This paper has been accepted
at ESEC/FSE ’99, Toulouse

Synergy between Component-based and Generative Approaches

Stan Jarzabek[1] and Peter Knauber[2]

[1]Department of Computer Science, School of Computing, National University of Singapore, Lower Kent Ridge Road, Singapore 119260; at sabbatical leave at Fraunhofer Institute for Experimental Software Engineering, Sauerwiesen 6, Kaiserslautern, Germany D-67661
stan@comp.nus.edu.sg

[2]Fraunhofer Institute for Experimental Software Engineering, Sauerwiesen 6, Kaiserslautern, Germany D-67661
knauber@iese.fhg.de

Abstract. Building software systems out of pre-fabricated components is a very attractive vision. Distributed Component Platforms (DCP) and their visual development environments bring this vision closer to reality than ever. At the same time, some experiences with component libraries warn us about potential problems that arise in case of software system families or systems that evolve over many years of changes. Indeed, implementation level components, when affected by many independent changes, tend to grow in both size and number, impeding reuse. In this paper, we analyze in detail this effect and propose a program construction environment, based on generative techniques, to help in customization and evolution of component-based systems. This solution allows us to reap benefits of DCPs during runtime and, at the same time, keep components under control during system construction and evolution. In the paper, we describe such a construction environment for component-based systems that we built with a commercial generator and illustrate its features with examples from our domain engineering project. The main lesson learnt from our project is that generative techniques can extend the strengths of the component-based approach in two important ways: Firstly, generative techniques automate routine component customization and composition tasks and allow developers work more productively, at a higher abstraction level. Secondly, as custom components with required properties are generated on demand, we do not need to store and manage multiple versions of components, components do not overly grow in size, helping developers keep the complexity of an evolving system under control.

1 Introduction

The interest in Distributed Component Platforms (DCP) [6,17,22] grows and vendors bring to the market many software development environments for building software systems out of pre-fabricated components. Most often components are provided in a binary form. By complying to the standards of the underlying DCP, these components can be easily interconnected with each other. It is claimed that component-based software engineering (CBSE) will cut development costs due to high level of reuse.

CBSE will also produce highly maintainable systems: as a system is represented by a collection of cooperating components, maintenance will be done by replacing components with other components providing richer functionality. This is viewed as a potentially simpler task than today's maintenance of big, monolithic programs.

DCPs, such as EJB™, ActiveX™ and CORBA™ implementations, offer many advantages for deployment and efficient execution of software systems, particularly systems that operate in a distributed environment. However, it still remains to be seen too what extent we can build maintainable large scale software systems out of implementation level components. Experimental studies should address specific questions such as: Which design objectives can be achieved with component-based approach and which cannot? What kinds of software systems (and which parts of them) can be built with independently developed components? Will a component-based system be still under control after a couple of years of maintenance?

Answers to these questions depend on a specific context in which questions are asked. The context also determines the meaning of basic terms of component, architecture and CBSE. For example, the required properties of components and architectures depend on whether we talk about program construction components or components of a runtime architecture. In many component-based approaches, components play a double role of construction components and also implementation, runtime components. However, different issues matter in program construction than during the runtime. The main trust in construction architectures and their components is flexibility, i.e., the ability to customize components to meet variant requirements of products and the ability to evolve over time to meet changing needs of the business environment. For construction components, we need to know how different sources of change affect them. On the other hand, issues that matter in runtime architectures include allocation of functions to components, deciding which logical components should be packaged into a single executable, parallel execution of components, data communication, invocation of services between components and synchronization. Those are two different perspectives that require two different mechanisms to effectively deal with related problems. While a unified program construction and runtime world prove very useful in rapid application development, we still do not have evidence that it is sufficient for development and evolution of large scale software systems.

Certain problems seem to be inherent to approaches in which we attempt to build systems out of pre-fabricated implementation components. Over years of evolution of a software system and also during customization for reuse, components are affected by independent sources of change [2,5]. Sources of change steam not only from new (or variant) functional and non-functional requirements, but also from new versions of a computing environment such as tools, operating systems and networks. If we need maintain a component version for each combination of these variants, components will grow in size and number. The cumulative effect of this uncontrolled growth may likely become prohibitive to reuse. Repositories with version control to store components may ease but will not solve the problem. This phenomenon is known to companies who have been using component libraries for some time [5].

The above problem can be avoided if we change the focus from the artifact to process, from customized components to customization process itself. Generative techniques allow one to create on demand a component with a required combination of variants, so there is no need to manage component versions. The basic idea would

be to keep specifications of how variants affect a generic component separately from the very component [2]. A generator interprets specifications of required variations and produces a custom component when it is needed. This scenario promises a way to avoid uncontrolled growth of components in size and number and also to automate some of the routine customization activities.

