself: the dependencies on other components and environment resources.

All the categories are important for performing component selection. However, the first two categories and the provided interfaces together describe the information one typically would use to determine if an available component provides the required functionality and that the target environment will not pose a problem to the available component. Ideally, this process, known as specification matching [33], should be automated to the greatest extent possible. The last three categories contribute to the question of reuse, with the final category contributing the primary information (assuming that the component in question has already met the selection criteria).

3. Approaches to Component Specification

Because of the large body of work in the area of component specification, this is not an exhaustive survey. We categorize the surveyed approaches in terms of (1) early approaches, (2) design-by-contract methods, (3) formal methods, and (4) framework based approaches.

3.1. Early approaches

Early on, components were specified using only their interfaces. The early versions of the Object Management Groups' CORBA [25] and Microsoft's COM/COM+ [11]

[1] Meaning most used or most cited

component frameworks relied entirely on interface definition language (IDL) to specify a component. The IDL specification contained a description of the operations and types that the component used in its interfaces, including the exact signature of the operation with its parameters. IDL was typically written in a programming-like language (both the CORBA IDL and COM IDL languages were based on the C++ language syntax, as that was the prevalent object-oriented programming language at the time), and was therefore easy to understand. IDL was also developed directly for tool support, having IDL compilers. IDL specifications are easily extended. They enable runtime enforcement of the interface and provide direct support for polymorphism by allowing interfaces to be inherited by other interfaces.

With the introduction of the CORBA Component Model (CCM) [26] by the Object Management Group and the .NET Framework [24] by Microsoft, both groups moved away from having IDL as the only method of specification. CCM extended its model to include provides and requires interfaces, event sources and sinks, and attributes. The .NET Framework went away from explicit specification of IDL choosing instead to rely on the components themselves to supply the necessary information (extracted via .NET's Common Language Runtime).

Another popular method for writing specifications is free-text. This is a carryover from more traditional programming methods, from the days of libraries of procedures and, more recently, of standard template libraries. The Swing library of GUI components available as part of Sun Microsystems Java^{TM} releases is specified using JavaDoc, a web-based specification method that is the de facto method of learning about Java components (as well as the Java library itself). Free-text-based specification is still popular because of two reasons. It is human readable and understandable, and with the advent of web hyperlinks, even easy to use. Free-text allows the component implementer to specify just about anything they wish about the component, to any level of precision.

Evaluation: From the perspective of automated component selection and reuse, these methods do not contribute anything. The IDL is only a description of the wiring required between components. It contains no operational specification, as it is useful typically only at the component implementation level. Reuse of components specified in IDL can only be accomplished by those experienced in both the environment in which the component was developed and the environment in which the component was to be deployed. The lack of pertinent selection and reuse information in the IDL specification almost always required that these two environments be actually the same (i.e., the same organization). In other cases, vendors supplied detailed analyses to indicate how much modification and configuration a component might need to operate in the customer's environment, and even then, considerable effort was needed to accomplish just that.

Free-text based specifications generally contain much more information about the component than just the interface signatures. The problem with this information is that it is not amenable to automated search – it is free-text. Also, since there is little or no format to the content of specification data and no method of verification outside of peer review, the levels of precision varied widely. Compared with IDL, free-text specification techniques do provide a good deal of reuse information. One disadvantage is that it requires that the component system engineer do a lot of reading.

3.2. Design-by-Contract

Another category of component specifications is one made popular by the Cheesman and Daniels component specification process [5] as well as by the Catalysis Approach of D'Souza et al. [7]. Both approaches employ design-by-contract methods, which are characterized by the assertion that a component can be precisely specified solely by the specification of operations in the interfaces offered by that component. These methods define the *contract* as a collection of assertions that describe what a component does and does not do. Not only do these assertions precisely describe the behavior of the operations in the component, they are also logical expressions of the *entities* in the information model of the component [15]. The key assertions of the contracts are invariants, pre-conditions, and post-conditions.

