Automating Decisions in Component Composition Based on Propagation of Requirements

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

[1] University Politehnica of Timisoara, Department of Computer Science and Engineering, Bd. V. Parvan 2, 1900 Timisoara, Romania
ioana@cs.utt.ro

[2] Katholieke Universiteit Leuven, Department of Computer Science Celestijnenlaan 200A, 3001 Leuven, Belgium

Abstract. Automatic component composition is a way to achieve self-customizable systems that are able to adapt themselves through structural configuration to changing conditions in their environment. In this paper, we propose an automatic composition strategy for multi-flow architectures with hierarchically composable components. Our composition strategy takes automatic decisions for the composition of a target that is specified through a set of required properties imposed over its given structural constraints. The composition decisions are taken knowing the properties provided by individual available components. Properties characterize functional or non-functional aspects of a component. The composition strategy is driven by a mechanism of propagation of required properties, detailed in this paper.

1 Introduction

Component-based development is a proven approach to manage the complexity of software and its need for customization. An important challenge is to build new systems that provide certain properties, by systematically composing reusable components. Our research approaches the problem of component composition from the point of view of the decisional question: how to decide what components will be deployed and what collaborations will be between them?

The need for rigorous strategies for compositional decisions appears particularly in circumstances when the composition decision must be a machine decision, as it is the case when automatic component composition is used as a means to realize self-customizable systems. Our work addresses self-customizable systems that are able to adapt themselves to their evolving runtime environment. Such automatic software composition is based on a compositional model that comprises:

– A *component description scheme and formalism.* This establishes what information is needed to be known about the components in order to make composition decisions.

– A well defined *requirements driven composition strategy.* This establishes the rules for selecting the necessary set of components and determining their integration, based on the available information about the components.

Many approaches tackle composition in domain-specific ways. We argue that compositional models should be architectural style specific and independent from application domains, to create a premise for generic solutions. We have developed a compositional model for multi-flow architectures based on composable components. It comprises a component description scheme for hierarchically composable components with its description language CCDL (introduced in [ŞVB03]) and a requirements driven composition strategy, which is described in this paper.

The composition strategy implements rules for finding a component composition with desired properties, based on the properties of individual components described according to the CCDL scheme ([ŞVB03]). Supporting unanticipated compositions (in terms of deployed components and structure) is a main objective of our composition approach. This paper presents the principles of our composition strategy, introducing the mechanism of propagation of requirements as its driving element.

The remainder of this paper is organized as follows: Section 2 presents briefly our architectural component model as the premise of our work on automatic composition. Section 3 details the mechanism of propagation of requirements and Section 4 presents the composition strategy. Finally, Section 5 discusses our research in the context of related work, while the last section summarizes the conclusions.

2 Architectural Model and Composable Components

This section presents briefly the main concepts of our component model.

A software system is viewed as a set of components that are connected by connectors. A software component is an implementation of some functionality, available under the condition of a certain contract, independently deployable and subject to composition [Szy97]. Moreover, a component in our approach is also an architectural abstraction.

We consider that the system architecture reflects *interaction* relationships among the components. A component has a set of *ports* for the interaction with the rest of the system. A port is "a logical point of interaction between the component and its environment" [AG97]. Distinction is made between input and output ports. In our approach all components are considered plug-compatible in the sense that an input port can be connected to an output port.

Our current work on composition investigates systems that have a *multi-flow architecture*. The concept of *flow* corresponds to 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). We define the multi-flow architecture as a variation of the pipes-and-filters architecture, where the architecture of a system is completely defined by the dataflow relations (the "flows" in our terminology).

The flows are fixed, while the positions of components on these flows are not important. Components must just fit on the fixed flows. For every component, the internal flows must be known so that they can be integrated in such a flow architecture.

Components can be simple and composed. A simple component is the basic unit of composition, has one input port and one output port. A composed component is an aggregation of several other components, it may have several input and output ports. Multi-flow architectural style applies to the internal configuration of composed components.

A component's contract specifies the services provided by the component and the obligations of its clients and environment. In our approach, contracts are expressed through sets of required-provided *properties*. A component property is a fact that is known about the component. In our approach, a property is expressed through a name from a vocabulary set and may have attributes or refining subproperties. The name of the property is treated in a semantic-unaware way: this means that a match between required and provided properties is established by matching names and attributes. Provided properties are associated with components as a whole, requirements are associated with ports.

