A Description Language for Composable Components

Ioana Şora[1], Pierre Verbaeten[2], and Yolande Berbers[2]

[1] Universitatea Politehnica Timisoara, Department of Computer Science
Bd. V.Parvan 2, 1900 Timisoara, Romania
ioana@cs.utt.ro
[2] Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Leuven, Belgium
{pierre.verbaeten,yolande.berbers}@cs.kuleuven.ac.be

Abstract. In this paper[1] we present *CCDL*, our description language for *composable* components. We have introduced hierarchically composable components as means to achieve finetuned customization of component based systems. A composable component is defined by a fixed contractual specification of its external view and a set of structural constraints for its internal configuration. The internal configuration of a composable component is not fixed, but will be composed according to different requirements and has to comply with the structural constraints. This permits a high degree of unanticipated variability. Our approach is architectural style specific and addresses multiflow architectures. The goal of CCDL is to describe contractual specifications and structural constraints of composable components, as guidelines for their composition. CCDL descriptions can be used by automatic composition tools that implement requirements driven composition strategies.

1 Introduction

The growing complexity of modern software systems has lead to an emphasis on compositional approaches for system development. The advantages of a component-based approach to software engineering are twofold. Firstly, it can reduce production time through discipline and re-use, leading to more manageable systems. Secondly, such assembled systems can be easily updated by changing one or more components. These changes make it possible to respond to unanticipated future requirements or to a larger and better offer on the component market.

An important research topic is to be able to predict the properties of a component assembly, based on the properties of its components. In the case of *software composition*, predictable assembly means to find a set of components and to determine the collaborations between them so that it forms a software

[1] This research has been partially carried out in order of Alcatel Bell with financial support of the Flemish institute for the advancement of scientific-technological research in the industry (IWT SCAN # 010319)

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 22–36, 2003.

system that complies with a given set of requirements. However, composing a whole system only from its requirements is not feasible, additional constraints and guidelines are needed.

Automatic component composition can be used as a mechanism to achieve (dynamic) self-customizable systems that are able to adapt themselves to changing user requirements or to their evolving environment. The general issues of software composition apply as well in the case of automatic component composition. Also, there is a need to support unanticipated customization: solutions should not 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 support for unanticipated customizations and the need for constraints that guarantee a correct composition of a system with required properties. Such constraints are essential especially in the case of non-computable properties, that means when the properties of the assembly cannot be calculated from individual components properties. The fact that in case of automatic component composition the composition decision is a machine decision requires additional rigor and thoroughness. Automatic software composition has to be built on a systematic *compositional model* that must comprise a component description scheme and formalism, and a coordinated, well defined requirements driven composition strategy.

We developed a composable components model, together with CCDL as its description formalism, in the context of a compositional model for self-customizable systems. The internal configuration of a *composable* component, while not fixed, must comply to a fixed set of structural constraints, part of the component description. These structural constraints are only guidelines for future composition of the internal structure and not a full configuration description. The structural constraints are flexible enough to allow unanticipated compositions. The actual internal structure of a composable component is dynamically determined according to current requirements. This component description model establishes what information is needed to be known about the components in order to make composition decisions while CCDL defines a formalism that can be processed by automatic composition tools. Compositional decisions are made in knowledge of the architectural style, but ignoring some details of the underlying component technology, as long as this complies with the architectural assumptions. Working at the architectural level has the advantage that it abstracts domain-specific issues and permits implementation of generic composition strategies. We are developing a compositional model, targeted at *multi-flow architectures*. This establishes a framework for finding a component composition with desired properties, based on properties of individual components. We have built a *Composer* tool to automatize the requirements driven composition of systems.

The remainder of this paper is organized as follows: Section 2 presents the overall picture of our automatic component composition approach. Section 3

introduces the basic concepts of our architectural component model. We describe the composable component approach and CCDL in Sect. 4. Section 5 presents deployment scenarios for CCDL descriptions. In Sect. 6 we discuss our approach in the context of related work. The last section presents the concluding remarks.