Design by contract methods generally specify the design of the component using a graphical language, most often UML [27, 28]. UML can be used, because almost all of the specification methods that fall in this category assume that the component developer will use object-oriented methods to design and implement component functionality. The processes described in [5] and [7] focus almost entirely on the development of the component. The extension mechanism of stereotypes in UML is used to build component specification architectures and component interaction diagrams. These models describe using UML, the interfaces provided by the component(s) in the model, and in the case of [5], the interfaces required by the component. Some design by contract methods may develop behavioral diagrams, such as collaboration diagrams, interaction diagrams, or even state machines that have been applied at the component, rather than object level. The heart of this UML specification is a model containing a set of class elements, stereotyped as components, each naming an interface, and each containing a list of operation signatures. These are the provided interfaces of the component.

The resultant component specification is specified in UML, with the contract constraints specified in some formal language; typically, they are expressed in OCL [31]. This results in a component specification using a graphical model with textual attributes attached to specific operations in the model.

In the description of their specification method, Liu and Cunningham [15] state that their method provides two complementary purposes: (1) to specify expected functionality required by a component to fill a specific need, and (2) to specify the actual functionality implemented in an available component. They, however, provide no reasoning for this position.

The Catalysis Approach offered by D'Souza *et al.* [7] employs an action language to specify operational constraints instead of using OCL. The claim here is that the action language offers fully operational specification capabilities.

Kim et al. [14] offers a methodology which, like D'Souza's, also employs an action system, instead of static OCL constraints. They state that the difference between their approach and others in this category is that they use this action system to express dynamic configuration properties of the component. In their method, components are again modeled in UML, defining object relationships within the component. Once interfaces are developed, the dynamic operational specification of each operation in the interface is specified in simple statements in an action language based on *existence dependencies*[2]. To obtain the rigor desired by the authors, these statements are then translated into Petri nets [13] to take advantage of formal analysis techniques. These Petri net specifications are then used to build an execution model. They state that their method offers theoretical means for describing and analyzing both structural and behavioral aspects of components.

Evaluation: The design by contract methods described above have shortcomings when it comes to completely specifying a component. First, the authors do not take into account the environment in which a component must run. The narrow view offered works well only in homogeneous environments within the same application development group. Environmental support is an important specification detail in that configuration information such as security, transactions, and persistence must be known to use a component. Secondly, even when these methods are extended to accommodate this feature, they typically do not address the different layers of the component specification (component, framework, and architecture). Next, the methods generally offer a weaker operational specification than others. Research like that found in [14] and others are generating better operational specifications for this approach, but they do not support component selection and reuse. Lastly, with the obvious development-centric focus in these approaches, the specifications, especially the constraints, depend heavily on the internal elements of the component. Constraints often are expressed in terms of a property that is internal to the component (that is, it was not used in the operation found in the interface). Other issues include lack of support for nested components.

[2] See Kim *et al.* [14] for their use of existence dependencies

3.3. Formal Methods

Unlike the design by contract approaches discussed previously, component specification via formal methods tends to focus more on verification of the specified model and component reuse. Formal specifications provide precise descriptions of problem requirements, component function, and component structure [20]. Formal inference defines a mechanism for reliably and formally comparing problem requirements and component specifications. However, most specification methods concentrate their efforts on the structural and operational aspects of the interfaces of the component.

Penix and Alexander [20] concentrate on formal specification of the component inputs and outputs using axiomatic expressions specifying the pre-conditions and post-conditions on the inputs to and outputs of, respectively, the component interface operations in terms of the components' modeled domain and range. They also allow for constraints to be specified not only for specific operations, but for the component as a whole. Properties describing the environment in which the component will execute are given only cursory attention, and there is no discussion of the dependencies between the specified component and other system elements.

