Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

CCDL: The *C*omposable *C*omponents *D*escription *L*anguage

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

[1] Politehnica University of Timisoara, Department of Computer Science, Timisoara, Romania
[2] Katholieke Universiteit Leuven, Department of Computer Science, Leuven, Belgium

April 2004

Abstract. Tools that automate component composition decisions need as inputs formal descriptions of following categories: the functional and non-functional requirements desired for the target; the structural constraints for the target; and the contractual specifications of available individual components. In this article we present *CCDL*, a description language able to cover these three aforementioned categories.

We define a *composable* component as an architectural entity described by an external contractual specification and a set of *structural constraints* for its variable internal configuration. The internal configuration of a composable component is not fixed, but is a target that must be composed from available components. This composition is driven by external requirements while complying with the fixed structural constraints. Such hierarchically composable components permit finetuned customization of component based systems with a high degree of unanticipated variability. Our composition approach is architectural style specific and addresses multiflow architectures.

The most important strength of CCDL is its ability to describe structural constraints of composable components (that represent composition targets), as flexible guidelines for their composition. CCDL descriptions can be used by automatic composition tools that implement requirements driven compositions strategies.

1 Introduction

In the component based software engineering approach, a software system is viewed as an assembly of components. The set of components and the manner how these components are connected with each other determines the properties (functionality and behavior) of the assembled system. Constructing a system with certain required properties starts with the compositional decision, the finding of an appropriate component assembly. Current research activities investigate when and how the compositional decision process can be automated.

Due to the complexity of the fully automatic composition decision problem, it remains appropriate in circumstances like dynamic self-customizable systems where it is reasonable to impose additional constraints for the target to be composed. In our work, we use automatic component composition as a means to achieve dynamic self-customizable systems. The current behavior of such a system is determined by its structure and the components from which it is composed.

Automatic software composition decisions require a systematic *compositional model* that must comprise a component description scheme and formalism, and a coordinated, well defined requirements driven composition strategy. The description scheme establishes rules for the specification of:

– *the contractual specifications of available individual components*: What must be known about each component ?
– *the functional and non-functional requirements desired for the target*: How should required properties of the target be described?
– *the structural constraints for the target*: What additional constraints must be specified for the target ? It is common sense that composing a whole system only from its requirements is not feasible, additional constraints or guidelines for the composition are needed. But, there is also 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 ex-

isting 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.

We developed a compositional model based on *composable* components in multi-flow architectures [7], together with CCDL (The Composable Components Description Language) as its description formalism. This model establishes what information is needed to be known about individual components and the composition target in order to make composition decisions while CCDL provides a formalism that can be processed by automatic composition tools.

The remainder of this paper is organized as follows: Section 2 presents a global view of our automatic component composition approach as the context where CCDL has been developed and deployed. Section 3 introduces the basic concepts of our architectural component model. We describe the composable component approach and CCDL, the Composable Components Description Language, in Section 4. Section 5 presents deployment scenarious for CCDL descriptions. In Section 6 we discuss our approach in the context of related work. The last section summarizes the concluding remarks.

2 Background and motivation

This section presents the context where CCDL (The Composable Components Description Language) has been defined and deployed, providing a global view of our work on component-based self-customizable systems and the tools developed for this.

We approach the automatic composition of systems based on our composable components model. A central element of our approach is the concept of *composable* component: such a component has its own identity but its internal configuration is not fixed, it may vary in the limits of its *structural constraints.* The internal structure of a composable component constitutes a *target* to be composed. The structural constraints have the role of guidelines for the composition, as they will be discussed further in Section 4.2.

The automatic composition of components into a target system comprises two activities:

1. *compositional decision.* This activity is carried out by a *Composer*. The inputs of the *Composer* are the contractual descriptions of available individual components, the functional and non-functional requirements for the target and the structural constraints for the target. The Composer produces a structure description of the composed target.
2. *runtime building.* In a second step, a *Builder* uses the structure description generated by the Composer to dynamically build or maintain the target system.

This Composer-Builder separation of concerns in automatic component composition is depicted in Figure 1.

As it can be seen in the figure, both the Composer and the Builder need to access information from different sources and repositories. We developed *CCDL* as the formalism deployed for the representation of the information handled during composition. It can describe individual components, functional and non-functional requirements, structural constraints for the target and structural descriptions of assemblies. Some of the descriptions are in form of CCDL documents (the component and implementation descriptions), while others can be internal description objects that comply with the composable components model description scheme.

2.1 Composer

The *Composer* finds the structure of the target system starting from the imposed requirements, acting as the compositional decision system.

The automatic composition problem, as we address it, can be formulated: *given a set of requirements describing the properties of the desired target system, and a component repository that contains descriptions of available individual components, the composition process has to find a set of components and the way to interconnect them in order to obtain the desired target system.* All components can be hierarchically composable, thus finding their internal structure is a recursive composition problem.