2 Background

The automatic composition problem, as we address it, is the following: *given a set of requirements describing the properties of a desired system, and a component repository that contains descriptions of available components, the composition process has to find a set of components and their configuration to realize the desired system.* All components can be composable, that means that determining their internal structure is also a recursive composition problem.

The compositional decision making system (the *Composer*) operates on an architectural model [OGT+99] 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 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 Fig. 1.

The Composer operates with requirements stated as expressions that contain component properties. In the case when self-customization is used as start-up time configuration according to user requirements, an optional *Translator* front-end (i.e., in form of a Wizard) could be deployed to generate the requirements in form of properties expressions from a more domain-specific form.

The Composer has access to a repository containing CCDL descriptions of available components. The target of the composition is also a composable component and it has its structural constraints described in CCDL. The structural constraints have the role of guidelines for the composition, as they will be discussed further in Sect. 4.2

As we describe in Sect. 3.2, our model comprises simple and composable components. 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, in order to finetune its properties.

We have used the Composer to achieve self customizable network protocol stacks, as presented in [SMBV02] and [SVB02].

3 The Architectural Component Model

The compositional model proposed by us assumes that the system architecture belongs to a certain architectural style. Some details of the component model

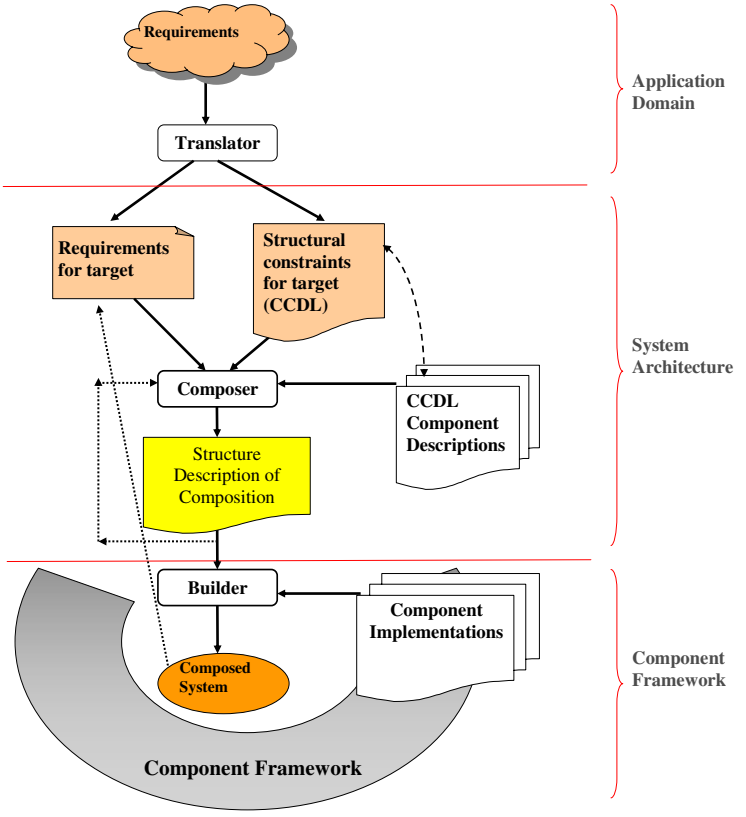

Fig. 1. Integrated approach for self-customizable systems

(like the programming interfaces) are not presented here, since they are not used in the composing phase, only later in the building phase of the system (as mentioned before in Sect. 2).

3.1 Basic Concepts

We present briefly the basic component concepts that we use and that are consistent, in the main, with the software component bibliography [BBB+00,Szy97]. We emphasize here our personal interpretations and particularities.

Software component: is an implementation of some functionality, available under the condition of a certain contract, independently deployable and subject to composition. A component in our approach is also an architectural abstraction.

