Managing Variability of Self-customizable Systems through Composable Components

Ioana Şora1†, Vladimir Creţu1, Pierre Verbaeten2 and Yolande Berbers2
1 Department of Computer Science, Politehnica University of Timisoara, Romania
2 Department of Computer Science, Katholieke Universiteit Leuven, Belgium

Self-customizable systems must adapt themselves to evolving user requirements or to their changing environment. One way to address this problem is through automatic component composition, systematically (re-)building systems according to the current requirements by composing reusable components. Our work addresses requirements-driven composition of multi-flow architectures.

This article presents the central element of our automated runtime customization approach, the concept of composable components: the internal configuration of a composable component is not fixed, but is variable in the limits of its structural constraints. In this article, we introduce the mechanism of structural constraints as a way of managing the variability of customizable systems. Composition is performed in a top–down stepwise refinement manner, while recursively composing the internal structures of the composable components according to external requirements over the invariant structural constraints.

The final section of the article presents our cases of practical validation. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: requirements-driven automated runtime composition

1. INTRODUCTION

Many of today’s computer systems need to be able to adapt themselves to changing requirements of their environment. The mechanisms of this adaptation should be transparent for their users, and often it is desired to occur with as few user interventions as possible. It is the case of self-customizable systems. A self-customizable system operates in an environment that imposes changing requirements for the properties of the system. Most often the evolution of the environment cannot be predicted at the system design time, so the complete variety of environmental requirements may be unknown at design time. These changing requirements for system properties must be solved dynamically at start-up or runtime when the system must customize its properties or behavior accordingly. Two
application domains that are in our view and where such self-customizable systems are needed are as follows:

1. A ‘generic terminal’ application. Such an application is a terminal independent service platform, supporting advanced telecom and secure internet value-added services (Georganopoulos et al. 2004). An end user interacts with this terminal that hides the particularities of the terminal and of the communication link. Realizing the generic terminal implies the specification and development of a generic architecture for accessing services, supporting dynamic communication protocols. We investigated these issues as part of the PEPITA project (http://pepita.objectweb.org).
   - In order to provide uniform access to all the services, the generic terminal must intelligently customize the corresponding protocol stack. This activity must be transparent for the user and hence the decisions must be taken automatically by the generic terminal.
   - Changes in the user environment (user mobility, notifying increased data loss) during the deployment of a service can later require dynamic protocol stack updates that also have to be initiated automatically.
   - In both cases, the customization of the protocol stack addresses both the composition of a stack from different protocol layers as also fine-tuning of individual protocol layers.

2. An adaptive virtual instrumentation environment for defining and executing tasks of measuring, monitoring and control.
   - Such a virtual instrumentation environment (Groza et al. 1998) consists of several virtual instruments with their connections defining a data-flow processing circuit. An adaptive environment has to configure itself according to the current monitoring task that has to be carried out, starting from a general enumeration of the desired requirements, without detailed user participation in the complete building of the measuring circuit.
   - At a certain moment during the runtime of the monitoring application, new external conditions could, for example, induce perturbations of the acquired input signals, requesting a dynamic change in the measuring circuit by adding special filters to cope with this situation.

Our research uses automatic component composition as a means of realizing self-customizable systems. A system is built from components and its properties and behavior are determined by its compositional structure. The compositional structure is given by the set of participating components and the connectors between these. The self-customizable system will adapt to the current requirements by adjusting its compositional structure. In the context of automatic component composition, the focus is on the decisional question: what components should be deployed and what connections should be between them? This composition decision is a machine decision implemented as a computerized search.

The research issue here is to define an optimal amount of information and initial restrictions that needs to be available in order to enable correct composition decisions. The challenge comes from the need to support unanticipated customization given by the following two facts:

- The variety of environmental requirements that could occur at runtime may be unknown at design time since the evolution of the environment cannot be predicted at the system design time.
- The variety of component types that will become available later during the systems lifetime is not known at the system design time.

The evolvable requirements for the system properties and the development of new component types are the sources of unanticipated situations that must be faced by self-customizable systems. A component-oriented system that adapts to the current requirements by adjusting its compositional structure must be open to discover and integrate new component types and to create new structural configurations. Thus, the customization solutions cannot be limited to the use of a set of known-in-advance components or configurations. Solutions must be open to discover and integrate new components and configurations, in response to new types of requests or to improve existing solutions when new components become available. The problem that arises here is to balance between the desired support for unanticipated customizations and the

need for constraints that guarantee a correct composition of a system with required properties.