The Composer determines a structural configuration that is appropriate for the target and maintains its description. This structure description is the architectural model [24] of the dynamic system.

The Composer is driven by the *requirements imposed for the target* stated as CCDL expressions. In the case when self-customization is used as start-up time configuration according to direct user requirements, an optional *Translator* front-end (i.e., in form of a Wizard) could be deployed to generate the requirements in the form of CCDL expressions from a more domain-specific form. In the case when self-customization should occur continuously at runtime, the requirements are provided by a *Monitor* module. Independently on the source of the requirements, their format at the input of the Composer is a CCDL compliant form, even if they are internal description objects and not explicit documents.

The Composer has access to a Component Repository containing *component descriptions* of available individual components. A CCDL component description specifies the external contract and, for composable components, also the internal structural constraints of the component. The Composer selects a set of appropriate components and determines the necessary connections. The *structural constraints for the target* are an important input for the Composer.

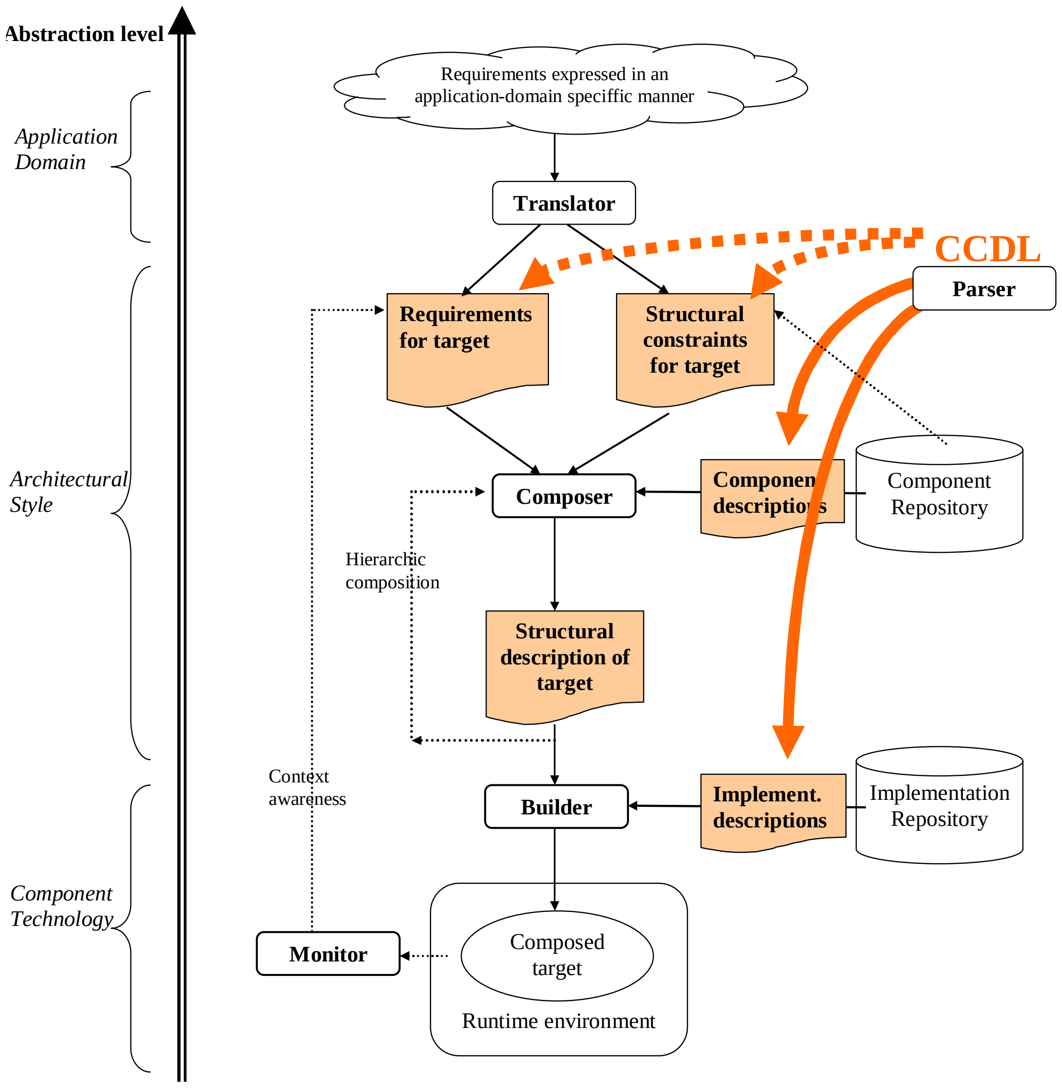

Fig. 1. Integrated approach for self-customizable systems based on automatic component composition

The composition can result through stepwise refinements: after a composition process has determined that a certain component type is needed in place, if this is a composable one, a new composition process may be launched for composing the internal structure of it.