Component contract: specifies the services provided by the component and the obligation of clients and environment needed by the component to provide

these services. In our approach, contracts are expressed through sets of required-provided properties.

Component property: "Something that is known and detectable about the component" [HMSW02]. In our approach, a property is expressed through a name (a label) from a *vocabulary* set. These names are treated in a semantic-unaware way [SMBV02]. In our more recent approach, values can be also associated with properties.

Port: "a logical point of interaction between the component and its environment" [AG97]. There are input ports, through which the component receives data, and output ports, through which the component generates data.

Flow: the data-flow relation among pairs of ports. A flow has parts where it is internal to a component (from an input to an output port of that component) and parts where it is between two components (a connection).

Multi-flow architecture: it is a variation of the pipes-and-filters [Gar01] architecture. An informal example is presented in Fig. 2. The particularity of this architectural style is that dataflow relations are defined first (the "flows" in our terminology) and components must fit on them. For every component the internal flows must be known so that they can be integrated in the flow architecture. In the example in Fig. 2, the system S has two flows on which it contains the subcomponents (Fig. 2a). The system S can be realized as different compositions of components on these flows, two possibilities are depicted in Fig. 2b.

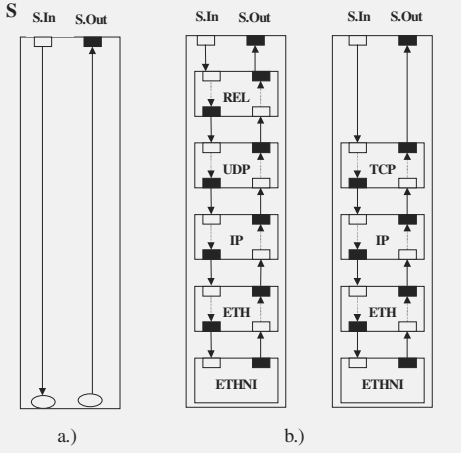

Fig. 2. Multi-flow architecture example. System S is defined by two flows and can be realized as different component compositions on these

3.2 Hierarchical Relationships between Components

Our model comprises simple and composed components. A simple component is the basic unit of composition, has one input port and one output port. A composed component appears as a grouping mechanism, may have several input and output ports. The whole system may be seen as a composed component, as S being both the system and a composed component in the example in Fig. 2. The internal structure of a composed component is aligned on a number of flows that connect its input ports with some of its output ports. For the internal structure of a composed component the same style of multi-flow architecture applies.

Composed components are "first class" components, they have their own properties and contractual interfaces and fixed internal flows. The composed component as a whole is always defined by its own set of provided properties, which expresses the higher-abstraction-level features gained through the composition of the subcomponents. The vocabulary used to describe the own provides of a composed component is distinct from the vocabulary deployed for describing the provides of its subcomponents. This abstraction definition must be done by the designer of the composed component. The properties of the subcomponent are causally linked to the properties of the composed component, but often they cannot be computed or deducted from them, as it is the case for example with the functional properties. Many properties of an assembly are emergent properties, they are related to the overall behavior of the assembly and depend on the collaboration of several components and can be seen as expressed at a higher abstraction level.

The internal structure of a composed component is mostly not fixed, these components are *composable* in the limits of certain structural constraints, as will be presented in Sect. 4.2.

3.3 Contracts Expressed through Properties

Contracts are expressed through required-provided properties. Provided properties are associated with the component as a whole. Requirements are associated with the ports. A contract for a component is respected if all its required properties have found a match. Requirements associated with an input port $C_x.In_y$ are addressed to components which have output ports connected to the flow ingoing $C_x.In_y$. Requirements associated with an output port $C_x.Out_y$ are addressing components which have inputs connected to the flow exiting $C_x.Out_y$.

In the case of composed components, provided properties can also be associated with ports, reflecting the internal structure of the component.