In response to the aforementioned issues, we propose a compositional model (Sora 2004) for self-customizable systems that copes with the need for unanticipated customizations by expressing and responding to new requirements and having the ability to integrate new component types in not beforehand fixed or known configurations. The central element of our model is the concept of hierarchically composable components: the internal configuration of a composable component is not fixed, but is variable in the limits of its structural constraints. We present in this article, our mechanism of structural constraints as a flexible way of managing the variability of runtime customizable systems. A system is customized at runtime start-up by automatically composing its structure according to the current environmental requirements and in the limits of its structural constraints. Characteristic for our composition approach is that it is domain-independent, handling composition decisions at an architectural level.

The article is organized as follows: the next section presents the basic concepts that serve as starting assumptions for our compositional model, Section 3 introduces the concept of composable components and describes our mechanism of structural constraints, Section 4 presents practical validation of our approach, Section 5 refers to related work, and the final section summarizes the conclusions.

2. BASIC CONCEPTS OF THE ARCHITECTURAL COMPOSITIONAL MODEL

This section resumes our perspective on the basic concepts of component-based software engineering, which are used in our work. A software system is viewed as a set of components that are connected by connectors (Allen and Garlan 1997). A software component is an implementation of some functionality, available under the condition of a certain contract, independently deployable and subject to composition, as defined in mainstream component bibliography (Szyopersky 1997, Bachman et al. 2000).

A component in our approach is also an architectural abstraction. Our insight is that architectural style-specific compositional models are needed. This permits generic solutions that are applicable to several application problems or domains that share this architectural style. The restriction is to build a system by assuming a certain defined architectural style. Treating component composition in the context of the software architecture is a largely spread approach (Hammer 2002, Wile 2003, Inverardi and Tivoli 2002, Kloukinas and Issarny 2000), as it makes the problem manageable and eliminates the dangers of architectural mismatch. Also, in our approach, compositional decisions are made at the architectural level, with knowledge of the architectural style, but ignoring technological details of the underlying component model, as long as this provides the infrastructural support needed for runtime assembly of components.

Each component has a set of ports as logical points of interaction with its environment. We distinguish between input ports and output ports, but, further, we consider that every input port is plug-compatible with every output port. The logic of a composition is enforced through the checking of component contracts expressed by means of properties, as will be discussed later in this article.

Our work addresses systems that share the multi-flow architectural style. A multi-flow system is a variant of the classical pipes-and-filters style (Garlan 2001), with an exclusive emphasis on the pipes (the flows). A multi-flow system is defined by a number of flows on which components are plugged one after the other. The concept of flow corresponds to a data-flow relationship between ports. A flow has parts where it is internal to a component and parts where it connects ports of different components. Types and positions of components on these flows play a secondary role in defining the system architecture. As we will present later in Section 3, such a system architecture can be fully described in terms of flows and properties.

Components may be simple or composed. A simple component is the basic unit of composition that is responsible for certain behavior, and has one input port and one output port. Composed components appear as a grouping mechanism and may have several input and output ports. The internal structure of a composed component also has to comply to the multi-flow style.

Components are described through their properties, seen as facts known about them — in a way similar to Shaw’s credentials (Shaw 1996). In our approach, a property is expressed through a name.
In our approach, component contracts are expressed as sets of provided and required properties. Each component as a whole provides a set of properties (its provides clause) and may have several requires clauses. In the case of simple components, provides clauses are associated with the component as a whole. In the case of composed components, provides clauses can also be associated with ports, reflecting from the internal structure of the component. A provides clause contains a set of properties, possibly with refining subproperties or attributes. A requires clause contains a set of properties, possibly with refining subproperties and attributes. Required properties may appear as positive or negative assertions (a certain property must be present or a property can not be present). The requires clauses may also impose ordering restrictions between the required properties. The requires clauses are always associated to particular ports of the component. This is not a limitation, but naturally emerges from the fact that a component requires a certain interaction from a specific data flow. Requirements may be associated with both types of ports, input or output ports. A requirement associated with an input port reflects the expectations that the component has regarding its incoming data. A requirement associated with an output port usually states a global system correctness requirement that comes from an incomplete functionality provided by the current component.

Figure 1 illustrates the concepts presented above and introduces the graphical notations used in this article to describe components on a simple example. Component descriptions are done formally with help of CCDL (Composable Components Description Language) (Sora et al. 2003), but, for illustrations, we prefer an informal graphical notation.