2.2 Builder

The *Builder* depends upon (or is part of) the underlying component technology and component framework. It must have the capability to dynamically load and connect components in order to instantiate the structure description which has been determined by the Composer. The Builder takes limited decisions, as to select one of different implementations for the same component type, if multiple implementations are available for it.

The Builder has access to an Implementation Repository. An entry in this repository contains an implementation of a component type that has been described in the Component Repository. Together with the actual implementation there is also an *implementation description* that specifies the implemented component type and additional implementation specific parameters. These parameters correspond to different metrics that are used to evaluate the quality of an implementation (performance, cost, etc.).

We have used this approach to achieve self customizable network protocol stacks ([9], [10]) by integrating the Composer over the DiPS [22] framework for protocol stacks. Also we have investigated applying this Composer in self-customizable systems of multi-flow architecture from the domain of virtual instrumentation in measurement and control ([7]).

2.3 Motivation for CCDL

As we have shown in previous subsections, automatic component composition needs formal expressed input regarding the contracts of available components, the functional requirements for the target and the structural constraints of the target to be composed.

Interface description languages that describe component contracts do not provide sufficient information about the described components for automatic component composition.

Various architectural description languages [23] are used to represent components, connectors and architectural configurations. Architectural configurations describe an architectural structure (topology) of components and connectors. The point is that, in our case, the internal structure of a composition target is unknown and only structural constraints for it should be specified. Most ADLs do not support this concept of describing only the guidelines for future configurations, which is the essence of the composable components. The internal structure of composable components needs to be *not* represented, its configuration must remain open. The structural constraints in our composable components description are just flexible guidelines for future configuration compositions and not a full architecture configuration description.

The purpose of CCDL is to combine and extend features belonging to interface description languages and architectural description languages, in order to unitary describe component contracts and constraints for composition targets.

3 Basic concepts of the composable components model

The compositional model proposed in this work addresses systems of the multi-flow architectural style. Also, it introduces the concept of composable components.

Some details of the component model (like the programming interfaces) are not presented here, since they are not used in the compositional decision phase, only later in the building phase of the system (as mentioned before in Section 2).

3.1 General concepts

We present briefly the basic component concepts that we use and that are consistent, in the main, with the software component bibliography [3], [26]. We emphasize here particularities of our approach.

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 their characteristics on one side and the obligations of clients and environment on the other side. Most often the provided services and their quality depend on services offered by other parties, being subject to a contract. In our approach, contracts are expresses through sets of required-provided properties.

Component property: can describe different kinds of characteristics of a component, "something that is known and detectable about the component" [17]. Properties are most often used to describe services, similar to the service-oriented component model from [5]. In our approach, a property is first defined by its name (a label) from a domain specific standard *vocabulary* set. A property may have refining subproperties or refining parameters with values, as will be discussed in section 4.1.3. Properties may refer to functional and non-functional characteristics of the component.

Port: "a logical point of interaction between the component and its environment" [2]. There are input ports, through which the component receives data, and output ports, through which the component generates data. *Connectors* can be applied between an output port from a component and an input port from another component.

3.2 Specific concepts

Our work defines composable components in the context of multi-flow architectures.

3.2.1 Multi-flow architecture

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

Multi-flow architecture: it is a variation of the pipes-and-filters [12] architecture. An informal example of multi-flow architecture is presented in Figure 2. The particularity of this architectural style is that dataflow relations are defined first (the "flows" in our terminology) and components must fit over the fixed flows. The number and the branches of the flows define the system architecture. For every component the internal flows must be known so that it can be deployed in the flow architecture.

In the example in Figure 2, the system S has four flows on which subcomponents can be aligned. The four flows are: $In1 \rightarrow Out1$, $In1 \rightarrow Out2$, $In1 \rightarrow Out3$ and $In2 \rightarrow Out4$. The system S can be realized as different compositions of components on these flows. In the example, system S is realized as the composition of components A, B, C and D. The internal flows of all components are known and they match with the internal flows of S where they are deployed.