In this paper, we illustrate the above mentioned problems with examples from our domain engineering project. We show how we alleviated the above problems by coupling component-based runtime architecture with construction environment based on generative techniques. We think there are some general lessons we learnt from the project. Strengths of generative techniques lie exactly where the open problems, and perhaps inherent weaknesses, of the component-based approach are. Generators are built to provide an effective way of dealing with variations in a domain. Variations are characterized in application domain terms, independently of the implementation, in particular, independently of the component structure of the runtime system. For given variant requirements, a generator can produce a member of a system family that satisfies the variants. The knowledge of how variants affect components, how to produce customized components and how to assemble customized components into a working system is within a program construction environment used by a generator. This construction environment includes customizable and compatible components organized within a generic architecture and global structures that help deal with changes. Coupling component-based approach with generative techniques allows one to benefit from distributed component architectures during runtime, without sacrificing flexibility required during program customizations and evolution.

The flow of the paper presentation is as follows. After discussing related work, we briefly describe a family of Facility Reservation Systems (FRS) and its component-based runtime architecture. We use examples from the FRS family to illustrate claims throughout the paper. In section 4, we discuss problems that arise if we develop and evolve FRS family in terms of implementation components such as in DCOM/ActiveX™, JavaBeans™ and CORBA™ implementations [17,22]. In section 5, we describe a program construction environment based on generative techniques. In this solution, a generator derives runtime components for a custom system from a generic architecture of the construction environment. Concluding remarks end the paper.

2 Related work

We do not review the literature on CBSE and DCPs as we believe this will be done extensively during other conference sessions. The two requirements for component-based systems that we emphasize throughout the paper are ease of customization and evolution. These two requirements are particularly important in the software system family situation and our experiments indeed involved a family of facility reservation systems. Software system families arise in situations when we need develop and maintain multiple versions of the same software system (for example, for different clients). The concept of program families was first introduced by Parnas [23] who proposed information hiding as a technique for handling program families. Techniques described in his early papers mainly address variations in design decisions across family members. Since then, a range of approaches has been proposed to

handle different types of variations (for example, variant user requirements or platform dependencies) in different application domains. Pre-processing, PCL [27], application generators [3,20], Object-Oriented frameworks [14], Domain-Specific Software Architectures [28], frame technology [2] and, most recently, distributed component platforms [6,17] – they all offer mechanisms to handle variations that can be useful in supporting program families.

Parson and Wand [24] argue that implementation level objects (read: components) are not suitable for analysis, as the realm of analysis is different from that of implementation. Their discussion hints at problems that are inherent to any approach that focuses too early at the implementation components. Bridging the gap between domain concepts and implementation has been recognized by many researchers and practitioners as a major challenge in software engineering. Much work on software design, architecture, domain analysis and application generators stem from this premise. The problem can be addressed in top-down direction, from domain concepts to implementation components, or in bottom-up direction, with a software architecture as a possible meeting point. Top-down and bottom-up research directions contribute different insights to the problem. In the context of component-based approach, compositional (scripting) languages [21] allow one to group components into higher level abstractions that are closer to domain concepts than individual components. Domain analysis approaches attack the problem from the top. They allow one to identify domain concepts that might correspond to component groups. Moving in that direction also provides insights into types of variability that a component should be able to accommodate.

Generative techniques are based on domain analysis. A generator builds custom systems from a set of generic, compatible components. A domain-specific generic architecture, incorporating compatible components, is the heart of a generator. A generic architecture implements commonalties in a domain, while a meta-language allows developers to specify variations to be implemented in the custom system. GenVoca is a method and tool for building component-based generators. In JST [3], based on GenVoca, a meta-language extends existing programming languages with domain-specific constructs. JST provides practical way of bridging domain concepts and implementation.

The authors of papers [15] and [26] suggest that generative techniques can alleviate some of the problems with component-based systems we discussed in the introduction. Boca [10] is a generator for component-based systems. Boca provides a meta-language to define business semantics. Business components such as customers, orders, employees, hiring and invoicing are specified in the meta-language, separately from the runtime program characteristics. All the future changes in requirements can be done at the level of meta-descriptions. A meta-language provides means for maintaining integrity of requirements for a system family during customization and evolution. Boca supports synthesis of component-based runtime systems from business and implementation-specific component layers. Digre [10] points out further benefits of separating a software construction architecture from the runtime architectures in the context of distributed component platforms: runtime components contain both business logic and platform-specific implementation details (such as, for example, code for sending and listening to events specific to Enterprise JavaBeans™ platform). A construction architecture makes it possible to separate business concerns

from platform concerns. Not only will this make the design simpler, but also will make business components independent of changes to the platform technology.

Many computational aspects of a program spread throughout the whole program and cannot be nicely confined to a small number of runtime components. Examples include business logic, platform dependencies or codes that have to do with system-wide qualities such as performance and fault tolerance. In aspect-oriented programming [16], each computational aspect is programmed separately and a mechanism is provided to compose aspects into an executable program. It is envisioned that both program development and maintenance will be done in terms of independently defined program aspects.