The example presents two simple components, named COMPRESSER and DECOMPRESSER represented as boxes. They each have one input port and one output port. Input and output ports are represented in figures through white and black rectangles respectively. Contractual clauses are associated to components and ports through dotted lines. In this example, the component named COMPRESSER provides property compression and requires property decompression at its output port. The compression property can be achieved through several different implementations of the COMPRESSER, using different compression algorithms (LZ, GZIP, etc). The particular compression algorithm will be seen as a subproperty of the compression property.

In our model, every input port can be connected to every output port. The meaningful compositions are determined by the criteria of correct composition, based on matched required–provided properties. The matching is done first at the level of properties’ names and after that at that of attributes and recursively subproperties. A property that is required without explicit subproperties can be matched by the corresponding provided property with any subproperties.

By default, it is sufficient that a required property finds a match in a provided property of a component that is present somewhere in the external flow connected to that port, not necessarily the

Copyright © 2004 John Wiley & Sons, Ltd.

Softw. Process Improve. Pract., 2004; 9: 000–000
Managing Variability of Self-customizable Systems

3. COMPOSABLE COMPONENTS

3.1. The Concept of Composable Components

Hierarchical relationships between components are a well-accepted way of structuring and managing complexity while providing fine-grained composition. For example, the OMG CCM specification (OMG 2003) sees component implementations either as monolithic (compiled) entities or as assemblies of other components, providing a recursive definition. A component implementation always implements a certain component interface. The same component interface can have several different implementations, thus several component assemblies can implement the same component interface. However, an implementation (also assembly) must be explicitly associated with an interface. The issue here is how can it be specified as to what kind of assemblies are acceptable to implement a specific interface? How can new assemblies be automatically generated for a given interface?

We define a composable component as a first class entity that has a well-defined own identity, but does not have a fixed internal structure. The identity of a composable component is given by its own provided properties and contractual requirements (its interface).
In order to ensure the preservation of the identity of composable components, some structural constraints must be attached as its invariants. The structural constraints have the roles of flexible guidelines for future compositions of the internal structure and are not a full configuration description. The structural constraints of a composable component determine what kind of component assemblies are acceptable to implement the internal structure of the component. We argue that component descriptions need to specify not only the elements of the component interface but also the structural constraints for the internal structure of the component. The structural constraints describe actually a composition target, a component assembly to be determined.

This article proposes a method of describing structural constraints for composable components. The structural constraints of a composable component in our definition are expressed through:

1. the set of fixed internal flows
2. relationships between flows (as continuation or connection relationships)
3. the properties that must exist on these flows
4. order relationships between properties on flows.

The structural constraints are a solution that balances the need to support unanticipated customizations of the internal structure of a composable component and the need for constraints that guarantee a correct composition so that it preserves the properties that determine the identity of the composable component. The insertion of subcomponents is permitted anywhere on the existing flows, as long as their component descriptions do not contradict existing requirements (structural constraints of the composed component or requirements of the already present components on that flow).

The structural constraints comprise the following two kinds:

- basic structural constraints
- structural context-dependent requirements for component.

Both kinds of structural constraints are expressed by means enumerated above and treated without discriminations. They appear as two different kinds because of their different origin (that establishes them). The basic structural constraints may contain items of all categories 1 to 4, while the structural context-dependent requirements may contain only items of categories 3 to 4. They will be detailed in paragraphs 3.2.1 and 3.2.2.

A composable component description must contain the external view of the component (ports, contracts) and the internal view stating the structural constraints or a structural description. Paragraph 3.2.3 illustrates the formalism used to describe structural constraints.

Section 3.3 discusses how component assemblies can be generated to be compliant with invariant structural constraints and in response to variable external customization requirements.

3.2. Structural Constraints

3.2.1. Basic Structural Constraints

The basic structural constraints describe the fixed internal flows and the minimal properties that must be assembled on particular flows for the declared provides of the composed component to emerge and virtually define a 'skeleton' of the composed component. This 'skeleton' is not a rigid structure; it fixes only the flows and establishes ordering relationships between properties that must be present on these flows (as, for example, to constrain properties \( x \) and \( y \) to be on flow1, with property \( x \) "before" property \( y \) in the direction of the flow, notation \( x \leq y \)). These constraints must be specified by the developer of the composed component.

As a simple illustrating example, we develop throughout this section the case of a composable component COMPRESSER. Such a component performs data compression by an arbitrary compression algorithm. The structural constraints for the COMPRESSER component are depicted using the informal graphical notation in Fig. 2.

The basic structural constraints depicted in the figure state that the input port is connected to the output port by an internal flow that must contain the property AlgoComp\( r \). These structural constraints permit a wide variability in the customization, according to external requirements, of the internal structure of the COMPRESSER. The only restriction is that a component providing property AlgoComp\( r \) is present on the internal flow of the COMPRESSER.