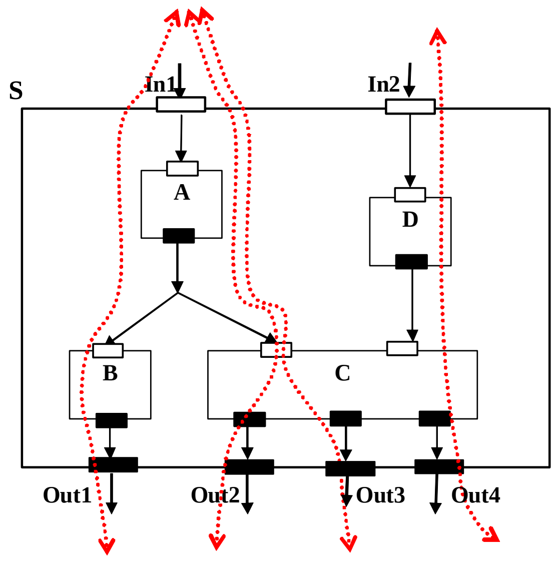

Fig. 2. Multi-flow architecture example. System S is defined by four flows and can be realized as different component compositions on these.

3.2.2 Composed and composable components

We define the basic unit of composition as simple component with one input port and one output port.

Components that have several input and output ports are considered composed components. These are built from hierarchical composed components. The whole system may be seen as a composed component, as *S* being the system and a composed component in the example in Figure 2. The internal structure of a composed component is aligned on a number of flows that connect its input ports with its output ports. For the internal structure of a composed component the same style of multi-flow architecture applies.

Composed components have a well defined identity: 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 provided services of a composed component is distinct from the vocabulary deployed for describing the provided services 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 deduced from these. 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.

For example, a *Sender* component may be composed in one instance from components *Encrypter*, *Compresser* and *Transmitter*. Each of the composing components provides a functional property: *encryption*, *compression*, and respectively *transmission*. The composed *Sender* component as a whole provides property *sending*.

As an important point of our approach, we consider that it is necessary to be able to define the identity of a composed component, even if its internal structure is not fully specified. We name such composed components as *composable* components. A composable component has a well defined external view (ports, properties). The internal structure of a composable component is not fixed, it is *composable* in the limits of certain structural constraints, as will be detailed in Section 4.2. Defining composable components is also a manner of specifying composition targets.

For example, the *Sender* component mentioned before could be realized through various other component compositions. Its external view states that it is a component with one input and one output, providing property *sending*. The internal structure of the *Sender* can be variable, with the single restriction that property *transmission* is provided inside it.

4 CCDL - The Description Language for Composable Components

We have developed *CCDL*, a description language that allows the specification of *composable* components. CCDL is able to describe the following three views of a compo-

```
<!--CCDL component description pattern -->
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="comp.xsd"
    name="SampleComponent">
    <componentExternals>
        ...
    </componentExternals>
    <componentInternals>
        ...
    </componentInternals>
</component>
```

Fig. 3. Example: Main elements of a CCDL component description

nent: the *external view*, the *internal view* if it is a composed component, and its *implementation descriptions*.

The external view contains the contractual specification of the component. It contains information about the ports and provided and required properties.

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.

An implementation description locates a binary implementation of a component and specifies additional properties of implementation metrics.

Usually the three views are not inclosed in a single document, as could be seen in Figure 1: the external view and the internal view form the component description document, stored in Component Repositories, while every implementation description may be separate documents, stored in Implementation Repositories.

CCDL, the component description language for composable components, is defined as a XML Schema [28]. The XML Schema standard is a meta-language suitable for developing new notations. This choice for XML [27] 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 used the Apache Xerces XML parser [15] in our implementation. Editing a CCDL description can be done with aid of syntax-directed editing tools for XML.

A CCDL component description has the general structure presented in Figure 3.

CCDL is defined by the XML schema `comp.xsd`. A component description corresponds to its root element `component`. The component type is identified through its `name`. The name must be unique, and is given as an attribute of the `component` element. The component is defined through the `componentExternals` and `componentInternals` elements, corresponding to the external respectively internal view descriptions.

4.1 External View Description

4.1.1 Ports

The points of interaction of a component are represented as ports. For each port, a `type` (classifying it as `input` or `output` port) must be specified. Optionally, a port can be declared as an `entrance`, that means as the point where the external requirements regarding its internal configuration are applied.

The component as a whole provides certain services, defined by global `provides` statements in the component description. In order to provide these services, it requires that other services are provided by the environment. Usually these required services must be provided to certain flows, thus the `requires` statements are attached to ports. An example is provided in the next subsection.

In the case of composed components, `provides` statements can also be associated with ports, reflecting particular services offered by composing subcomponents.

4.1.2 Contracts

We assume that in the multi-flow architecture every input port may be syntactically connected to every output port. It is the role of component contracts to enforce semantic meaningful compositions.

Contracts are expressed through required-provided properties. A contract for a component is respected if all its required properties have found a match. For a multi-flow architecture, on every flow, all properties required must be matched by properties provided by components connected to that flow.

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 $C_x.Out_y$.