Ability to deal separately with different computational aspects is, in our view, a general requirement for construction environments. Frame technology [2] fragments programs into generic components called frames. Frames organized into a hierarchy form a generic architecture, from which programs incorporating specific variants can be produced. A frame is a text written in any language (e.g. IDL or Java). Frames can be adapted to meet requirements of a specific system by modifying frame's code at breakpoints. Each frame can reference lower level frames in the hierarchy. This is achieved using frame commands (e.g. .COPY, .INSERT) that can add to, or subtract, from lower level frames. This "editing" is performed at distinct breakpoints in the lower level frames. Each frame can also inherit default behavior and parameters from higher level frames. Frames are organized into frame hierarchies for the purpose of constructing a member of a system family. The topmost frame in a frame hierarchy, called a specification frame, specifies how to adapt the rest of the frame hierarchy to given variant requirements. Therefore, each source of change can be traced from the specification frame throughout all the affected frames. A frame processor is a tool that customizes a frame hierarchy according to directives written in the specification frame and assembles the customized system. Industrial experiences show that while building a generic architecture is not easy, subsequent productivity gains are substantial [2]. These productivity gains are due to flexibility of resulting architectures and their ability to evolve over years. In [9], we described a frame-based generic architecture and customization method that we developed for a Facility Reservation System family. In this paper, we generalize experiences from our domain engineering projects, highlighting the advantages that generative techniques offer to component-based systems.

3 An FRS family and its runtime architecture

In this section, we briefly introduce our working example, a family of Facility Reservation Systems (FRS for short).

3.1 A brief description of the Facility Reservation System (FRS) family

Members of the FRS family include facility reservation systems for offices, universities, hotels, recreational and medical institutions. An FRS helps company staff in the reservation of rooms and equipment (such as PCes or OHPs). An FRS maintains records of reservations and allows users to add new reservations and delete

an existing reservation. When reserving a room, the user should be able to specify an equipment that he/she needs to be available in the room.

Here are some of the variant requirements in the FRS domain:

1. Different institutions have different physical facilities and different rules for making reservations.
2. In certain situations, FRS should allow one to define facility types (such as Meeting Room and PC) and only an authorized person should be allowed to add new facility types and delete an existing facility type.
3. The calculation of reservation charges, if any, may be performed according to discounts associated with each user and the payment classification of each facility.
4. Some FRSes should allow users to view existing reservations by facility ID and/or by reservation date.
5. Sometimes an FRS may need maintain a database of all the facilities in a company. One should be able to add new facility of a given type, enter facility description (if applicable) and delete an existing facility.
6. Some FRSes should also allow the user to search for available Rooms with the necessary piece of equipment.
7. Most often, users (individuals or companies) manage their own reservations with an FRS. In some companies, however, users send reservation requests to a middleman who makes reservations for them. In general, users may only perform certain functions according to the permissions assigned to them

Even this simplified description of the facility reservation domain hints variant requirements, dependencies among requirements and potential problems with handling them during the design and evolution of FRS components.

3.2 A component-based runtime architecture for FRS

We shall aim at the component-based runtime architecture for FRSes, depicted in Fig. 1. The architecture is a three-tiered architecture. Each tier provides services to the tier above it and serves requests from the tier below. Each tier is built of components. A tier itself is yet another large granularity component. The FRS client tier contains, among others, components for reservation, user and facility management. These components initialize and display Java panels and capture user actions. The middle tier, in addition to implementing FRS business logic, provides the event-handling code for the various user interface widgets (e.g. buttons). It also contains components that set up and shut down connections with the DBMS tier. User interface components together with their event-handling code and business logic counterparts form higher level components organized using PAC design pattern [12]. For example, “Reservation Management Panel”, “Reservation Management Logic” and event-handling components together form a higher level component “Reservation Management” (not shown in the diagram).