Two of the possible variants of realizing the internal configuration of a COMPRESSER are shown in Fig. 3 and Fig. 4.

The first variant (depicted in Fig. 3) deploys the component HuffmanComp as a provider of
Managing Variability of Self-customizable Systems

the AlgoCompr property. As this compression algorithm uses information about the distribution of characters occurring in the initial data, the component HuffmanComp has at its input port the requirement CharFrecv. This requirement of component HuffmanComp leads to component Analyzer being added on the flow above it.

The second variant (depicted in Fig. 4) deploys an adaptive compression method, described through property AdaptiveCompression provided by component AdaptiveComp. This component has no other own requirements.

In both variants, after establishing the internal configuration for the COMPRESSER, the generic property compression will get specific subproperties from the components that have been deployed inside the COMPRESSER. As mentioned in an earlier section, these subproperties will get to the pair requirement decompression. Thus, if the first variant has been chosen for the COMPRESSER, subproperty Huffmann refines property compression and its pair requirement decompression. The composition of a DECOMPRESSER component will be done, in these circumstances, according to the basic structural constraints of DECOMPRESSER and the additional requirement Huffmann put at its input port, following a process of requirements-driven composition, as described in Section 3.3.

As this simple example shows it, an important strength of our approach is that by defining...
structural constraints in the above-described way, the customization of composed components is not limited to filling in a given skeleton with right implementations. In our example, the internal configuration of the composable COMPRESSER component is not limited to a fixed structure skeleton: variant 1 deploys two components, while variant 1 deploys one component, and more different structural configurations are possible. It is possible that new components, which can provide further enhancements or customizations for the composed component, are discovered. The insertion of these new components is permitted anywhere on the existing flows, as long as their component descriptions do not contradict existing requirements (structural constraints of the composed component or requirements of the already present components on that flow).

3.2.2. The Structural Context–dependent Requirements

The structural context–dependent requirements express requirements related to other components when deployed here as subcomponents. The basic structural constraints of a composed component allow new subcomponents to be added, as long as their properties are required and are not in contradiction with the existing constraints. Sometimes, these new components have properties that interact with other properties present in the skeleton. The relationships that must be expressed are in terms of assignment to flows and ordering relations with other properties. These interactions cannot be captured in the basic structural constraints because the developer of the composed component is not aware of the existence or possible use of the new subcomponents in its context. These structural context–dependent requirements will be added by the developer of these subcomponents. The presence of these requirements in the description of the composed component does not introduce mandatory requirements for having these properties provided here, but specifies the terms under which a certain subcomponent may be deployed here, if considered necessary. Structural context–dependent requirements do not mean that a certain property has to be provided in the structure of the composed component, but if this property is requested there by external reasons, these structural context–dependent requirements specify how and where it is appropriate to place that property.

Structural context–dependent requirements offer the possibility to update the structural constraints of a composable component. In the case where new components are defined and implemented, there might appear situations in which the existing requirements (own requirements of component and structural constraints of composed component) are not enough to exclude inaccurate compositions (are not able to prevent the new component to be placed in inappropriate places inside a composable component). In this case, the provider of the new component will have to specify a set of structural context–dependent requirements to be added to the structural constraints of the composed components in which this new one could be deployed. Below, we discuss an example where this situation occurs.

In the case of the COMPRESSER component, an external requirement could solicit the additional feature of measuring the compression rate by comparing the size of the initial with the compressed data, corresponding to a CompareSize property. We assume that the component repository contains component CS that provides property CompareSize, requiring property Size at its input port. A component S provides property Size. Applying the external requirement CompareSize over the basic structural constraints of the COMPRESSER

![Diagram](Image)

Figure 5. Counter example: incorrect variant of internal structure
Managing Variability of Self-customizable Systems

component, a semantic incorrect configuration like that depicted in Fig. 5 can result.

The own requirements of component CS will place component S above it on the flow. The issue here is that only the basic structural constraints and the own requirements of the involved components are not sufficient information in order to eliminate semantically incorrect compositions. In consequence, a configuration like that of Fig. 5 could result. In order to eliminate such erroneous configurations, additional information is needed. This information will be given by the structural context-dependent requirements.

In our running example regarding the composable COMPRESSER, the designer of component CS will have to add to the structural constraints of COMPRESSER the following context-dependent requirements, as depicted in Fig. 6. These context-dependent constraints state that, in case that a CompareSize property will be present on the internal flow of COMPRESSER, it must be after the property AlgoCompr and the property Size must be before property AlgoCompr. With these additional constraints, a correct configuration using CS inside the COMPRESSER is depicted in Fig. 7.