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 *prop-*

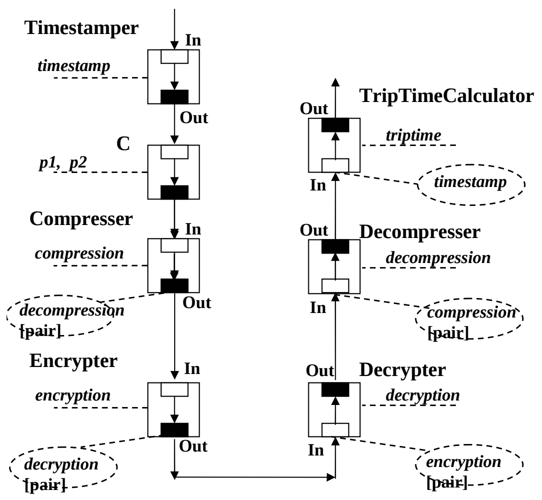

Fig. 4. Example: Contracts expressed through required-provided *properties*

agate. We have presented the mechanism of propagation of requirements and the basic composition strategy derived of it in [9].

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 Figure 4 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 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* 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.

As an example, we present the CCDL description of simple component *TripTimeCalculator* (Figure 5).

The component is identified through its name `TripTimeCalculator`. The component has two ports, one input port named `In1` and one output port named `Out1`. The service provided by this component is described by the provided property `triptime`. In order to provide this service, it requires that timestamps are attached to the data on its incoming flow (has the requirement `timestamp` at its input port `In1`).

Required properties are elements of a `requiredPropertyType` which extends the usual `propertyType` with the attributes `assertion`, `pairto` and `immediate`. The `assertion` attribute specifies if the properties is required to be present or if, on the contrary, it is required not to be present as being incompatible. The `pairto` attribute declares that the required property refers to a pair and specifies which provided property it refers to. For example, referring to the situation in Figure 4, the component *Compresser* globally provides the property *compression* and requires at its port *Out* property *decompression* as pair to *compression.* In the case that the component provides more properties, not only one global property, the name that is the value of the `pairto` attribute is essential in order to be able to distinguish between them.

4.1.3 Refining specification of properties

Properties have been presented in the previous subsection as simple names. They can be further detailed by values that are configuration parameters or by subproperties.

The list of subproperties represents finetuning options of the main property. A property element may specify as many subproperties as necessary. Each subproperty is also a property. A list of subproperties is introduced by a `with` element.

```
<!--CCDL for simple TripTimeCalculator component-->
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="comp.xsd"
    name="TripTimeCalculator">
    <componentExternals>
        <provides>
            <property name="triptime"/>
        </provides>
        <port name="in1" type="in">
            <requires>
                <property name="timestamp" assertion="yes" pairto="nil" immediate="no"/>
            </requires>
        </port>
        <port name="out1" type="out" />
    </componentExternals>
</component>
```

Fig. 5. Example: CCDL description of simple component TripTimeCalculator

For example, the property named *sending* provided by a *Sender* component may be refined through subproperties like *encryption*, *fragmentation* and *acknowledgement*.

Configuration parameters can be used to specify quantitative aspects of properties. Such parameters are introduced as `parameter` elements inside a `property` element. A parameter is specified by its name, type and value. For example, property *fragmentation* can have a parameter describing the fragmentation size, and property *acknowledgement* may be specified regarding the policy used (*ack* or *nack*).

The property *sending* with subproperties and parameters is expressed in CCDL syntax as in Figure 6.

Subproperties and configuration parameters are present in the definition of a composed component if its structure is already fixed.

A composable component usually does not specify subproperties in its definition. Subproperties are often used to express and finetune *requirements* addressed to a composable component. In example, if property $p1$ with subproperties $p11$ and $p12$ is required, 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 $p12$.

The description of a generic composable *Sender* component will thus specify provided property `sending`, without subproperties and parameters. The external view description of component *Sender* is given in Figure 7.

A client who wants to deploy a *Sender* component will be able to specify in its requirements finetuning subproperties for the *sending* property, as in Figure 6. The matching of such properties occurs level by level. First, components providing the global property *sending* are searched, then the search continues for the subproperties. If a component providing the exact subproperties is found, this can be used directly. If no component providing all subproperties is found, then the composable component providing the generic property will be composed according to the desired subproperties.

4.2 Internal View Description

Component internals can be given either as a *structural description* of the internal configuration or as *structural constraints* in the case that the internal configuration is not fixed.

A structural description will completely specify all individual component types that are deployed and all connections between their ports, in a classical style of an architectural description language. We will not detail here further this aspect.

The specification of structural constraints is an important feature of CCDL. It is needed as 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 [7].

4.2.1 Expressing structural constraints: flows, on-flow properties, inter-flow dependencies

As mentioned before (in Section 3.2.2), composable components are first class entities, that have their own interfaces with ports, provided and required properties. These are the fixed elements of a composable component. The internal structure is not fixed, but still some *structural constraints* must exist in order to always ensure the identity of the global component description.