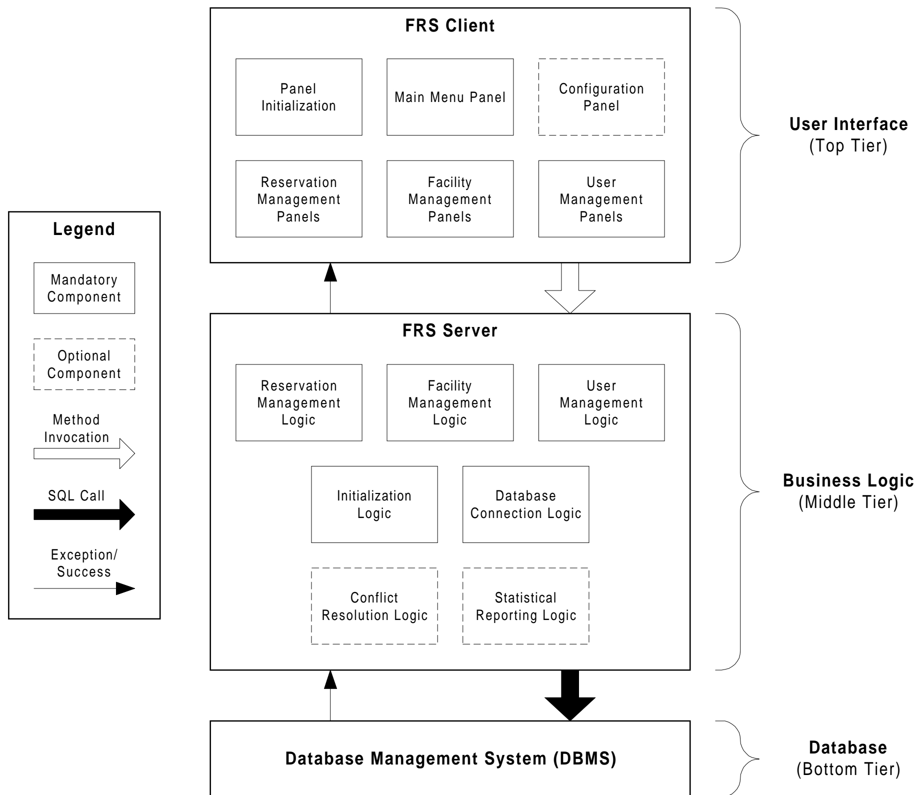

Fig. 1. Three-Tiered Structure of Runtime FRS Architecture

The FRS client and server tiers communicate via method invocation calls. The communication between tiers is detailed in Table 1.

Table 1. Description of Connectors

Connector	Description
FRS Client to FRS Server	Method invocations that are triggered as a result of user actions (e.g. pressing a button to reserve a facility) on the client. Examples are adding a reservation and viewing the details of a facility.
FRS Server to FRS Client	May be reservation, user or facility data that is returned as a result of a method invocation. In some cases, error or exception data may be returned.
FRS Server to DBMS	SQL call to retrieve data from one or more databases managed by the DBMS. An example of such calls could be a request to find the largest existing reservation ID.
DBMS to FRS Server	Data from the DBMS' database(s) that is returned as a result of some SQL call.

4 Supporting an FRS family with runtime component architecture

A development situation described in this section reflects capabilities of DCPs, such as EJB™ or ActiveX™, and their visual software development environments. An FRS runtime architecture is depicted in Fig. 1. Facilities are provided to introspect and customize components [17].

4.1 Addressing variant requirements

To see how developers would customize and evolve FRS components, consider variant requirements listed in point 4 of section 3, namely viewing reservations by facility and reservation date. These are anticipated variant requirements, therefore they should be implemented within the components of FRS architecture. Components relevant to the viewing reservation requirement include user interface panels for selecting reservation viewing options and displaying the results and functional components (middle tier) implementing retrieving reservations from the database. All the anticipated variant requirements must be implemented within these components. A developer will customize components (for example, by setting property values) to indicate which reservation viewing methods and which event handlers are needed in a custom FRS. After customization, components will reveal only required functions.

There are two problems with the above method of implementing variant requirements. Firstly, if a given variant requirement affects many components, it will be up to a developer to keep track of all of these components. This may not be easy for large architectures and variations that affect many components. Secondly, components will grow in size as all the variants must be implemented within them. Customized FRSes will also contain implementation of all the variants, even though some of these variants will never be used.

4.2 Evolving an architecture

Suppose now we wish to evolve the FRS architecture to accommodate a new requirement that had not been anticipated before. For this, assume we want certain FRSes to produce a list of all reservations for a specified user. As we think that many FRSes will need this feature in the future, we wish to include the new requirement into the FRS architecture for future reuse.

The following plan leads to implementing this new variant requirement: the "Reservation Management Panels" component in the top tier (Fig. 1) provides functionality for building the menu for viewing reservations. It also displays the available retrieval methods for reservations and elicits the user's choice of retrieval method. Based on the user's choice of retrieval method, this component calls the appropriate function from the "Facility Management Logic" component (middle tier) that implements the logic for a given retrieval method. Under the list of available retrieval methods, we must add an additional choice "view reservations made by user". We also need augment middle tier components to include event handling code for the new retrieval method and add implementation of a new function that retrieves reservations from the database into the "Facility Management Logic" component. Implementation of the new requirements affects interfaces of some components. In particular, we need type declarations for a data structure to retrieve reservations for a particular user and interface declarations for operations that will allow the FRS client to obtain a list of registered users and reservation data.

If the source code for the user interface and "Facility Management Logic" components is not available, there is no simple way to implement this new requirement. We may need to re-implement affected components, as components cannot be extended in arbitrary ways without the source code. If the source code for relevant components is available, we could use either inheritance or a suitable design pattern [12] to create new components that would include functionality for viewing reservations by user ID. Any visual environment supports the former solution and some support the latter [25]. While this method of addressing new requirements is sufficient in the rapid application development situation, it presents certain dangers in the context of system families. Over years of evolution, an architecture may be affected by many new requirements. Implementation of new requirements will add new components (or new versions of old components) to the architecture. As certain requirements may appear in different combinations – we may end up with even more components. As a result, our architecture may become overly complex and difficult to use.

To illustrate the above problem, let us assume that, over years of evolution, four sources of change have affected a component X. These changes might include unexpected requirements or changes in computing technology. If these changes were independent of each other, then chances are that we shall have four new versions of component X in the architecture. If it also happens that the four new requirements are optional and may appear in any combination in the systems built based on the architecture, then we might have as many as $2^4=16$ versions of component X. A configuration management system may be needed to manage versions and valid combinations of components.