3.2.3. Specification of Structural Constraints

The structural constraints are part of the component description. A composable component description must contain the external view of the component (ports, contracts) and the internal view stating the structural constraints or a structural description.

The external view description of a component can be seen as an interface description. When the internal view is given as a full structural description, this is similar to an architectural description. Interface Description Languages and Architectural Description Languages can handle such specifications.

The issue is that when the internal view consists of structural constraints, these cannot be expressed using languages from these two families. Describing the structure of (hierarchical) component assemblies in terms of component instances and connections between their ports is a common feature of ADLs. The difficulty that arises here is to generally describe structural constraints that will serve as guidelines in the generation of component assemblies with certain assembly properties. In order to fill this gap, we prototyped CCDL, a description language for composable components. This language is detailed in (Sora et al. 2003).

We give here as an example the CCDL description of the COMPRESSER component with its structural constraints:

The strength of CCDL resides in its ability to specify the structural constraints for the component internals. The component Internals part of the

---

Copyright © 2004 John Wiley & Sons, Ltd.
I. Šora et al.

Research Section

1. description is relevant in the context of the current section. This part differs essentially from an architectural description: while an ADL describes the structure of a component assembly, the structural constraints specify only flexible guidelines for possible structures. In the example in discussion, the structural constraints state that the composable component COMPRESSER contains one internal flow from port in to port out and that a property AlgoCompr must be contained on this flow. Any component assembly that contains a component-providing property AlgoCompr will match the basic structural constraints of the COMPRESSER.

3.3. Requirements-driven Composition

The internal structure of a composable component will be established at runtime through automatic requirements-driven composition. The requirements for the composable target result from its invariant structural constraints and from the current requirements imposed by the external environment. For example, the DECOMPRESSER composable component mentioned in the example from paragraph 3.2.1 will be composed according to the requirements resulting from its structural constraints (which state that it has one internal flow containing property AlgoDecompr) and from the current requirements imposed by its external environment (which are the Huffmann property imposed by the already composed COMPRESSER).

The criterion for a correct composition is matching all required properties with provided properties and complying to imposed ordering relationships on every flow in the system. This criterion is used as well for validating a composition as for generating
Managing Variability of Self-customizable Systems

The right composition of a system from a set of given desired properties.

We have the mechanism of propagation of requirements as an essential element of our requirements-driven composition strategy. This mechanism of propagation works according to the principle of ‘ask someone else to solve something that you cannot solve yourself’. In a composition where a simple component B is connected to an output port of component A, while not providing matches for all requirements associated with that output port of A, these unmatched requirements are added (virtually propagated) to the output port of B. It becomes the responsibility of B to find a connection that provides matches for all these requirements. A similar propagation occurs with requirements associated on in-ports. In the case of composed components (with multiple input and output ports), the propagation of requirements follows only the internal flows originating in the connecting port. It is natural to limit propagation along internal flows as these determine which output ports are really affected by one particular input port.

The overall process of generating the structure of the target is driven by the requirements. The required properties for the target are put on the main flow of the target and propagated from that point on, while adding components. The addition of new components on the flow occurs according to the current requirements, which are those propagated from the initial requirements together with those of the new introduced components. A component is added to the solution if it matches at least a subset of the current requirements.

The mechanism of propagation of requirements used in our approach is a generalization rooted in Perry’s mechanism of propagation introduced in (Perry 1989). Perry defined a semantic interconnection model based on preconditions, postconditions and obligations, for the verification of program semantics at the level of procedural programming. Our approach brings two important contributions. First, we generalize the principle of propagation to multi-flow structures also adapting it in the context of components. Second, we use propagation as the driving force for composition (generation of the structure of the target) rather than verification of a given composition as we know related works. (Batory and Geraci 1997) and (Batory et al., 2000) use a similar propagation model for the verification of component compositions in GenVoca architectures (layered systems).

A composition step deals with composed components as units. After a composition step has determined that it wants a certain component in place, a new composition step may be launched for composing the internal structure of that component. The composition will result through top-down stepwise refinements. Such recursive compositions occur specially when a required property has refining subproperties (a requirement like p1 with refining properties (p11 and p12)). In this case, a composable component found to provide p1 will have to be fine-tuned, so that its internal structure is compliant to the set of properties (p11, p12).

A solution is considered complete when the current requirements set becomes empty. It is possible that for a certain set of requirements no solution can be found.

The mechanism of propagation of requirements briefly resumed here was formally described in (Soro et al. 2004), an article that also gives a complete description of the automatic composition strategy.