We express structural constraints by means of fixed internal flows, properties that must be present on these flows, and inter-flow dependencies.

```
<property name="sending">
    <with>
        <property name="encryption" />
        <property name="fragmentation">
            <parameter name="fsize" type="integer" value="1024"/>
        </property>
        <property name="acknowledgement"/>
            <parameter name="kind" type="string" value="nack"/>
        </property>
    </with>
</propert
```

Fig. 6. Example: refining properties through subproperties and parameters

```
<!--CCDL for Sender component-->
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="comp.xsd"
    name="Sender">
    <componentExternals>
        <provides>
            <property name="sending"/>
        </provides>
        <port name="in1" type="in" entrance="true"/>
        <port name="out1" type="out"  entrance="true">
        <requires>
                <property name="receiving"
                          assertion="yes" pairto="sending" immediate="no"/>
            </requires>
        </port>
    </componentExternals>
    <componentInternals>
    ...
    </componentInternals>
</component>
```

Fig. 7. Example: External view of component Sender

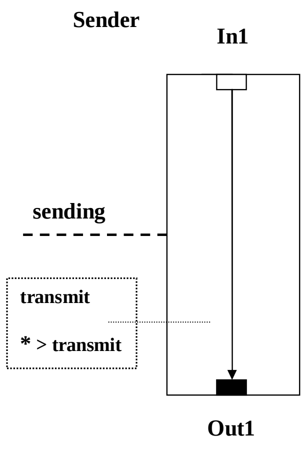

Fig. 8. Exampke: Basic structural constraints example for composable component Sender

The structural constraints describe the minimal properties that must be assembled on particular flows so that the declared provided properties of the composed component 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 8 presents as an example the basic structural constraints for the composable component *Sender*.

The basic structural constraints of *Sender* specify that it has one internal flow. On this flow arbitrary components can be added. The structural constraints specify that at the bottom of the layer the property *transmit* must be provided.

The CCDL description of the composable *Sender* component can be completed with the internal view, as in Figure 9.

Flows are introduced as `flow` elements, with attributes `name`, `from` and `to`. Attributes `from` and `to` specify the flows endpoints. These endpoints can be names of ports or names of internal reference points. The internal reference points are constructions which enable the specification of flow ramifications (`rp`), junctions of flows (`join`), flow sources(`start`) or sinks (`end`).

Contained properties are introduced with help of the `containedProperty` element. Such an element specifies that a certain property (*transmit*, in our example) must be present on a certain flow (it must be provided by a component having ports connected to that flow).

Between properties belonging to the same flow, order relations may be imposed by the `orderRelation` element. In our example, property *transmit* must be at the bottom of the *Sender* (it must be below `any` other property).

A concrete *Sender* will be built after determining its structure according to specific requirements. The requirements imposed at its port $In1$ can be, for example, these specified in Figure 6. Applying the composition strategy based on propagation of requirements as we described it in [8] an adequate structure of the *Sender* will result.

4.2.2 Updating structural constraints

The structural constraints of a composable component must be defined by the designer of that component.

It is possible that the initial specification of a composable component may be updated. The need to update the specification of a composable component C can appear when designing a special component X that is meant to be deployed here as subcomponent. Most of the time, the initial specification of C's structural constraints can handle by itself various subcomponents to be deployed. In the case that the designer of a new component X wants to make additional specifications regarding the possible deployment of it in certain contexts, this possibility must remain open. The need for such an update is however an exceptional situation.

Thus we distinguish two categories of structural constraints of composable components, according to their source: the basic structural constraints, defined by the designer of the component, and the structural context-dependent requirements.

The structural context-dependent requirements are added at the explicit request of other components. Syntactically, the structural context-dependent requirements 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 in the case of the basic structural constraints. They specify the terms under which a certain subcomponent may be deployed here, but only if it is considered necessary by external requirements.

For example, the developer of the *TripTime* and *TimeStamp* components will want to specify that these components are to be used such that property *triptime* is below any other property on the flow to be measured while property *timestamp* is above any other property present on the same flow. This will lead to an addition to the structural constraints of the *Sender* specified initially in Figure 9, as presented in Figure 10.

Adding the order relation relative to property *timestamp* as context dependent constraint does not in any case always enforce the presence of *timestamp*. Only if an external requirement imposes *timestamp*, an adequate component will be searched and the order relation rule relative to *timestamp* is activated, so that this property is placed correctly.

4.3 Implementation description

The implementation description specifies the location of an implementation for that component type. The location can be that of the implementation classes or that of a complete structure description for a composed component.