Morel and Alexander [19] also offer an approach formalizing the pre- and post-conditions of component interfaces. Their motivation is the development of a component adaptation framework, stating that matches found between components in the selection process will never likely meet all the requirements. Thus, closely matching components will need to be adapted in order to be assembled into the target component system. They have extended the ideas in [20] to include system level properties. The component specifications are written in Rosetta, which is a systems level design language for modeling heterogeneous systems. A facet specifies a particular aspect of a system or component. Facets can describe behaviors and more general requirements. A facet operates in a declared domain (system environment), which defines semantics available to the facet.

Chen and Cheng [6] state that one of the major contributors inhibiting component reuse is *architectural mismatch*, or a disconnect between the assumptions made by the com-

ponent about the environment in which it will execute. They state that their method alleviates this mismatch by formally specifying both the architectural and functional aspects of components, using the LOTOS and ACT ONE specification languages. Like [20], the motivation for their method was to allow automated reasoning of components to facilitate their reuse in systems. In their method, a component is viewed as a black box containing a set of input and output gates, where visible events occur. For the component, the set of *dynamic* behaviors is specified. A behavior describes not only a capability of the component (a *functional* aspect), but also the method in which the capability is provided to the client (an *architectural* aspect). In other words, the architectural aspect describes the protocol governing interaction with other components. The method encapsulates each behavior in a syntactic unit termed a *port*. Constraints imposed on a specific port and imposed on the component as a whole are also modeled. The complete set of ports and constraints constitute the *interface* of the component.

Petri nets form the basis of the formal component specification method of Chackhov and Buchs [4]. Again, the focus in on the functionality of the components offered by its operations. However, the authors have added support for modeling distributed systems, both in abstract and concrete terms, thereby beginning to include environmental properties into component specifications. Their model was developed initially to work on the JavaBeans component model.

Evaluation: While this formalism does provide specification that is amenable to *automated* reasoning and analysis, and therefore promotes an automated selection process for reusable components, these advantages are offset because the information being modeled in the specifications is still incomplete. While some dependencies are being considered, a general method for specifying "requires" information has not been developed in any of the methods studied. Also, the support for environmental and architectural aspects of the component, having had a good beginning, need to be characterized and extended.

Additionally, specification by formal method becomes increasingly difficult as the complexity and size of the item being specified grows. Even in the above methods, where formal specification is limited to a single port (behavior) of a component, specification of a behavior can be complex simply because of implicit interactions and dependencies with other behaviors provided by the component. Also, use of formal specifications would require that the component itself be tested to see that the implementation of the component meets the specification, and that the specification be "proven" to model the desired behavior. Both of these tasks are currently known difficult problems in software engineering research. Lastly, the current offered approaches in formal component specification are weak in support for non-functional requirements and none of the methods studied provided a mechanism to support polymorphism.

3.4. Frameworks

Frameworks offer a foundation for methods of component specification. They direct the contents and format of specifications, but do not dictate a process or prescribe methods to obtain the information that makes up the contents of the specifications. Frameworks are far more comprehensive in defining the amount of information required for a component specification because they focus on generalizing component operation in a specific environment.

Three prominent commercially available frameworks (.NET, CCM, and EJB) [25, 26, 17, 24] focus on implementation and development of components that are constrained to the environment for which they were developed. Even though they provide a good set of *practical* solutions to many of the issues facing component development, there is a lack of support for component selection.

The burden of selection should be easier in these environments, at least for specifically targeted components, because the environments share a common set of environment and system assumptions, and configuration properties. It seems, however, that since these component frameworks have been implemented by different vendors (excepting COM/.NET), the burden of selection was not apparent, as component 'experts' were able to 'select' components based on experience and advice from others.

Within their own environments, reusability of components is fairly good, due to the set of common assumptions on the environment. Even between different implementations of the same framework (e.g., CORBA), component reuse here is better than the general case.

We look at two frameworks to see what they offer in the context of specifying components for selection and reuse.