Two challenges of unanticipated customization were identified in the introductory section as the variety of environmental requirements and the variety of available component types. Our composition approach permits such unanticipated customizations. The composition strategy treats in the same way any requirement, indifferent to the set of properties or ordering relationships included in the requirement. New properties can be given as requirements at any time, as long as in the component repository there are components described to provide a match of these properties.

This comes from the fact that the composition strategy is driven by the propagation of requirements rather than on the basis of some domain-specific configuration knowledge. Also, our approach can easily discover and use new components. This comes from the fact that it searches for properties rather than component types. The mechanism of structural constraints, as defined in the previous section, permits significant variations (as number and types of deployed components) in the structure of a composable target.
4. PRACTICAL VALIDATION

This section presents applications that use our approach of structural constraints as the way of expressing invariants for composition targets. Automatic composition is used as a means to realize adaptive systems that dynamically customize themselves at runtime. In such systems, the composition decision is implemented in a Composer tool.

Section 4.1 describes our Composer tool and Section 4.2 details an automatic requirements-driven composition example from the domain of network protocols.

4.1. Architectural Composer

A Composer tool that implements the automatic composition decision for multi-flow architectures of composable components was built. Given a set of requirements describing the properties of the desired system, and a component repository that contains descriptions of available components, the Composer has to find a set of components and their configuration to realize the desired system.

The compositional decision-making system (the Composer) builds and operates on an architectural model (Oreizy et al. 1999) of the system. This architectural model is a structure description of the composed system. The Composer finds the structure of the target system starting from the imposed requirements. The Composer is architecture style–specific, the composition decisions implemented by the Composer do not contain application-specific code. The Composer determines and maintains the structure description of the composed system, while a Builder uses this structure description to build or maintain the executable system. The Builder depends upon (or is part of) the underlying component technology and framework. This integrated approach for self-customizable systems is depicted in Figure 8.

The Composer operates with requirements stated as expressions that contain component properties. The proposed adaptation model makes sense also in dynamic systems where the customization requirements have to be extracted from their changing context. Through monitoring of the context, the customization requirements can be collected and translated into required properties. The Composer works the same with the required properties, no matter where they originate from. The Composer has access to a repository containing CCDL descriptions of available components. The target of the composition is also a composable component defined by structural constraints. The composition will result through stepwise refinements: after a composition process has determined that it wants a certain component type in place, and this is a composable one, a new composition search may be launched for composing the internal structure of it. The Composer implements the requirements-driven composition strategy mentioned above in Section 3.3.

Initially, the Composer was developed and used in the context of self-customizable network protocols (http://pepita.objectweb.org). A composition decision example from this domain is given in the next Section. Later, we experimented with this method to make an virtual instrumentation environment for measurements and control (Groza et al. 1998) more self-adaptive. As our experiences with the two above-mentioned application domains confirmed, the strategy used for composition is not dependent on the application domain. There are

Figure 8. Self-customizable systems: Translator-Composer-Builder

Copyright © 2004 John Wiley & Sons, Ltd.
Softw. Process Improve. Pract., 2004; 9: 000–000
composition policies that apply generally to systems that are of the same architectural style, but do not interfere with the application domain. Reuse of composition policies occurs not by domain, but by specific architectural style. Such an architectural approach of composition has advantages as well as drawbacks that must be balanced: on the one hand, we want to use the same composition strategy for a whole family of composition problems sharing the same architectural style, on the other hand, end users should not be confronted with the problem of stating their requirements in a form that matches the underlying architecture style formalism.

The notion of requirements, as used in the context of the composition strategy implemented in the Composer, refer to properties (functional or semantic) that the composition target will have. It is clear that in case of a direct interaction with the end-user, the requests should use more meaningful concepts from the application domain so that they are not confronted with a domain that is different from their familiar application domain. The deployment of translation layers may be in the form of domain-specific front-end tools that accept client requirements expressed in a description language with a higher, domain-specific abstraction level and translate them in the terms of the domain-independent description language. Without such a tool, the end user who is also the application developer must make a mapping between the end-user-understandable configuration settings and the more technical configuration settings that implement requirements on the component description level. Deploying a translation layer enables the end user to express requirements on a higher, more abstract level and also depending on the user expertise. It may be useful to enable the end user different degrees of specificity according to his technical expertise with respect to the application domain. In this present research, we did not investigate further this aspect of domain-specific translation front-ends.