We assume that in the underlying component model, every input port may be connected to every output port. The meaningful compositions are determined by the criteria of correct composition, based on matching required with provided properties. A properties match is primarily defined by the match of the properties names. Properties values, if present, are used as parameters for further component configuration. By default, it is sufficient that requirements are met by some components that are present in the flow connected to that

port, these requirements are able to *propagate*. The mechanism of propagation of requirements and the basic composition strategy are presented in [SMBV02]. One can specify *immediate* requirements, which are not propagated, these apply only to the next component on that flow. *Negative* requirements specify that a property should not be present on the referred flow. *Pair* requirements refer to pairs that must be always matched in the same relative order.

As an example, in Fig. 3 is presented a simple assembly of fully matched components. The example presents a data flow part of a sender-receiver system, where encryption and compression of the transmitted data must occur and also the data transmission time must be calculated. The example system in the figure comprises seven components, and the assembly fulfills the system requirements and all component requirements are fully matched. The $TripTimeCalculator$ component calculates the time delay on a given flow. It requires that timestamps are attached to the data on its incoming flow (has the requirement *timestamp* at its input port $TripTimeCalculator.In$. This is a propagateable requirement, it can be provided by a component at any place in the incoming flow of $TripTimeCalculator$, as it is the case with component $Timestamper$ that provides the property *timestamp*. The *Encrypter* component has the requirement for *decryption* on its outgoing flow, declared as a *pair* requirement. Also the *Compresser* has a pair requirement for *decompression*. Since requirements declared as pairs must be matched in the same order as they were posed, the $Compresser - Decompresser$ sequence may either contain the $Encryptor - Decryptor$ sequence or be contained by it. A sequence like $Encryptor - Compresser - Decryptor - Decompresser$ is not permitted due to the requirements being declared as pair.

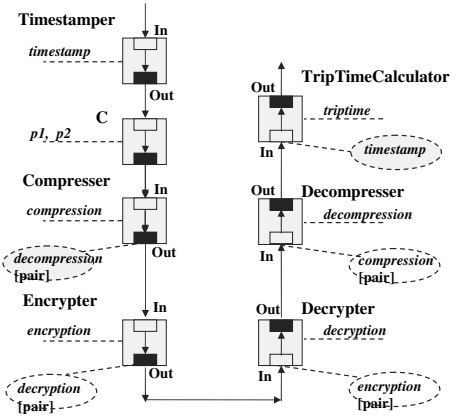

Fig. 3. Contracts expressed through required-provided *properties*

4 CCDL – The Description Language for Composable Components

We have developed *CCDL*, a description language that allows the specification of *composable* components. Current interface definition languages do not provide sufficient information about the described components for automatic component composition. Architectural description languages do not support the undefined variable internal structure of composable components. As presented in [MT00], a condition for a language to be an ADL is to be able to represent components, connectors and architectural configurations. Architectural configurations describe an architectural structure (topology) of components and connectors. The point is, in our case, the internal structure of composable components needs to be *not* represented, its configuration must remain open. In CCDL, only "structural constraints" have to be specified, leaving open the possibilities of composition. The structural constraints in our component descriptions are just flexible guidelines for future configuration compositions and not a full architecture configuration description.

We have defined *CCDL*, our component description language for composable components, as a XML Schema. The XML Schema standard is a meta-language suitable for developing new notations. This choice for XML [XML00] simplifies the implementation and later the use of the description language due to the large availability of tools for creating, editing and manipulating XML documents. We use the Apache Xerces XML parser in our implementation. Editing a CCDL description can be done with aid of syntax-directed editing tools for XML.

The CCDL description of a component comprises two main parts: the external view (contractual specification through information about global provided properties and information on the ports) and the internal view, if it is a composed component. The internal view may specify either structural constraints, if the component is composable, or a complete structural description, if the component has an already fixed internal configuration. The description of the external and internal view will be detailed in the next sections.