The implementation description can also contain additional implementation specific parameters. These parameters correspond to different metrics that are used to evaluate the quality of the implementation (performance, cost, etc.) and do not refer to functional properties of the component. The parameters are described by a simplified type of properties, the `implemPropertyType`. This contains a name and an optional value. For example, the implementation description of a component may define as parameters the following:

```
<provides>
    <impl_parameter name="cost" value="free">
    <impl_parameter name="memory" value="high">
 </provides>
```

```
<!--CCDL for Sender component-->
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="comp.xsd"
    name="Sender">
    <componentExternals>
        ...
    </componentExternals>
    <componentInternals>
        <structuralConstraints>
            <basicStructuralConstraints>
                <flow name="f" from="in1" to="out1"/>
                <containedProperty name="transmit"  flowlocation="f" />
                <orderRelation below="transmit" above="any" flowlocation="f" />
            </basicStructuralConstraints>
        </structuralConstraints>
    </componentInternals>
</component>
```

Fig. 9. Example: Internal view of composable component Sender

```
        <structuralConstraints>
           <basicStructuralConstraints>
              ...
           </basicStructuralConstraints>
           <contextDependencies>
                 <orderRelation below="any" above="timestamp" flowlocation="f" />
           </contextDependencies>
        </structural
```

Fig. 10. Example: Context dependent structural constraints

This means that this implementation version is freely available and it has high demands for memory.

The requirements for the target may contain also a the desired implementation parameters. Unlike the properties describing functional requirements for the target, the implementation parameters will refer globally to all components of the target system.

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, that specify the component externals and internals. 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. 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 introducing new component types.

The decision to use an existing component is made by the Composer based on the component description. Later, the Builder must find an implementation 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, the implementation characteristics are handled by the choice of a right implementation, based on the implementation parameters. 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 Figure 11, 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 domain vocabulary. This vocabulary of properties must be part of the corresponding domain specification (as for example the OMG domain standards [25].

Secondly, a CCDL description of the new component CX must be added. It could happen that special in-

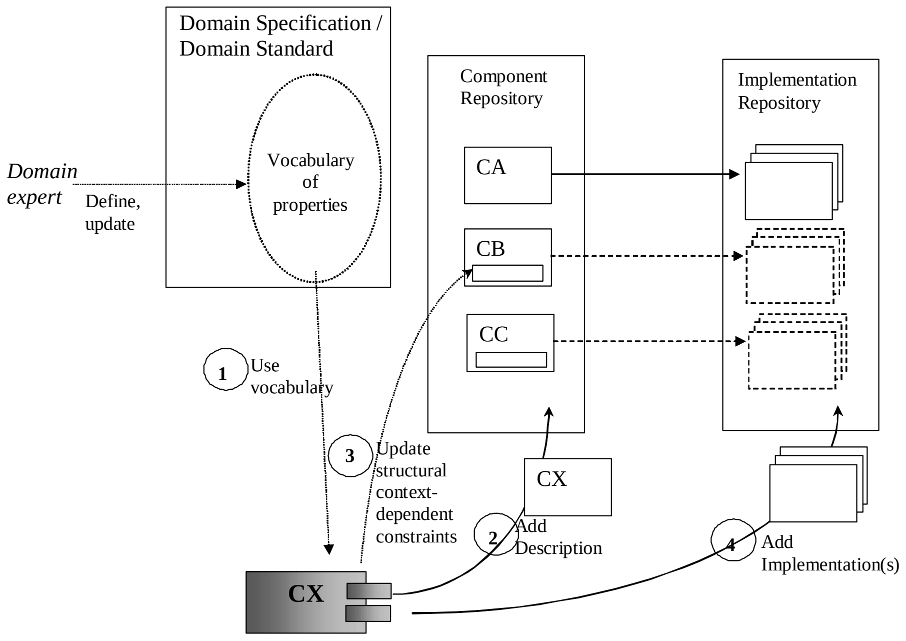

Fig. 11. Deployment of repository tools. Example of introducing a new component CX to the system

teractions for the component CX when used in certain contexts should be specified. It is the task of the designer of the component CX to provide the updates for the use of CX in these special contexts. 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. The update lists provided by CX will become CB's context-dependent structural constraints.

The final step is to provide implementation(s) for the new component CX, either in form of implementation classes or a complete description of the internal structure of a composed component. Composed components must not 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 ([16], [19], [30], [18], [4], [21]), as it makes the problem manageable and eliminates the problems of architectural mismatch [13]. In this context, we present a model for composable components in multiflow architectures, together with CCDL, its description language.

In specifying the architecture of dynamic systems, different approaches have been used, most of them in the category of architectural description languages. In another approach [29], the Chemical Abstract Machine (CHAM) formalism has been used to describe architectural styles and reconfigurations.