4.2. Self-customizable Network Protocol Stacks

Much research has explored the composition of network services, as, for example, well-known projects like the x-kernel (Hutchinson and Peterson 1990, Abbott and Peterson 1993, O’Malley and Peterson 1992), Horus (van Renesse et al. 1995), Ensemble (Liu et al. 1999). Many of these provide the infrastructure for stacking protocol layers and components on top of each other in a dynamic mode at runtime, using component-based approaches of various granularity in order to build flexible communication systems. Configurations may be checked against specifications to see if a given stack provides a set of required properties (Liu et al. 1999, van Renesse et al. 1995). General methods for checking design rules of such systems are extracted in (Batory and O’Malley 1992).

However, in the case of a self-customizable system, the automation must go beyond verification of a given component assembly: an appropriate component assembly must be automatically generated starting from the specification of its desired properties, the composition decision must be an automatic decision. As presented in the motivation contained in the introductory section, there are situations where self-customizable network protocols are needed.

Our solution for self-customizable network protocols is to integrate the Composer described in Section 4.1 into a component framework that is able to provide the infrastructure for dynamic protocol stacks. We have deployed DiPS, the Distrinet Protocol Stack framework (Matthijs 1999), as such infrastructure. DiPS ensures the runtime support for dynamic protocol stack changes and provides the infrastructure support for the runtime composition of components.

A whole protocol stack can be described as a composable component STACK. The structural constraints of the composable STACK define two flows, a downgoing and upgoing path, require that a network interface (corresponding to property netwint) is present at the bottom of the stack. These structural constraints are depicted in Figure 9. A property netwint must be present on both flows, with ordering restrictions that require any other property to be provided only over it. The actual structure of the protocol stack will be determined according to external requirements and respecting the structural constraints of the stack.

At a certain moment, let us consider that an application needs a reliable communication link for multimedia transmissions. This translates into the global required properties rel, transp, non-local. Since a particular kind of reliability was required, property rel is refined by subprop-100 erty multimediarel. Through propagation of requirements, the composition of the stack could
result in two solutions: TCP on IP on ETH or REL on UDP on IP on ETH, both combinations providing reliable transport. Most of the components used in this example implement the well-known protocols, REL is a custom reliability protocol. In a next step, the reliability property has to be fine-tuned for multimedia transmissions. This fine-tuning is not possible when composing only from monolithic coarse-grained components, as the TCP component. The TCP reliability retransmission strategy does not match multimediarel, thus the composition TCP on IP on ETH will be rejected. The REL component will be composed according to the requirement multimediarel applied over its structural constraints. The starting steps for composing a stack from requirements are presented in Figure 10.

The REL component is a composable component, it has a set of structural constraints derived from its basic functionality. The basic functionality that contributes to all reliability protocols is quite simple: in order to recover from data loss, the sending part will resend the data until an acknowledgement from the receiver has arrived. It has two flows, corresponding to the downgoing and upgoing paths through the protocol stack. The basic structural constraints thus state that on the downgoing flow a retransmission strategy has to be provided (property RetransmStrategy), followed by a header construction (property HeaderConstructing). On the upgoing flow, there has to
be a header parsing (property HeaderParsing), a dispatching element that routes differently data and feedback, creating a flow ramification, and, on these two flows, there has to be an acknowledgement receiving (property ACKReceiving) and an acknowledgement sending respectively (property ACKSending). Between the two flows, the downgoing and upgoing flow, there is a 'continuation' relationship. A graphical representation of these basic structural constraints of the composable component REL is depicted in Figure 11. The internal flows as well as the properties that must be present on these flows can be identified in this figure. A configuration for the REL component complying with the multimediaRel requirement is given in Figure 12. The multimediaRel requirement is forwarded to the downgoing flow of the component, leading to the selection of the MultimediaRelStrategy component for providing the right retransmission strategy (it provides properties RetransmStrategy and multimediaRel). The component MultimediaRelStrategy requires further support for readjustment of the retransmission timeout (requires property trip-time at its output port) – this leads to inclusion of a RoundTripTimeCalculator, placed, according to its own and structural requirements, on the upgoing flow. The RoundTripTimeCalculator needs time stamps to be attached on its incoming flow – so a TimeStampAttacher component is placed on the downgoing flow after the retransmission strategy. Acknowledgement sending and receiving has to be handled, according to the skeleton of the composed component. Since no preference for the acknowledgement strategy exists, positive acknowledgements are chosen (the AckReceivingUnit and AckSendingUnit components). AckSendingUnit is a composable component that has to be composed. A filter is needed, and component NextSequenceFilter will be chosen, since it is compatible with the multimedia retransmission strategy on its incoming flow.