Han [12] offers an approach to software component specification containing four specific sets of information. First, the properties, operation, and events of the component form the *signature* of the component interface. Second, *constraints* further restrict and precisely define component interface. Han states that the signature and the constraints together characterize the *component capability*. The third set of information is component *usage in context*. Configurations are defined based on the component usage scenarios. A configuration identifies the roles and defines the role-based interfaces of the component in a given use context. Lastly, the component's *non-functional properties* are useful in assessing the component's usability in given situations and in analyzing properties of the enclosing systems.

A framework whose primary focus is the architectural connectivity between components, proposed in [29] and [21], is named SOFA, for SOFtware Appliances. Self de-

scribed as a platform for components, SOFA views applications as hierarchies of nested components, wherein each component is either primitive or composed of primitive components. As a result of hierarchy, *all* functionality is in primitive components. There is no direct mechanism for inheritance of components. The SOFA component model specifies several framework-level features. First, SOFA specifies interfaces, much in the way as has been done in other methods and approaches. It then defines Frames, which are black-box views of the component, containing a view of the *provided* and *required* interfaces plus some component configuration parameters. Next is the Architectures, more of a gray-box view, which define the *ties* between the subcomponents of the modeled component. SOFA defines four types of ties: *binding*, wherein a requires-interface is bound to a provides-interface; *delegating*, wherein a provides-interface of component is bound to a subcomponent's provides-interface; *subsuming*, wherein a subcomponent's requires-interface is bound to a requires-interface of component; and *exempting*, wherein the interface of a subcomponent is exempted from any ties. Non-exempted ties are realized via a connector, which models the interconnection between components, implements interaction semantics, and takes account of the deployment details.

In addition to these structural specifications, SOFA utilizes behavior protocols. These protocols represent the formal capture of communication between components, modeled as events (specifically emit method call, accept call, emit return, accept return). Sequences of events are called a trace. SOFA then defines that the behavior of an entity (an interface, frame, or architecture) is the set of all traces which can be produced by that entity. All the information is captured using SOFA's Component Description Language, or CDL. CDL is like the IDLs we have seen above, but has extensions to support frames, architectures, and the behavior protocols. These extensions make CDL a rather complex specification language.

Evaluation: Although the framework offered by Han [12] seems to include much of the information needed to enable component selection and facilitate reuse, he offers little in the way of details, and does not address the problems of support for polymorphism, re-entrant conditions, explicit specification of dependencies (requires) information; information that is necessary for both selection and reuse. As for the non-functional properties, Han is in agreement that such specification is currently a difficult problem, and as such should not cause undue delays in development of specification in the other areas listed in his research.

SOFA, as a framework, is quite powerful. It accounts for different layers of specification. With its concept of ties and connectors, it directly supports the specification of composition of components. However, SOFA too is incomplete with respect to specification for component selection. It is currently questionable if its aggregation model (composition) fully supports polymorphism. Specification of environment dependencies not related directly to interfaces seems to be missing. Overall, this architecture-centric approach seems to de-emphasize the individual components, failing to mention any provisions for 'contracts'-like specification between the components themselves.

4. Findings and Recommendations

4.1. Information about Required Interfaces and Resources

This must be a high priority area for the advancement of component specification techniques in the context of component selection. In many cases, this information is left outside of specification methods, (1) to be defined implicitly by framework defaults, or (2) to be determined from configuration parameters or trial and error, or, (3) as most of the time components were built and used by the same organization, to be known by the experienced designer, developer, or assembler.

Component specification approaches should include all required functionality and resources needed by the selected component. Listing all the dependencies that the selected component has with its environment (components and any other computing resource), should greatly reduce the burden of assembling the component into a target system. For instance, assembly and testing are much more difficult if the component being composed into the system implicitly delegates some of its functionality to another component based on how the system is configured. Explicit specification of this delegation behavior places more information in the hands of both the individual responsible for setting configuration properties of the selected component as well as the tester.

4.2. Co-locate Component and Specification

With all the available component specification approaches and techniques, almost none of them (the exception being IDL-based specification) included a mechanism for keeping the specification together with the deployable component. If components are to be independently deployable, there needs to be a way to bring together the component and its specification.

