Support for Evolution of Systems Through Software Composition

R. Pandey
CS Department,
University of California, Davis

V. Akella
ECE Department
University of California, Davis

P. Devanbu
CS Department,
University of California, Davis

{pandey@cs, akella@ece, devanbu@cs}.ucdavis.edu

1 Introduction

A typical difficulty in maintaining large systems is non-localization. Changes that appear to be *prima facie* local and simple can cascade unexpectedly through the system, greatly multiplying the effort required to effect the change. The goal of *evolvable systems design* is to seek design methodologies, architectures, and implementation techniques that can reduce this cascading effect. We seek to ensure that small changes in a system's functionality require only small changes in the implementation–we want changes to be *proportional*. Consider changes to a system assembled from components which are interconnected with some connection medium. There may be *direct* changes, which directly modify a component that directly implements some functionality. Such changes are usually proportional. However, *indirect changes*, on components that are not directly involved in the functionality, but affected due to a direct change, may also be required. For example, consider the task of adding a new, more efficient type of indexing to a database to improve performance. This will surely require a direct change to the data management subsystem. If this also requires changes to security and integrity mechanisms, or to the concurrency control and locking mechanisms, these are considered indirect changes. The target components for direct changes may usually be learned from design documents or even from models represented in design formalisms. However, targets of indirect changes are not usually directly inferable, and are often only identified from faults uncovered during unit or integration testing. Thus, indirect changes can increase costs, cause delays, and reduce quality.

We are concerned with techniques which support all stages of the process of developing evolutionary systems. Specifically, we are interested in addressing the following:

- Software design phase: How should a software system be structured so that the system can be easily changed?
- Evolution phases: How can the elements of a software system that have changed be identified automatically given some direct changes?
- Maintenance phase: How can support for predefined actions (such as system replacement, compilation, verification, and testing) be integrated so that many of the implied changes can be effected automatically in response to changes in the system?

This paper briefly describes our approach for supporting evolutionary software design. We show that designs of software systems are more amenable to changes if specifications of components and interactions among components are completely separated. The separation allows one to limit the scope of indirect changes. Further, we model interactions among components in terms of relationships which not only capture interactions but also form the basis for detecting the scope of changes and for applying corrective changes.

The goals of our research are identification of various kinds of interactions and their representations within a relational or constraint based framework, development of composition techniques that allow specification of both static and dynamic evolution of software systems, and development of tools that allow automatic analysis of components and relationships for implied changes and that allow corrective actions to be taken depending on the changes. In this position paper we only describe our general approach and an abstract model for specifying software design. The detailed design of the model is currently under development. We will present a preliminary version of the model in the complete paper.

2 Support for evolutionary software development and management

A complex system is composed of two kinds of entities: components and interactions. Although our primary focus is on interactions, we begin with a discussion of components.

2.1 Components

Components implement specific parts of a complex system. A component is defined in terms of an interface and an implementation. The interface includes two parts: a set of *input interfaces*, defining the services that the component provides, and a set of *output interfaces*, enumerating the services that the component uses. Note that each interface description contains a set of names and the semantics associated with the name. An implementation of the interface uses the various names within the context of the semantics associated with the names.

Example 2.1. *(Procedure signature).* Assume the following interface $R\ f(T_1\ p_1, T_2\ p_2, \cdots, T_n\ p_n)$ denotes the signature of a function f. Note that the signature defines both the names (or position) of the parameters as well as their semantics (types). The semantics specify how the names are used inside an implementation of `f`. ■

Example 2.2. *(Port).* In many software architecture description languages, the interface of a component defines the set of input and output ports through which an instance of the component may communicate with other components. The specification of a port typically includes its name and its semantics. The semantics of the port may specify valid operations (such as send and receive), synchronization behavior of the operations and other attributes (such as buffer size). ■

The semantics of an interface entity can be quite complex. We ignore the actual details of the semantics of names as it allows us to reason about the evolvability of software systems without worrying about the details. We will represent each interface as a set of (name, semantics) pairs in this paper. We show later in the paper that the nature of bindings of both name and semantics has implications on evolvability of software systems.

2.2 Interaction

Interactions define how components are semantically related to each other. They determine how components affect each other. Examples of interactions are synchronization relationships among concurrent programs, data connections among distributed programs, coordination among various requests to a file server, security constraints among two components, and real time constraints.