To illustrate how our approach may handle unanticipated customizations, suppose that a new component, MultipleSending, is developed and could be used to enhance the performance of the REL layer. The requirements of this component impose that it is used on an outgoing flow of a retransmission strategy. This implies that, when multiple sending is required, such a component is deployed, as shown in Figure 12.

5. RELATED WORK

We relate to certain aspects of works to ensure the management of software variability in different fields: predictable component composition, dynamic architectures and automatic component composition, generative programming and product families.

An important research topic in component composition is the prediction of the assembly-level properties of a component composition as in (Hissam et al. 2002, Crnkovic et al. 2001). Here, most effort is directed toward prediction of ‘measurable’ properties (end-to-end latency, memory consumption), where the same property of an assembly can be calculated from the properties of the components. We consider mostly noncomputable properties in our model. The properties of a composed component in our model are usually seen as abstract features, expressed at a higher semantic abstraction level than the properties of the parts. Having the structural constraints as part of a composed component description specifies which properties put together and assembled will emerge the higher-level assembly property.
Research in the field of composition of products from a family also addresses aspects of automatic requirements-driven generation. For describing the requirements, approaches such as product lines and generative programming (Czarnecki and Eise- necker 1999, Batory et al. 2000) usually rely on a feature model, meaning that the features of the desired system are organized in different kinds of feature diagrams, containing hierarchies of feature trees with mandatory, optional and alternative features. Feature modeling introduces composition rules to specify how features may be combined to build correct products. More similarities with our approach, based on structural constraints, presents the work of (de Bruin and van Vliet 2003). They present an approach for the top–down composition of software architectures. It is based on a feature-solution graph that links requirements to design solutions.

All known approaches of product lines base their configuration decisions on domain or product knowledge expressed directly, even if with different means. This works well for product lines, where decisions are made statically in order to synthesize a product. Product lines are meant to solve variability at a pre-delivery moment (van Gurp et al. 2001). We work in the field of runtime customization that occurs post-delivery at start-up or runtime at the customers’ side, and other decisional strategies, as well as support from the runtime environment, are
Managing Variability of Self-customizable Systems

components from a component repository. We con-

For runtime compositions, 'blue-print-like' ap-

This approach limits the possibilities of unantici-

A strength of our approach is that it solves prob-

6. CONCLUSIONS

We address self-customization of systems through
requirements-driven automatic component compo-

Recent research on dynamic and self-organizing
software architectures investigate ways of doing con-

In (Kon and Campbell 2000, Kloukinas and
Issarny 2000, Issarny and Bidan 1996). We relate
to the automatic component composition approach
of Aster, a framework for runtime customization
of distributed systems (Issarny and Bidan 1996). It
offers tools for selecting and integrating middle-
ware components, starting from an architectural
description of the application and its nonfunctional
requirements. An essential step toward the possi-

required. The problem is with features that are not
predictable at initial design time and cannot be
included beforehand in a model and thus would
difficult to be taken into account at runtime
customization.

A strength of our approach is that it solves prob-
lens of unanticipated customizations: it permits to
easily formulate and solve new requirements, to

Copyright © 2004 John Wiley & Sons, Ltd.

 Softr. Process Improve. Pract., 2004; 9: 000–000

17
discover and use new component types with minimal user intervention and to variate the structural configuration of the customized system. Composable components and the mechanism of defining them through their structural constraints, as presented in this article, offer the necessary flexibility, while guaranteeing a predictable assembly.

REFERENCES


Managing Variability of Self-customizable Systems


QUERIES TO BE ANSWERED BY AUTHOR

IMPORTANT NOTE: Please mark your corrections and answers to these queries directly onto the proof at the relevant place. Do NOT mark your corrections on this query sheet.

Queries from the Copyeditor:

AQ1 Kindly clarify if the word ‘component’ here should be changed to ‘compressor’.
AQ2 We have modified this part of the sentence. Please clarify if this is fine.
AQ3 Please provide the place of publication for this reference.
AQ4 Please provide the place of publication for this reference.
AQ5 Please provide the place of publication for this reference.
AQ6 Please provide the place of publication for this reference.
AQ7 Please provide the place of publication for this reference.
AQ8 Please provide the volume number for this reference.
AQ9 Please provide the place of publication for this reference.
AQ10 Please provide the place of publication for this reference
AQ11 Please provide the place of publication for this reference
AQ12 Please provide the place of publication for this reference
AQ13 Please provide the place of publication for this reference
AQ14 Please provide the place of publication for this reference
AQ15 Please provide the place of publication for this reference
AQ16 Please provide the place of publication for this reference