An important element of our approach is that composed components are also "first class" components, they have their own properties and contractual interfaces. A 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 properties 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 internal structure of a composed component is mostly not fixed, these components are *composable* in the limits of certain structural constraints. These structural constraints ensure the preservation of the identity of the composable component. As they have been introduced in [ŞVB03], the structural constraints are flexible guidelines for future compositions of the internal structure but not a full configuration description. Structural constraints of a composable component are expressed through: the set of internal flows, the properties that must exist on these flows, and possibly the order relationships between these properties. The structural constraints are a solution that balances between the need to support unanticipated customizations of the internal structure of a composable component and the need to preserve the properties that determine the identity of the composable component.

An important strength of our approach is that it does not limit the customization of composable components to filling in a given structure with right implementations. It is possible that new components, which can provide further enhancements or customizations for the composed component, are discovered. The insertion of these new subcomponents is permitted anywhere on the existing flows, as long as their component descriptions do not contradict existing requirements (structural constraints of the composed component or requirements of the components already present on that flow). The composition strategy automatically decides which subcomponents to deploy on the internal flows of the composed target, starting from the requirements imposed by the client and complying with the structural constraints of the target. This strategy is based on a mechanism of propagation of requirements that will be detailed in the next sections.

3 The Mechanism of Propagation of Requirements

The operation of matching required properties of one component with provided properties of other components can be a complex process. The complexity resides in the fact that interactions of properties cannot be isolated to pairs of interacting components, but most often there are large groups of components with transitive interaction relationships between them. In order to manage the complexity of such situations we define and use a mechanism of propagation of requirements. This is discussed here, starting in subsection 3.1 with a simplified case and developing to the general case in subsection 3.2.

3.1 The Linear Case

First we introduce the mechanism of propagation of requirements in the linear case, corresponding of a single-flow system containing a sequence of *simple* components.

In this case, each component has one input port and one output port. The requirements associated with the input port address components that are before the current one on the flow. These requirements are *upward requirements*. The requirements associated with the output port address components that are below the current component and we name them *downward requirements*. By default, it is sufficient that a required property associated with a port is provided by a component that is present somewhere on the flow connected to that port. The requirements of a component are not necessarily met by immediate neighbors of that component, but by some components situated further on the corresponding flows. One can specify immediate requirements, which apply only to the next component on that flow. Also negative requirements (a property should not be present in a flow) are possible.

Given a component C, it has, at an arbitrary moment during the composition process, a set CU of n upward requirements, $CU = \{CU_i\}_{i=1...n}$ and a set CD of m downward requirements, $CD = \{CD_i\}_{i=1...m}$. To ensure that the contract of C is fully complied, C must be part of a composition where the components placed above C fulfill all its upward requirements CU and the components below C fulfill all downward requirements CD.

The goal of the composition process is to find the two sets of components that placed above C and below C fulfill all its requirements. Of course, each of these components introduces their own requirements, that have to be also fulfilled.

The initial requirements of C can be propagated to its neighbor components, if the neighbor component does not provide them itself. This mechanism works like delegating the responsibility for these requirements to the neighbor components.

Let a component X provide the set of properties XP and have the upward requirements XU. It makes sense to connect component X above component C (make X the top neighbor of C by connecting the output port of X to the input port of C) if it provides a part of component's C current upward requirements. The subset of C's upward requirements that are fulfilled by component X is a set of properties named $XPCU$,

$$XPCU = XP \bigcap CU \tag{1}$$

If $XPCU$ is not void (that means, component X provides at least some of the upward requirements of component C), component X will be connected at the input port of C.

Most often, component X does not fulfill all the current upward requirements CU of C, such that a subset $XNPCU$ of CU remain not fulfilled:

$$XNPCU = CU - XPCU \tag{2}$$

All the properties belonging to the set $XNPCU$ are requirements that must be fulfilled by other components connected above C. These properties will be added to the set of upward requirements of component X, process that is called *upward propagation of requirements.*

Following this propagation, the new set of upward requirements of component X becomes XU':

$$XU' = XU \bigcup XNPCU \tag{3}$$

The set of properties XU' is a new set of requirements to continue the upward searching of components.