We need to develop mechanisms to maintain the component specification with the component itself, either as part of the component, or via some strongly linked external artifact. When a component gets selected and used, the spec-

ification must be obtainable using deployment policies or lookup services.

4.3. Specification in Terms of Component Internals

An important issue in component specification is the degree of exposure of internals of the component. Components are units of encapsulation and deployment. Still, some of the properties and types within a component must be exposed if they form part of the interface definition. Often, the actual *names* of the types and the operations found in the interface are used, for example, in the IDL and Design-by-contract based approaches. This is good for implementation of components, and definitely aids in the assembly of a component into a component system. But often, the specification of components includes references to internal elements of the component – elements that are not exposed in any of its interfaces. This usually occurs in the specification of constraints on the use of the component, typically a description of some state that the component must be in before an operation may be invoked [23].

Specification in these terms makes the process of selection more difficult, as we must now match a specification containing information based in the *implementation* of the component. It is likely that components that offer equivalent functionality would have different implementations.

Adding to this problem is the use of *names* in general. Component specifications typically express their interfaces in terms of operations using specific operation and type names. Again, components developed in different organizations would likely use different names for similar concepts. With different names for the same concepts, selection becomes even more difficult [22]. Matching elements of anything (model, component, program) using non-name-based information is currently a known, even more difficult problem in software engineering research.

In general, the formal methods specification approaches do not suffer from exposing component internals. These methods sometimes produce specifications that are quite removed from the actual element names and concepts in the component implementation (see discussion later in this section). These approaches typically express specifications using concepts, rather that exact names of the elements found in the implementation.

The CBSE community must begin to develop techniques and mechanisms that will allow specification of the behavior and constraints of a component without referring to specific type names and internal elements of the component. Techniques may use generally agreed upon concepts or *template placeholders* as used in the Aspect Oriented Modeling approach [8], which can be instantiated with specific names for each component to component matching process.

4.4. Non-functional Properties of Components

In general, the above discussed approaches provide very little support for automated selection that would include specifications for non-functional properties of software components. Even those that discussed the need for specifying these types of properties (see [12]) admit that this is a difficult area.

Specifying non-functional properties of software systems is a well known problematic area even outside the domain of CBSE. For any given non-functional property, a couple of issues need to be addressed. First, we need to agree on how to define and communicate the aspects of that property in the context of components. Second, we need to determine how to analyze that property in the context of the component system in which it will be assembled. Important non-functional properties are performance and security. Performance aspects of a single component can have far reaching impacts in the application in which it is operating, far beyond the components with which it interacts directly. Similarly, one insecure component may make the whole system insecure.

Franch [1] propose a notation, NoFun, to deal with non-functional properties of individual components, component libraries and systems. The approach can be used to select the best implementation of a component with respect to the non-functional needs. Frolund and Koisten [9] propose a quality of service modeling language, QML, for components. QML may be used at the time of design, implementation, or deployment of components. To use either method, one would require the use of a separate notation for the functional and non-functional specifications. Alves et al. [2] propose using a goal oriented requirements strategy and quality models to support the matching of components during COTS selection.

4.5. Formal Methods

We found that those approaches that provided specifications the most precise information for selection were those that provided the least amount of information for reuse. At this time, these approaches are the ones that employ formal methods as the primary specification technique. Formal methods, having a good deal of tool support, provided specifications that are currently the most amenable to automated specification matching [33]. For this reason, formal methods based approaches may be used to develop component specification approaches that also facilitate component reuse.

Since there is a perception in the community of practicing designers and developers that use of formal methods comes at too high of a cost (time, understanding,

etc.) [12, 23], the following approach may mitigate the cost of formal methods in component specification:

- *Start with Simple Components.* – Less time should be required to develop a formal methods-based specification for simple components, and the specifications themselves should be easier to understand.