4.1 External View Description

The component externals contain the set of provided properties and information about all the ports. Sets of properties are used to describe the component contracts – for requirements and provides. As mentioned before, properties are in the essence names, and they are treated in a semantic-unaware way by the Composer. Properties can be further specified by values that are configuration parameters. Properties may come with a list of subproperties (introduced by the WITH tag after a property definition). The list of subproperties represents finetuning options. Subproperties are often used to finetune a requirement addressed to a composable component. I.e., for a requirement $p1\ WITH\ p11, p12$, a match is a component C that exposes the provided property $p1$, and the internal structure of C must be further composed so that it will provide the finetuning properties $p11$ and $p12$. Subproperties can also be present in the definition of a

composed component if its structure is already fixed. A composable component has usually no subproperties in its definition.

4.2 Internal View Description

The composable components do not have a fixed internal structure. In this approach lies a powerful part of the customization capability: the full internal configuration of the component will be composed as a result of external requirements allowing fine-tuning of properties [SJBV02,SVB02].

As mentioned before (in Sect. 3.2), composed components are first class entities, that have their own interfaces with ports, provided and required properties. They have also well defined internal flows. These are the fixed elements of a composed component. The internal structure is not fixed, but still some *structural constraints* must exist in order to always enforce compliance with the global component description.

The structural constraints of a composable component comprise: basic structural constraints, structural context-dependent requirements for components and inter-flow dependencies.

The *basic structural constraints* describe the minimal properties that must be assembled on particular flows for the declared provides of the composed component to emerge. These constraints virtually define a "skeleton" of the composed component. This "skeleton" is not a rigid structure, it fixes only the flows and establishes order relationships between properties that must be present on these flows. Figure 4 presents as an example the basic structural constraints for a component S, that represents a customizable network protocol stack. The basic structural constraints of S specify that it has two flows, corresponding to the downgoing (sending) and upgoing (receiving) path of packets through the stack. On these flows arbitrary protocol layer components can be added. The structural constraints specify that at the bottom of the stack the property $NetwInt$ must be provided.

The basic structural constraints must be specified by the developer of the composed component.

The *structural context-dependent requirements* express requirements related to other components when deployed here as subcomponents. If a certain component X can act as a subcomponent in the context of a composed component C, and the basic structural constraints of C are not sufficient to fully specify the correct deployment of X in the context of C, then appropriate specifications have to be added to the structural context-dependent requirements of C. These structural context-dependent requirements in C will be added by the developer of the subcomponent X. Syntactically, they are expressed also in terms of properties contained on flows and order relationships between these properties. The presence of these contained properties or order relationships as context-dependent requirements does not impose a mandatory skeleton as for the basic structural constraints. They specify the terms under which a certain subcomponent may be deployed here, but only if it considered necessary by external requirements.

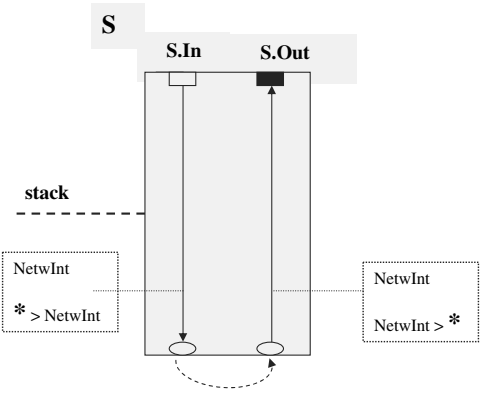

Fig. 4. Basic structural constraints example for composable component S

The *inter-flow dependencies* specify relationships between the flows. These relationships between flows can express continuation (a flow might be a logical continuation of another) or connections between flows. In the example in Fig. 4, there is a logical continuation relationship between the two flows.

The full CCDL description of S, the composable stack component, is given in Fig. 5.

A concrete stack will be built after determining its structure according to specific requirements. For example, if the current requirements are:
stack $WITH$ *rel*, *transp*, two possible component assemblies that match these requirements are these depicted in Fig. 2b.