Of course, the above scenario is pessimistic and, if we have access to the component's source code, we can reduce the number of versions by re-designing component X. Such refinements should be routinely done on the evolving

architectures, but again we have to face the problem of implementing into components lots of optional functionality and having flags to activate or deactivate the options. These components, growing in size and complexity, have to be included into any system built based on the architecture, independently of whether the options are need or not. In long-term, accumulative result of this practice is likely to be prohibitive to effective reuse.

5 A generative approach to development of component-based systems

In this section, we shall describe how we can prevent components from growing in size and number. We shall consider a program construction environment with a generator that produces component-based custom systems from a generic architecture. Further motivation for our construction environment stems from the following argument: Suppose some variant requirement R affects five runtime components. These five components may be loosely coupled during runtime. However from the program construction and maintenance perspective, these five components are tightly coupled by means of the same source of change that affects all of them. To facilitate the change, it is important that a construction environment reflects tight coupling between these components. Therefore, required properties of a generic architectures and components of a construction environment are often different from those of runtime architectures and components.

For clarity, we shall call elements of the runtime architecture components, while the design and source code elements of a generic architecture in a construction environment – construction units.

5.1 Guidelines for a construction environment for evolving component-based systems

Based on the previously discussed problems, we propose the following guidelines for a construction environment to support customization and evolution of component-based systems:

1. *An explicit domain model and architecture constraints.* The domain model should clarify common and variant requirements to be supported. Architectural constraints referring to the component structure of runtime systems (as well as other runtime concerns) should be also described.
2. *Levels of customization.* Customization of a generic architecture to meet variant requirements should be done at the architecture level first (for example, by selecting architecture construction units affected by variants, modifying interfaces of relevant runtime components, etc.) and at the code level next. A variation at the architecture level may be equivalent to many variations at the code level. Therefore, levels of abstraction simplify the mapping between variant requirements and a generic architecture.
3. *Composition of construction units of a generic architecture.* Like in aspect-oriented programming [16], frame technology [2] or Boca [10], we should be able

to define different aspects of a system (e.g., business logic, platform-specific logic for event handling, optimization techniques, etc.) in separate construction units. Also, processing elements relevant to different sources of changes (e.g., variant requirements) should be defined separately from each other. A composition operation should be provided to produce custom runtime components that have required properties from relevant construction units. The actual form of a composition operation will depend on the nature of construction units.

4. *Focusing on customization process* rather than on customized components. We need specify customizations separately from affected construction units and components. Composition operation should be automated so that routine customizations can be performed automatically. This requirement calls for using generative techniques. A generator will read specifications of required variants, customize affected components and assemble them into a custom system. This arrangement will allow us to keep the number of construction units and components of a runtime architecture under control. Customization process can be repeated whenever required. Also, the knowledge of how variant requirements affect the architecture will not be lost. By studying the customization process for anticipated variant requirements, developers will better understand how to evolve a generic architecture by implementing new unexpected requirements into it.
5. *Human-oriented construction environment*. Customization and evolution of a generic architecture and its construction units cannot be fully automated. A generic architecture should be understandable to developers. Understanding the impact of variant requirements on the architecture is particularly important.

In selecting a technology for a construction environment, we kept in mind that a generator should be able to cope with both anticipated and unexpected changes. There is a wide spectrum of generative techniques but independently of the specific approach, a generator is always implemented on top of a generic architecture. Some generators (for example, frame processor [2]) make their generic architectures open to developers, while others (for example, compiler-compilers) allow developers to manipulate the architecture indirectly through a meta-language. Revealing an underlying architecture through a specification window is elegant but may limit the ways we can deal with new unexpected changes. This may impede architecture evolution. Therefore, we found an open architecture generator more appropriate for the problem we are dealing with in this paper.

5.2 A construction environment for component-based systems

Fig. 2 outlines the logical structure and main elements of a solution that complies to the guidelines listed in the last section. We shall first describe our solution in general terms and then explain how we implemented it using frame technology [2]. A construction environment includes a generic architecture and two global structures, namely, a domain model and a Customization Decision Tree (CDT). A generic architecture is a hierarchy of construction units from which custom components of a runtime system are produced. The double arrow in Fig. 2 represents processing that is required to produce runtime system components from customized construction units of a generic architecture. This includes selection and customization of construction units, generation of custom components, assembling components into a custom system and testing.

The role of a domain model and CDT is to facilitate understanding of variations in a domain, understanding of how variations are reflected in a generic architecture and what customizations are required to incorporate variations into components.