2.2.1 Role of interaction on evolution

Much of the complexity in software evolution arises due to the inability to determine easily the overall *impact* of a change on the system. Direct changes that affect only a single component or a small set of components are easier to handle, precisely because the scope of the change is *local*. Indirect changes that cause a *chain reaction* are difficult because of the subtle and complex dependencies between components of a system. This is because changes in a component may change the way it interacts with other components, thus forcing changes in its interactions; this may lead to changes in other components. An important step in reducing the complexity associated with detecting and applying changes is to understand the role of interactions in software system design. Specifically, the following questions need to understood and addressed: i) What is the general nature of interactions? What role does it play in evolvability of software systems? ii) How can interactions be represented? iii) How can their representation be used to *identify* changes in software systems? iv) How can their representation be used to *make* changes in software systems? We address these questions by presenting a model of interactions in the next section.

2.2.2 Relationships

We model interactions among components as relationships. The notion of relationship extends the notion of connectors [7, 1] of architecture description languages in order to include evolutionary characteristics of software systems. The primary goal of a relationship is to define how components can be composed together. A relationship is, thus, specified by defining the following:

i) Interface binding: The first part of the composition involves binding the input and output interfaces of different components. It includes matching (implicitly or explicitly) names in the interfaces and specifying (implicitly or explicitly) semantics for the names in the interfaces.

Example 2.3. *(Interface).* Assume that a procedure component, p, has an output interface (n_p, s_p), where n_p is the name of an entity it accesses and s_p is semantics associated with n_p. Assume that a procedure component, q, has an input interface (n_q, s_q). Procedures p and q are composed by binding names n_p and n_q and associating semantics for s_p and s_q. These bindings can be defined explicitly or implicitly. In the case of implicit bindings (used in many programming languages), n_p and n_q denote identical names, whereas s_p and s_q together may denote a procedure call semantics. ■

We will see that the ability to change p or q depends on the ability to change the bindings of names as well as their semantics.

ii) Attributes: Attributes define additional properties of a relationship. For instance, a data connection relationship may specify the size of the buffer used for communicating between two processes.

iii) Polarity of changes: The polarity of changes of a relationship R specify how changes in components or relationships that R combines affects the components or the relationships. For instance, if a relation R is used to compose components C_1 and C_2, its polarity specifies the components and/or relationships that need to be changed if C_1, C_2, or R changes. Polarity of changes can be used to determine the scope of changes in a system automatically. The notion of polarity of changes is similar to the dependency relationship specified between a source and a target of the *make* program. Also, it occurs in many programming language abstractions implicitly. An example is the relationship between the interface and implementations of an ADT and how the semantics of the relationship between the interface and the implementation is used for supporting separate and automatic compilation.

iv) Semantics of changes: The semantics of changes specifies the set of syntactic and semantic operations that need to be carried out if components or relationships change. It provides the basis for carrying out some of the changes automatically. For instance, a relationship R can specify that changes in the size of a shared buffer may require that a different algorithm be used for communicating between interacting components.

2.2.3 Interface binding and evolution

The ability to make modifications in a system depends on the ability to specify relationships clearly and precisely. Further, it depends on how different aspects of relationships are implemented. In this section, we argue that the ability to reduce the impact of changes depends on the ability to separate specification of relationships from specifications of components.

A relationship can be realized in many ways: it can be implemented within components, separately from components or through a combination of the two. Let us examine what it means to implement *interface bindings* in different ways and the corresponding implications on evolvability.

The first approach involves defining names and semantics of input and output interfaces inside components. An example of this is the case when a procedure, p, calls another procedure, q. The body of p binds the name of the procedure (q) and semantics of q (a synchronous procedure call). The implications of this binding are that changes in q involves changing body of p. Further, semantics of q in p is fixed as well. The second approach involves binding the names externally but the semantics internally. This means that procedure p can call many procedures by binding it with different procedures. Hence,

changes in the names of the procedure q will not require any changes in p. However, the semantics of procedure call is still fixed. The third approach involves binding both names and semantics externally. This gives the added advantage that the semantics of calls to q can be changed easily without requiring any changes in p; calls to q can become concurrent by associating a different semantics with q. Interaction between p and q can, thus, be changed without requiring any changes in p or q.