5 Repositories

The deployment of components is supported by information from a *component repository* and an *implementation repository*. The component repository contains CCDL descriptions of components. The implementation repository contains implementation descriptions for existing component description. Not every component description must have a known implementation, the composable components usually do not have known implementations. Different ways of implementation descriptions are possible: by having specified the implementation classes or referring to the structure of a composed component. The implementation description may add also the implementation characteristics – a set of properties particular only for the implementation. This decoupling between component descriptions and implementation descriptions facilitates an easy deployment as well for using components as for introducing new component types.

The decision to use an existing component is made based on the component description. Later, an implementation must be found for that component, using the implementation repository. While the choice of a component made on hand of its properties handles the functional features of the composed system, non-functional requirements are handled by the choice of a right implementation,

```
<?xml version="1.0" encoding="UTF-8"?>
<!--CCDL for composable Stack component-->
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="E:\comp.xsd" name="Stack">
    <componentExternals>
        <provides>
            <property name="stack"/>
        </provides>
        <port name="in" type="in" entrance="true"/>
        <port name="out" type="out"  entrance="true"/>
    </componentExternals>
    <componentInternals>
        <structuralConstraints>
            <basicStructuralConstraints>
                <start name="start_up">
                    <out name="out"/>
                </start>
                <end name="end_down">
                    <in name="in"/>
                </end>
                <flow name="downgoing" from="in" to="end_down.in"/>
                <flow name="upgoing" from="start_up.out" to="out"/>
                <containedProperty name="nw_interface" flowlocation="downgoing"/>
                <containedProperty name="nw_interface" flowlocation="upgoing"/>
                <orderRelation below="nw_interface" above="any" flowlocation="downgoing"/>
                <orderRelation below="any" above="nw_interface" flowlocation="upgoing"/>
            </basicStructuralConstraints>
            <interflowDependencies>
                <continuation from="downgoing" to="upgoing"/>
            </interflowDependencies>
            <contextDependencies/>
        </structuralConstraints>
    </componentInternals>
</component>
```

Fig. 5. CCDL description

based on the implementation characteristics. If there is no known implementation, or if the implementation is not according to all the requirements, the component will be composed according to the requirements and in the limits of the structural constraints specified in the component description.

New components may be easily introduced to the system, following a scenario like depicted in Fig. 6, that presents the actions to make a new component CX known to the system. First, the new component has to be described, using for its properties terms from the established vocabulary, or extending the vocabulary when introducing new properties. The vocabulary of properties can be extended with new properties definitions. However, the extension of the vocabulary as well as deploying properties in component description should be subject to certification. Secondly, a CCDL description of the new component must be added. It may appear the need to define special interactions for the component CX when used in certain contexts. This leads to an update of the structural requirements of the components where it might be deployed. It is the task of the designer of the component CX to provide a list of updates for the use of CX in the

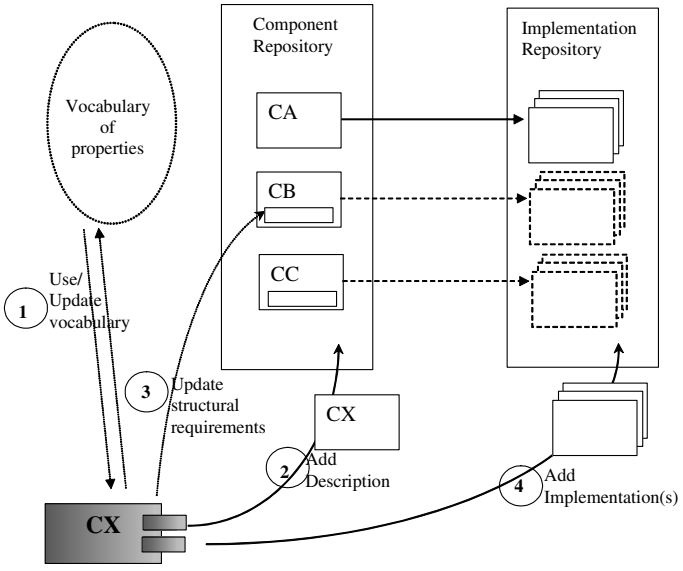

Fig. 6. Deployment of repository tools. Example of introducing a new component CX to the system