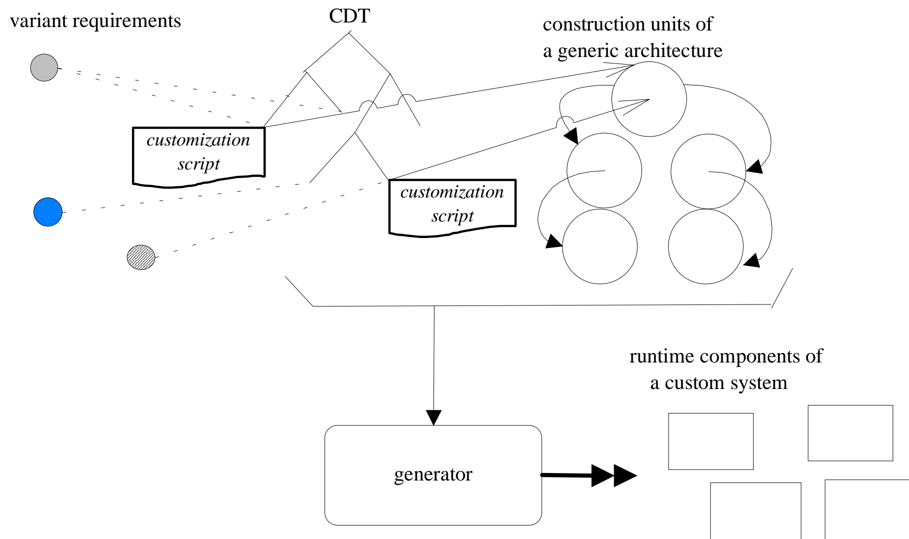

Fig. 2. A construction environment for component-based systems

The reader may refer to [8] for a detailed description techniques we used to model the variant requirements in the FRS domain. A Customization Decision Tree (CDT) helps understand customizations that lead to satisfying variant requirements. A "customization option" refers to a particular way to customize the generic architecture. A CDT is a hierarchical organization of customization options. At each level of a CDT, there may be either conceptual groupings of customization options or customization options themselves. The hierarchical organization facilitates the visual location of a particular customization option. Dotted lines in Fig. 2 link variant requirements to a corresponding customization option. For each customization option in a CDT, there is a *customization script* that specifies a chain of modifications to the construction units of a generic architecture in order to meet the corresponding variant requirement. Scripts are in machine-readable form and are executed by the generator. Therefore, customizations to accommodate anticipated variant requirements are automated, can be repeated on demand, but still may be subject of human analysis and modification. This is important when developers need extend a generic architecture with a new unexpected requirement. Developers start by inspecting other similar requirements under the related option in a CDT (there is always one at some level of a CDT) to find out how they are implemented. Developers should at least obtain certain clues as to how implement a new requirement consistently with a generic architecture rationale and structure. Once a customization script for a new requirement is written, a CDT is extended to reflect a new requirement.

As indicated in Fig. 3, a CDT may describe a wide range of choices that occur at construction-time or runtime, and refer to both functional and non-functional requirements, runtime architecture for target systems, platforms, etc.

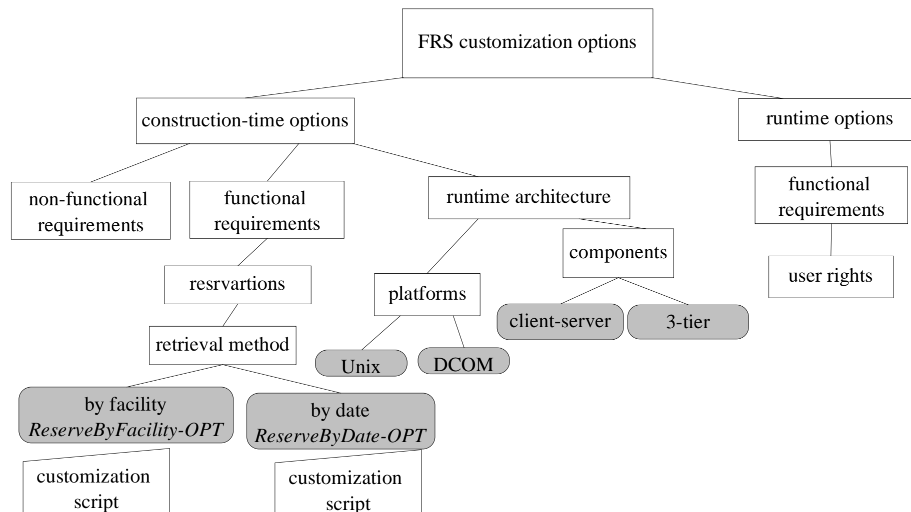

Fig. 3. Partial Customization Decision Tree with Annotations

There are many technical scenarios to realize the above concept of a construction environment. We believe Fig. 2 reflects fundamental elements that appear in any software evolution situation. Of course, these elements will appear in different, not necessarily explicit, forms, depending on a software development method and technology used. For example, in most of the companies there is no explicit domain model, links between variations and related customizations remain undocumented and we have a manual customization process instead of a generator. Most often, the emphasis is on management of already customized components. The actual representation of a generic architecture may also range from program files instrumented with conditional compilation commands to Object-Oriented frameworks and component frameworks, just to mention a few possibilities.

In our project, we used frame technology [2] (reviewed in section 2) to implement a generator-based construction environment for the FRS family. We designed a generic architecture as a hierarchy of frames. The construction units of the generic FRS architecture directly correspond to runtime components depicted in Fig. 1 (generally, this need not be the case). Construction units of the generic FRS architecture are framed Java applications and applets and framed IDL[1] specification of component interfaces. We "framed" both component code and component interfaces. A frame includes breakpoints at which a frame can be customized to accommodate variant requirements. The specification frame (the topmost circle on the right hand side of the Fig. 2) specifies all the customizations needed to incorporate required variations into a custom system. A generator (frame processor, in this case) propagates changes to the frames below. The above arrangement complies with guidelines 3 and 4: customizations are specified separately from construction units and can be executed automatically when needed. The reader can find a detailed description of frame-based architecture for FRS family in [9].