The downward propagation of requirements is defined in a similar manner. Let a component Y provide the set of properties YP and have the downward requirements YD. It makes sense to connect component Y below component C (connect the output port of C to the input port of Y) if it provides a part of component C's current downward requirements. The subset of C's downward requirements that are fulfilled by component Y is a set of properties $YPCD$,

$$YPCD = YP \bigcap CD \tag{4}$$

A subset $YNPCD$ of component C's downward requirements remain not fulfilled by component Y:

$$YNPCD = CD - YPCD \tag{5}$$

All the properties belonging to the set $YNPCD$ will be added to the set YD of downward requirements of component Y, process that is called *downward propagation of requirements.*

$$YD' = YD \bigcup YNPCD \tag{6}$$

Figure 1 depicts an example of linear propagation of requirements.

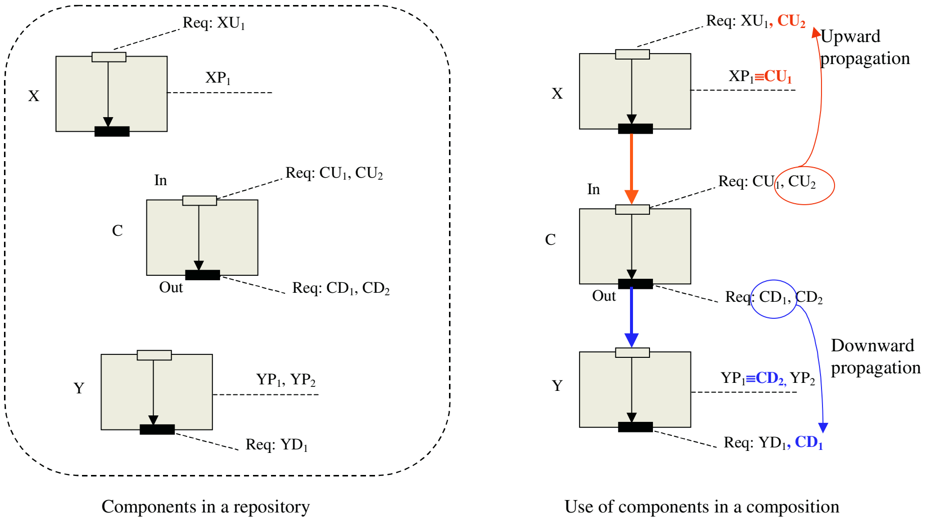

Fig. 1. Linear propagation of requirements

The example in Figure 1 involves three components: component C, component X, and component Y. Component C has the upward requirements CU, where $CU = \{CU_1, CU_2\}$ and the downward requirements CD, with $CD = \{CD_1, CD_2\}$. A composition that solves the requirements of C must be found. Let a component repository contain among others components X and Y. Component X provides $XP = \{XP_1\}$ and has own upward requirements $XU = \{XU_1\}$. Component Y provides the set of properties $YP = \{YP_1, YP_2\}$ and has own downward requirements $YD = \{YD_1\}$. We ignore at this step of the example the upward requirements of Y and the downward requirements of X.

It is given that property XP_1 matches property CU_1 and property YP_1 matches property CD_2. That means that component X fulfills one of C's upward requirements and component Y fulfills one of C's downward requirements. Component X will be connected on top of component C, making a connection $X.Out \rightarrow C.In$ and component Y will be connected below component C through a connection $C.Out \rightarrow Y.In$. After these connections, the subset $XNPCU$, $XNPCU = \{CU_2\}$, of C's upward requirements remain unfulfilled and will be propagated to the port $X.In$.

The new set of upward requirements of X is now $XU' = \{XU_1, CU_2\}$.

Similarly, the subset of C's downward requirements not fulfilled by Y is $YNPCD = \{CD_1\}$, and will be propagated from C to the port $Y.Out$. The new downward requirements of Y are now $YD' = \{YD_1, CD_1\}$. After having connected component X as the neighbor on top of C and component Y the neighbor below C, the searching for new components continues having XU' and YD' as driving requirements.

The downward requirements of component X as well as the upward requirements of component Y have been ignored until now. If C does not fulfill the downward requirements of X, then a propagation of these from X to C will also occur; the same for upward requirements of Y.

An example of using linear propagation of requirements for the automatic composition of customized network protocol stacks is our early work[1] [ŞMBV03], where a protocol stack is automatically built as a composition of protocol layer components according to client requirements.