context of different compositions. For example, when using CX in the composition of CB, special restrictions might appear as to where CX should be placed in CB's internal structure. This is the case when CX provides new properties, which were unknown at the moment of CB's development. Ordering relations on the flows between these new properties and the other properties that might be involved must be specified. Final step is to provide implementation(s) for it, either in form of implementation classes or a complete description of the internal structure of a composed component. Composed components do not necessarily have implementations specified.

6 Discussion and Related Work

Our insight is that a composition model should address the architectural level, to be usable across different applications that share that architectural style. The approach is to build a system by assuming a certain defined architectural style. This approach of component composition being treated in the context of architecture is largely accepted in the research community [Ham02,IT02b, Wil01,IS01,BG97,KI00], as it makes the problem manageable and eliminates the problems of architectural mismatch [GAO95]. In this context, we present a model for composable components in multiflow architectures, together with CCDL, its description language.

CCDL is neither an interface description language nor an architectural description language, but presents some concepts related with both of them. Issues

of composition of architectural components have been addressed within ADL's (see [MT00] for an comprehensive overview). In these cases, deciding a good component combination is done statically and relies completely on the application programmer. Even within ADL's that support dynamic architectures, the dynamism is a "programmed" one. Here the goal of CCDL is different than that of ADL's. The role of ADL's is to model and describe software architectures, with their explicit configuration. The information contained in an architectural description can be used in tools to analyze properties of the architectural structure. On the other hand, CCDL does not fully describe a composed system, neither a composed component. It states only guidelines for future composition of that system or component, in form of structural constraints. The role of CCDL is to describe minimal requirements for the system configuration, leaving the configuration itself open. Tools take CCDL descriptions and generate the concrete structural configuration according to requirements. We might relate to xADL [DvdHT01], that has the capability to make conceptually distinction between architectural prescription (design-time template) and architectural description (runtime state of system). However, xADL prescriptions accept a reduced degree of variability, it can specify that certain components are optional. Nearer to our goal are [IB96] with ASTER, an interconnection language for specification of application requirements. It is used to automatically build a distributed runtime system customized to meet the requirements [KI00].

We relate also with research on predictable component assembly. An important research topic in component composition is the prediction of the assembly-level properties of a component composition [HMSW02,CBS01]. Here most effort is directed toward prediction of static properties (end-to-end latency, memory consumption [FEHC02]), where the same property of an assembly can be calculated from the properties of the components, requiring no additional information. For non-quantitative properties, approaches focus on usually one property: deadlock [IT02a], reliability [SM02]. We consider mostly non-computable 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 composable component's description specifies which properties put together and assembled will emerge the higher-level assembly property. The structural constraints are a flexible mechanism to enforce a predictable assembly of non-quantitative and non-computable properties.

7 Conclusions

In this paper we present *CCDL*, a description language for composable components in multi-flow architectures. We have introduced hierarchically composable components as a means to achieve finetuned customization of component based systems.

The goal of CCDL is to express guidelines for the component composition. CCDL descriptions can be used by automatic composition tools that implement

requirements driven compositions strategies. We have built a Composer and used it in achieving self-customizable network protocols.

A strength of our approach is that it permits a high degree of unanticipated variability, it permits to easily formulate and solve new requirements and to discover and use in given composition problems new component types, with minimal user intervention, which is very important in the case of self-customizable systems. Composable components as deployed in our model and the mechanism of defining them through their structural constraints offer the necessary flexibility, while guaranteeing a predictable assembly.