---

[1] IDL is a trademark of Object Management Group

5.3 Addressing variant requirements

As before, we shall consider reservation viewing options (variant requirement 4 of section 3). The customization proceeds as follows: We inspect the CDT (Fig. 3) in top-down manner, trying to locate a customization option that describes a variant requirement we wish to include into the FRS. We notice customization options corresponding to variant requirements retrieval *ReserveByFacility* and *ReserveByDate* under the “Retrieval Method” grouping. For each of these options, we find a customization script that specifies a chain of modifications to generic FRS architecture components that are required to address a corresponding variant requirement. We check to see if requirements retrieval *ReserveByFacility* and *ReserveByDate* are not in conflict with other variant requirements we wish to address. If not, we use the proper customization script to customize the generic FRS architecture.

A generator of a construction environment automates routine customizations. The developer is not concerned with customizations triggered by anticipated variant requirements: they are indicated in the CDT and are automatically executed by the generator. The developer does not have to remember which components and component interfaces must be modified and how to implement modifications consistently across different components. On contrary, if runtime components is all we have, the developer must be concerned with all those issues. Finally, generated runtime components of the customized FRS will contain implementation of only those variant requirements that are really required.

5.4 Evolving an architecture

With a construction environment, we implement a new requirement to view reservations by user ID in the following way. We start by inspecting the CDT (Fig. 3) to determine if a customization option for this requirement exists. We find the “Retrieval Method” grouping, but currently there is no customization option for the requirement to view reservations by user ID. To implement the new requirement, we inspect the two existing customization options under the “Retrieval Method” grouping, namely retrieval *ReserveByFacility* and *ReserveByDate*. For each of these customization options, we find (at the CDT node) a script that describes a chain of modifications required to implement a corresponding requirement. We study these scripts to understand which construction units are selected and customized. Then, we proceed to specifying modifications required to implement the new requirement. Finally, we add the customization option for new requirement *ReserveByUser* under the “Retrieval Method” grouping.

With the construction environment, evolution of the generic architecture is done in a systematic way and does not unnecessarily complicate the structure of a generic architecture. Rather than an end product of customization, we record the customization process itself, within customization scripts organized around the CDT. This record shows the trace of modifications triggered by a given requirement and reveals the rationale for modifications. The architecture becomes a library of organized and documented variant requirements together with specifications of how to include different variant requirements into the target product.

6 Conclusions

In the paper, we discussed advantages that generative techniques offer to component-based systems. In component-based systems, components tend to grow in size and number as more and more new requirements are implemented into them. We analyzed and illustrated with examples the mechanism that produces this unwanted effect. Then, we described a general scenario of how the problem can be alleviated with an aid of generative techniques. Finally, we described a construction environment, based on a commercial generator of frame technology [2], that we built to support a family of component-based Facility Reservation Systems (FRS). Our construction environment contains a generic architecture that is open to developers and a Customization Decision Tree (CDT) that guides both developers and the generator in customizing components. Nodes in the CDT represent variant requirements. Attached to the nodes are customization scripts that specify how to include required variants into components of a custom system. The CDT also helps developers evolve a generic architecture in case of new unexpected requirements. In the paper, we compared development and evolution of component-based systems in two situations: The first one reflected capabilities of DCPs (such as EJB™ or ActiveX™) and their visual environments. In the second situation, we extended a DCP with a construction environment based on a generator. We showed how our construction environment aids developers in customizing and evolving a component-based system family.

The results presented in this paper are based on experimental work that was limited in depth and breadth. In our project, we addressed variations in functional requirements. It is not clear if system-wide qualities such as performance, reliability or fault tolerance can be addressed in the same way. It would be interesting to evaluate in detail which computational aspects of a program can be most conveniently separated using techniques such as aspect-oriented programming [16], frame technology [2] or Object-Oriented techniques. Although we explicitly model requirement dependencies in our domain model [8], during customization of a generic architecture, we deal with dependent requirements in ad hoc way. This raises the issue of how we can avoid (reconcile?) conflicts during customization and how we can assure the correctness of the customized product. Currently, transition from a domain model to generic and runtime architectures is also rather ad hoc. More experiments are needed to formulate guidelines to help developers in this difficult task. We plan to address the above open problems in future work.

We believe the domain engineering process itself is not well understood yet. In our FRS project, having described an FRS domain, we came up with a component-based runtime architecture for the FRS family and then we developed a generic FRS architecture and a CDT. We believe that a systematic methodological framework is required to coordinate different activities involved in domain engineering. PuLSE [4] proposes such framework based on experiences from many domain engineering projects. We are exploring the approach described in this paper as one of strategies within a general framework provided by PuLSE.