- *Provide Support for Layers.* – Formal methods-based approaches to component specification should include support for the three prominent views or layers concerning components: the component and implementation layer, the framework layer, and the architectural layer. Formal specification methods should be extended to include support for these layers. This could take form of direct layer support, or a closely linked set of three separate specifications.

- *Provide Support for Polymorphism.* – Formal component specification methods need to support polymorphism [23]. This does not necessarily mean that the formal method needs to support inheritance. We need to explore different mechanisms to provide support for polymorphism without requiring inheritance. For example, the concepts of ties and connectors from SOFA [21, 29] could provide a starting place for such a mechanism.

- *Composition Mechanism for Components* – To aid in the reuse of components, we must also supply generally accepted component composition mechanisms. A standard method of composing components will enable more reuse and thereby increase our understanding of the entire CBSE space.

- *Composition Mechanism for Specifications* – If components are to be composed, and then selected and redeployed as *composite* components, there needs to be a complete component specification of this new composite component. Therefore, we need to also develop a mechanism to compose component specifications. Such specifications need to be as complete and rigorous as those from which the composite component was composed.

- *Component Indexes and Repositories* – Once we have developed components and specifications amenable to selection, we need a way of looking for sets of components from which we may select candidates for reuse. First, we need a way of locating the specification portion of a software component, so that we might proceed with our selection process. Some registration and cataloging process not unlike that for hardware components or even library books would provide the initial functionality that is needed. This would result in a searchable index of component specification information. Once selected, the index information would indicate the mechanism to obtain the actual deployable component from one of a set of many repositories. Again, the model of a library of books comes to mind. There is some initial work going on in this area, including in the commercial sector [16, 18, 30].

- *Support for Component Versions* – To support component search and selection, we need to maintain the ability to select components and their specifications by their versions [23]. In order to support component indexes and component repositories, we also need to develop a mechanism to distinguish components solely by their version or revision number. Some initial work is being done in this area [3, 10], but again, the final outcome of this area depends heavily on the approaches and methods developed for complete component specification.

5. Conclusions

As companies start developing component repositories for their own use or for selling components to third parties, techniques must be developed for automated component selection and reuse. This paper presented the issues that must be addressed as we look to develop complete software specifications that both enable automated selection of software components and facilitate the reuse of those selected components. Some recommendations have been made that may ease the advancement of development of component specifications.

Certain aspects of research in the area of formal method specification and analysis could positively affect the development of component specifications if such techniques were made more accessible via tools that capture and present information in a more human-understandable way to commercial component designers and developers. Commercial acceptance is equally important as is the academic research needed to develop and validate such methods.

References

[1] *Systematic Formulation of Non-Functional Characteristics of Software*. IEEE Computer Society, 1998.

[2] C. Alves, X. Franch, J. P. Carvallo, and A. Finkelstein. Using goals and quality models to support the matching analysis during cots selection. In *Proceedings of the 4th International Conference on COTS-Based Systems, Bilbao, Spain, February 2005*, volume 3412 of *Lecture Notes in Computer Science*, pages 146–156. Springer, 2005.

[3] P. Brada. Component revision identification based on idl/adl component specification. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held*

jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, pages 297–298, New York, NY, USA, 2001. ACM Press.

[4] S. Chachkov and D. Buchs. From formal specifications to ready-to-use software components: The concurrent object oriented Petri Net approach. In *Proceedings of the International Conference on Application of Concurrency to System Design, 2001*, pages 99–110, June 2001.

[5] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component- Based Software*. Addison Wesley, 2001.

[6] Y. Chen and B. H. C. Cheng. Formally specifying and analyzing architectural and functional properties of components for reuse. Proc. Eighth Annual Workshop on Software Reuse (WISR8), Columbus, OH, March 1997.

[7] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[8] R. France, I. Ray, G. Georg, and S. Ghosh. Aspect-based approach to design modeling. *IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 15(4):173–185, 2004.

[9] S. Frolund and J. Koisten. Qml: A language for quality of service specification. Technical Report HPL-9810, HP Labs, February 1998.