References

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[BBB+00] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Technical concepts of component-based software engineering, CMU/SEI-2000-TR-008. Technical report, Carnegie Mellon Software Engineering Institute, May 2000.

[BG97] Don Batory and Bart Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2), February 1997.

[CBS01] Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering dedicated to Component Certification and System Prediction, 2001.

[DvdHT01] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor. A highly-extensible, XML-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, Netherlands, 2001.

[FEHC02] A.V. Fioukov, E.M. Eskenazi, D.K. Hammer, and M.R.V. Chaudron. Evaluation of static properties for component-based architectures. In *Proceedings 28th EUROMICRO conference on Component-based Software Engineering*, Dortmund, Germany, September 4th–6th 2002.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, Seattle, Washington, April 1995.

[Gar01] David Garlan. Software architecture. In J. Marciniak, editor, *Wiley Encyclopedia of Software Engineering*. John Wiley & Sons, 2001.

[Ham02] Dieter K. Hammer. Component-based architecting for component-based systems. In Mehmet Askit, editor, *Software Architectures and Component Technology*. Kluwer, 2002.

[HMSW02] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau. Packaging predictable assembly. In *IFIP/ACM Working Conference on Component Deployment (CD2002)*, Berlin, Germany, June 20–21 2002.

[IB96] Valerie Issarny and Christophe Bidan. Aster: A framework for sound customization of distributed runtime systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 586–593, Hong-Kong, May 1996.

[IS01] Paola Inverardi and S. Scriboni. Connectors synthesis for deadlock-free component based architectures. In *Proceedings of the 16th ASE*, Coronado Island, California, USA, November 2001.

[IT02a] Paola Inverardi and Massimo Tivoli. Correct and automatic assembly of COTS components: an architectural approach. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, USA, May 19–20 2002.

[IT02b] Paola Inverardi and Massimo Tivoli. The role of architecture in component assembly. In *Proceedings Seventh International Workshop on Component-Oriented Programmin (WCOP) at ECOOP*, Malaga, Spain, June 2002.

[KI00] Christos Kloukinas and Valerie Issarny. Automating the composition of middleware configurations. In *Automated Software Engineering*, pages 241–244, 2000.

[MT00] N. Medvidovic and R. Taylor. A classification and composition framework for software architecture description languages. *IEEE Transactions on Software Engineering*, Vol. 26 (No. 1):70–93, January 2000.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

[SM02] Judith Stafford and John McGregor. Issues in predicting the reliability of composed components. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, USA, May 19–20 2002.

[Szy97] Clemens Szypersky. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1997.

[SJBV02] Ioana Şora, Nico Janssens, Yolande Berbers, and Pierre Verbaeten. A component composition model to support unanticipated customization of systems. In *Workshop on Unanticipated Software Evolution (USE) at ECOOP 2002*, Malaga, Spain, June 2002.

[SMBV02] Ioana Şora, Frank Matthijs, Yolande Berbers, and Pierre Verbaeten. Automatic composition of systems from components with anonymous dependencies. In *Proceedings of TOOLSEE 2001 - Technology of Object-Oriented Languages and Systems (TOOLS) East-Europe 2001*, Sofia, Bulgaria, March 2002.

[SVB02] Ioana Şora, Pierre Verbaeten, and Yolande Berbers. Using component composition for self-customizable systems. In I. Crnkovic, J. Stafford, and S. Larsson, editors, *Proceedings - Workshop On Component-Based Software Engineering: Composing Systems from Components*, pages 23–26, Lund, Sweden, 2002.

[Wil01] D.S. Wile. Ensuring general-purpose and domain-specific properties using architectural styles. In *CBSE4 Proceedings*, Toronto, Canada, May 2001.

[XML00] Extensible Markup Language (XML) 1.0 (second edition) W3C recommendation 6 october 2000, `http://www.w3.org/tr/rec-xml`, 2000.