Each technology mentioned in this paper contributes a useful solution to a certain development problem, but none delivers a complete solution. Problem domain, software design, runtime architecture and evolution - all seem to form inter-related but different dimensions of software development problem. To effectively deal with them, we need specialized methods that fit the purpose. It is a challenge for software

engineering to make those methods compatible with each other, so that they can help developers build better software systems.

Acknowledgments

This work was supported by Singapore-Ontario Joint Research Programme SOJRP0009, funded by Singapore National Science and Technology Board and Canadian Ministry of Energy, Science and Technology. Cheong Yu Chye's experimental work on engineering an FRS family contributed much to claims formulated in this paper. We are indebted to Netron, Inc. for letting us use their product Fusion for our projects.

References

1. Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice*, Addison-Wesley, 1998
2. Bassett, P. *Framing Software Reuse - Lessons from Real World*, Yourdon Press, Prentice Hall, 1997
3. Batory, D., Lofaso, B. and Smaragdakis, Y. "JST: Tools for Implementing Domain-Specific Languages," *Proc. 5th Int. Conf. on Software Reuse*, Victoria, BC, Canada, June 1988, pp. 143-153
4. Bayer J., DeBaud, J.M., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K. and Widen, T. "PuLSE: A Methodology to Develop Software Product Lines," *Proc. Symposium on Software Reusability, SSR'99,* Los Angeles, May 1999, pp. 122-132
5. Biggerstaff, T. "The Library Problem and the Limits of Concrete Component Reuse," *Proc. 3rd Int. Conf. on Software Reuse*, Rio de Janeiro, Brazil, November 1994, pp. 102-109
6. Brown, A. and Wallnau, K. "The Current State of CBSE," *IEEE Software*, September/October, 1998, pp. 37-46
7. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley & Sons, 1996
8. Cheong, Y.C. and Jarzabek, S. "Modeling Variant User Requirements in Domain Engineering for Reuse," in *Information Modelling and Knowledge Bases*, Eds. Hannu Jaakkola, Hannu Kangassalo and Eiji Kawaguchi, printed in IOS Press, Netherlands, pp. 220-234
9. Cheong, Y.C. and Jarzabek, S. "Frame-based Method for Customizing Generic Software Architectures," *Proc. Symposium on Software Reusability, SSR'99,* Los Angeles, May 1999, pp. 103-112
10. Digre, T. "Business Component Architecture," *IEEE Software*, September/October 1998, pp. 60-69
11. Fowler, M. *Analysis Patterns: Reusable Component Models*, Addison-Wesley, 1997
12. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*, Addison-Wesley, 1994
13. Jacobson, I., Griess, M. and Jonsson, P. *Software Reuse – Architecture, Process and Organization for Business Success*, Addison Wesley Longman, 1997, pp.39-42.
14. Johnson, R. and Foote, B. "Designing Reusable Classes," *Journal of Component-Oriented Programming*, June 1988, Vol.1, No.2, pp. 22-35.
15. Kara, D. "A Flawed Analogy: Hardware is not Software," *Component Strategies*, November 1998, pp. 78-80

16. Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. "Aspect-Oriented Programming," *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Finland, Springer-Verlag LNCS 1241. June 1997.
17. Krieger, D. and Adler, R. "The Emergence of Distributed Component Platforms," *IEEE Computer*, March 1998, pp. 43-53
18. Macala R., Stuckey, L. Jr. and Gross, D. "Managing Domain-Specific, Product-Line Development," *IEEE Software*, May 1996, pp. 57-67Lewis Ted et al *Component-Oriented Application Frameworks*
19. Neighbours, J. "The Benefits of Generators for Reuse," *Proc. 4th Int. Conference on Software Reuse*, April 1996, Orlando, Florida, p. 217
20. Neighbours, J. "The Draco Approach to Constructing Software from Reusable Components," *IEEE Trans. on Software Eng.*, SE-10(5), September 1984, pp. 564-574
21. Nierstrasz, O. and Meijler, T.D. "Research Directions in Software Composition," *ACM Computing Surveys*, vol. 27, no. 2, June 1995, pp. 262-264.
22. Orfali, R and Harkey, D. *Client/Server Programming with Java and CORBA*, John Wiley and Sons, 1998, second edition
23. Parnas, D. "On the Design and Development of Program Families," *IEEE Trans. on Software Eng*., March 1976, p. 1-9
24. Parsons, J. and Wand, Y. "Using Objects for System Analysis," *Comm. of ACM*, Vol. 40, No. 12, Dec. 1997, pp. 104-110
25. San Francisco Framework, http://www.ibm.com/Java/SanFrancisco
26. Sant'Anna, M., Leite, J. and Prado, A. "A Generative Approach to Componentware," *Proc. Workshop on Component-based Software Engineering*, *ICSE'20*, April 1998, Kyoto, Japan
27. Sommerville, we. and Dean, G. "PCL: a language for modelling evolving system architectures," *Software Engineering Journal*, March 1996, pp.111-121
28. Tracz, W. Collected overview reports from the DSSA project, Technical Report, Loral Federal Systems – Owego. (1994).