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 [23] 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 main particularity 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 [11], 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 is ASTER [20], an interconnection language for specification of application requirements. It is used to automatically build a distributed runtime system customized to meet the requirements [21]. Georgiadis et al [14] propose the architectural specification of a self-organising system through a set of constraints. These constraints define an architectural style and can be used to generate or verify a specific architectural instance for compliance. The composition language PeerCAT [1], intended to describe composition of peer services into new applications permits the declaration of a minimal composition. This is different from our structural constraints in the fact that an actual minimal structure is given. Also, pure service-oriented approaches as [5] do not consider that a composed application has a structure with a defined topology.

The ensemble of our work can be put also in the context of research on predictable component assembly. An important research topic in component composition is the prediction of the assembly-level properties of a component composition [17], [6]. The structural constraints as part of a composable component's description specify which properties put together and assembled will lead to the higher-level assembly property. The advantage of the structural constraints is that they are a flexible mechanism to enforce a predictable assembly of functional (non-quantitative and non-computable) properties.

7 Conclusions

Automatic component composition decisions require precise specifications of the contracts of individual components, of the functional and non-functional requirements for the target and of additional structural constraints of the target. In this article we present *CCDL*, a description language for composable components in multi-flow architectures, that is able to specify all the needed categories.

We have introduced composable components as a means to achieve finetuned customization of component based multiflow architectures. A composable component is simultaneously a building block and a composition target. CCDL distinguishes itself especially through its capability to express guidelines for the composition of composable targets.

CCDL descriptions are used by automatic composition tools that implement requirements driven compositions strategies in self-customizable systems.

References

1. Sascha Alda. Component-based self-adaptability in peer-to-peer architectures. In *Proceedings of the 26th International Conference on Software Engineering ICSE 2004*, 2004.
2. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
3. 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.
4. Don Batory and Bart Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2), February 1997.
5. Humberto Cervantes and Richard Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *Proceedings of the 26th International Conference on Software Engineering ICSE 2004*, pages 614–623, 2004.
6. Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors. *Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering dedicated to Component Certification and System Prediction*, Toronto, Canada, 14–15 May 2001.
7. Ioana Şora. *Model compozitional bazat pe componente compozabile in arhitecturi multi-flux (Compositional model based on composable components in multi-flow architectures, in Romanian)*. PhD thesis, Politehnica University Timisoara, Romania, 2004.
8. Ioana Şora, Vladimir Creţu, Pierre Verbaeten, and Yolande Berbers. Automating decisions in component composition based on propagation of requirements. In Michel Wermelinger and Tiziana Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, Proceedings*, number 2984 in Lecture Notes in Computer Science, pages 374–388. Springer Verlag, 2004.
9. Ioana Şora, Frank Matthijs, Yolande Berbers, and Pierre Verbaeten. Automatic composition of systems from components with anonymous dependencies. In Theo D'Hondt, editor, *Technology of Object-Oriented Languages, Systems and Architectures*, pages 154–169. Kluwer Academic Publishers, 2003.
10. 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 at IEEE-ECBS 2002*, pages 23–26, Lund, Sweden, 2002.
11. 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.
12. David Garlan. Software architecture. In J. Marciniak, editor, *Wiley Encyclopedia of Software Engineering.* John Wiley & Sons, 2001.
13. 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.

14. Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the ACM SIGSOFT Workshop on Self-healing Systems WOSS'02*, 2002.
15. Apache Group. Xerces Java Parser. http://www.apache.org, 2001.
16. Dieter K. Hammer. Component-based architecting for component-based systems. In Mehmet Askit, editor, *Software Architectures and Component Technology*. Kluwer, 2002.
17. 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.
18. 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.
19. 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.
20. Valrie 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.
21. Christos Kloukinas and Valerie Issarny. Automating the composition of middleware configurations. In *Automated Software Engineering*, pages 241–244, 2000.
22. Frank Matthijs. *Component Framework Technology for Protocol Stacks*. PhD thesis, Katholieke Universiteit Leuven, Belgium, December 1999.
23. 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.
24. 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.
25. The Object Management Group (OMG). Catalog of Domain Specifications. http://www.omg.org/technology/documents/domain_spec_catalog.htm
26. Clemens Szypersky. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1997.
27. World Wide Web Consortium W3C. eXtensible Markup Language (XML) 1.0. http://ww.w3.org/TR/1998/REC-xml-19980210, 1998.
28. World Wide Web Consortium W3C. Xml schema. http://ww.w3.org/XML/Schema, 2001.
29. Michel Wermelinger. Towards a chemical model for software architecture reconfiguration. In *Proc. of the 4th Intl. Conf. on Configurable Distributed Systems*. IEEE Computer Society Press, 1998.
30. David Wile. Revealing component properties through architectural styles. *Journal of Systems and Software, Special Issue on Component-Based Software Engineering*, 65(3):209–214, March 2003.