We therefore see that as we reduce the amount of information that is bound inside components, changes can be incorporated relatively easily without requiring modifications in components. *The complexity in modifying or extending a complex software system can be reduced if components and relationships are specified separately* [4]. Separation of implementations of components and relationships allows one to evolve implementations of complex software systems relatively easily. Separating the components and their interactions relationships localizes the possible changes that may need to be made when system evolves.

In addition to localizing changes, change polarity and change semantics information can be used to automatically determine the relationships and components that may need to be changed. Further, the associated corrective actions can be used to effect changes. We are currently looking at mechanisms for different kinds of relationships, how they can be specified separately from components, and how we can introduce the notion of change polarity and change semantics in their specifications.

3 Related work

The problems and costs involved in evolving a software system to meet new requirements are well-understood in the software engineering community. In the past the focus was primarily at the level of programs (code) and has been subject to substantial research in the software reuse world. Recently, it has been recognized that more substantial savings in effort and costs can be realized by focusing on a more abstract view of the software system, i.e. at the level of the structure and composition of the basic modules.

There are two prominent approaches to dealing with evolution of software systems in the current literature: (i) Software Architectures or Architecture Description Languages (ADLs) [5, 7, 3, 1] and (ii) Design Patterns [2, 6]. The work proposed in this paper builds on the key ideas embodied in both the these approaches. The software architecture research emphasizes the role of "connectors" in the development of software systems. Most modern ADLs have constructs to model connectors. Evolution is supported in some of the ADL frameworks by way of subtyping and refinement of connectors. Our goal is to support evolution by more detailed modeling of the syntactic and semantic bindings implicit in connectors; by "lifting" these binding mechanisms into the connector, we hope to reduce the impact of changes on components, and more carefully track and report indirect changes.

Design patterns [2] are used in OO systems as design primitives. By using patterns to solve known design problems, designers can increase understandability, evolvability, and modularity of the design. However, if maintainers make changes that fail to honor the conventions of a design pattern in a specific instance of the pattern in an OO system, these advantages evaporate. Consider, for example a *Template Method* instance ([2], page 325) instance: let's say that we have a "database-independent" base class `dibClass` which performs some operation `doResult` on a database: this method operation calls several virtual functions such as `evalQuery` and `findTuple`, which are not defined in the base class. To implement this base class, we define a database-specific derived class `dsdClass` derived from `dibClass`, and define the bodies of the virtual functions to perform the `evalQuery` or `findTuple` operations on a specific database (such as Oracle^{TM}). In this manner, the `doResult` operation in `dibClass` can remain platform-independent and ignorant of the details of the specific database. However, after the system is written and shipped, there may be *no* documentation that indicates that `dibClass, dsdClass`, and the above mentioned member function constitute a *Template Method* instance.

Now suppose a programmer, unaware of this design intention, changes `dibClass::doResult` to call some (non-virtual) member function of `dsdClass`, that may be implementation specific to some particular database. This breaks the database independence of `dibClass`, and introduces a potential source of bugs when `dibClass` needs to be targeted to another database. Our goal in modeling inter-component relationships is to allow designers to explicitly model maintenance rules; tools would then be able to detect the dangers of such modifications and alert maintainers so that other alternatives could be considered.

In summary, the key contribution of our work would be to identify the key ingredients of a relationship between the components which would facilitate the evolutionary design of software system, independent of the architecture description language. However, some the ingredients in a relationship could be domain-specific i.e. security, real-time, finance, networking, etc.

References

[1] CLEMENTS, P. A Survey of Architecture Description Languages. In *Proceedings of the Eighth International Workshop on Software Specification and Design* (1996).

[2] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Sofware*. Addison-Wesley, 1995.

[3] MEDVIDOVIC, N. A Classification and Comparison Framework for Sofware Architecture Description Languages. Tech. Rep. UCI-ICS-97-02, University of California, Irvine, 1997.

[4] PANDEY, R. *A Compositional Approach to Concurrent Programming*. PhD thesis, The University of Texas at Austin, August 1995.

[5] PERRY, D. E., AND WOLF, A. L. Foundations for the Study of Software Architectures. In *ACM SIGSOFT Software Engineering Notes* (Oct. 1992), pp. 4–52.

[6] SCHMIDT, D. A Family of Design Patterns for Application-Level Gateways. *Theory and Practice of Object Systems, Special Issue on Patterns and Pattern Languages 2*, 1 (1996).

[7] SHAW, M., AND GARLAN, D. *Software Architecture: Perspectives on An Emerging Discipline*. Prentice Hall, 1996.