[10] J. Gergic. Towards a versioning model for component-based software assembly. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003).*, pages 138–147, September 2003.

[11] A. Gordon. *The COM and COM+ Programming Primer*. Microsoft Technology Series. Prentice Hall PTR, New Jersey, 2000. ISBN: 0-13-085032-2.

[12] J. Han. An approach to software component specification. In *Proceedings of 1999 International Workshop on Component Based Software Engineering*, May 1999.

[13] K. Jensen. A brief introduction to colored petri nets. In *Proc. Workshop on the Applicability of Formal Models, 2 June 1998, Aarhus, Denmark*, pages 55–58, 1998.

[14] H. H. Kim, D. K. Kim, H. T. Jung, Y. D. Chung, and D. H. Bae. An operational component specification method. In *6th Asia Pacific Software Engineering Conference, 1999. (APSEC '99)*, pages 38 – 45, December 1999.

[15] Y. Liu and H. C. Cunningham. Software component specification using design by contract. In *Proceedings of the South-East Asia Software Engineering Conference*. National Defense Industry Association, Tennessee Valley Chapter, April 2002.

[16] D. Lucredio, A. Prado, and E. de Almeida. A survey on software components search and retrieval. In *Proccedings 30th Euromicro Conference, 2004*, pages 152–159, August–September 2004.

[17] V. Matena, S. Krishnan, L. DeMichiel, and B. Stearns. *Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform*. The Java Series. Addison-Wesley, 2nd edition, 2003. ISBN: 0-201-91466-2.

[18] R. Meling, E. Montgomery, P. S. Ponnusamy, E. Wong, and D. Mehandjiska. Storing and retrieving software components: a component description manager. In *Proceedings of the Australian Software Engineering Conference, 2000*, pages 107–117, April 2000.

[19] B. Morel and P. Alexander. SPARTACAS: automating component reuse and adaptation. *IEEE Transactions on Software Engineering*, 30(9):587–600, September 2004.

[20] J. Penix and P. Alexander. Using formal specifications for component retrieval and reuse. In *Proc. of the 31st Hawaii International Conference on System Sciences, 1998.*, volume 3, pages 356–365, January 1998.

[21] F. Plasil. Enhancing component specification by behavior description - the SOFA experience. In *Proc. 4th Int. Symp. on Information and Communication Technologies (WISICT)*, pages 185–190, January 2005.

[22] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, 2001.

[23] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, London, 2nd edition, 2002.

[24] T. Thai and H. Q. Lam. *.NET Framework Essentials*. O'Reilly, 2nd edition, Feb 2002.

[25] The Object Management Group. The common object request broker: Architecture and specification, revision 3.0. Internet, 2002. formal document: 02-06-33 (`http://www.omg.org/`).

[26] The Object Management Group. CORBA component model (CCM). Internet, June 2002. `http://www.omg.org`.

[27] The Object Management Group. Unified modeling language specification, version 1.5. Internet, March 2003. Available from `http://www.omg.org/uml`.

[28] The Object Management Group. Unified modeling language: Superstructure, version 2.0, final adopted specification. Internet, August 2003. Available from http://www.omg.org/uml.

[29] The SOFA Website. The SOFA component model. Internet, 2002. Available at: `http://nenya.ms.mff.cuni.cz/projects/sofa/tools/doc/compmodel.html`.

[30] P. Vitharana, F. Zahediand, and H. Jain. Knowledge-based repository scheme for storing and retrieving business components: a theoretical design and an empirical analysis. *IEEE Transactions on Software Engineering*, 29(7):649–664, July 2003.

[31] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language — Precise Modeling with UML*. Addison Wesley, 1998.

[32] R. Zahavi. *Enterprise Application Intergration with CORBA: Components and Web-Based Solutions*. OMG Press. John Wiley & Sons, 2000. ISBN: 0-471-32720-4.

[33] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. Software Engineering Methodology*, 6(4):333–369, 1997.