The linear case is simple and intuitive, yet not sufficient for the building of more complex systems according to fine-tuned requirements. In the next subsection, the mechanism of propagation of requirements is generalized for components with an arbitrary number of input and output ports that are part of multiflow architectures of hierarchically composable components.

3.2 The General Case

In the general case, the terms "upward" requirements and "downward" requirements become obsolete as they loose their semantics. In this general case, one cannot identify one component as being "over" or "under" another component. This kind of order relationships can be established only between ports that are connected to the same flow. The components have requirements associated with their ports (input ports as well as output ports). The requirements associated with input ports address the flow that comes into this port, while the requirements associated with output ports address the flow that goes out this port.

In order to be able to accurately study the interactions between components with multiple ports that are in a chain of connections, it is necessary to know for each component the relationships between its input port and output ports (the intracomponent pathways as they are named in [SW01]). In our model, the internal flows fixed by the structural constraints of a composable component identify the intracomponent pathways. Propagation of requirements in the case of components with multiple ports will occur only along the intracomponent pathways.

Given a component C, it has NI_C input ports and NO_C output ports. The requirements associated with an output port $C.Out_o$, $o \in [1 \ldots NO_C]$ are a set CO_o of properties.

A component Y has NI_Y input ports and NO_Y output ports and provides the set of properties YP. The component Y fulfills a subset $YPCO_o$ of the requirements CO_o associated with port $C.Out_o$,

$$YPCO_o = YP \bigcap CO_o. \tag{7}$$

If $YPCO_o$ is not empty, the decision to connect port $C.Out_o$ to an input port of component Y (the port $Y.In_i$, $i \in [1 \ldots NI_Y]$) is taken. The selection as connection port of the port i out of the NI_Y input ports is based on additional tests of contracts and is part of the composition strategy, hence not discussed in this section. After doing this connection, most of the cases there will still remain some requirements of Y that are not provided by $C.Out_o$, the set $YNPCO_o$

$$YNPCO_o = CO_o - YPCO_o. \tag{8}$$

[1] Paper written in 2001, delayed in publication.

In the component Y, the input port $Y.In_i$ affects only a subset YFI_i of all the output ports $Yout$ of the component,

$$YFI_i \subset YOut, YOut = \{Y.Out_o | \forall o \in [1 \dots NO_C]\} \tag{9}$$

The elements of YFI_i are those output ports of Y that are situated on intra-component pathways originating in $Y.In_i$. The properties in the set of unfulfilled requirements $YNPCO_o$ will be propagated to all the ports in YFI_i. After the propagation, at every port $Y.Out_o$ the new set of requirements YO'_o will be:

$$\forall Y.Out_o \in YFI_i : YO'_o = YO_o \bigcup YNPCO_o \tag{10}$$

This is the mechanism of propagation of requirements associated with output ports. The propagation of requirements associated with input ports is defined in a similar way.

Figure 2 presents an example of the general case of propagation of requirements.

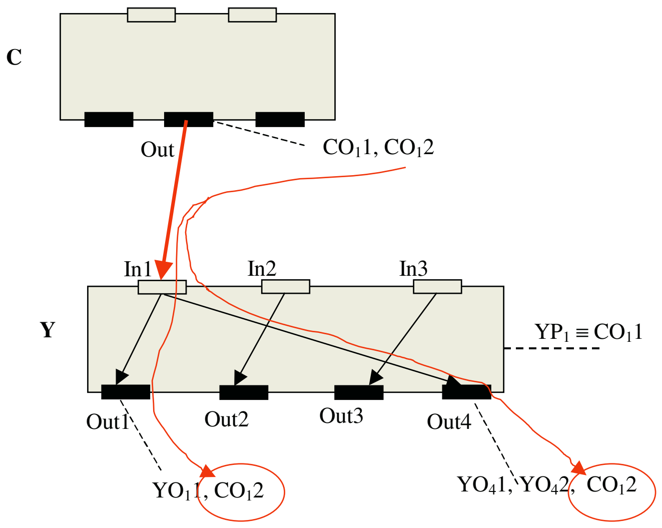

Fig. 2. Propagation of requirements – the general case

The example contains a component C that has one output port $C.Out_1$ with the associated set of requirements $CO_1 = \{CO_11, CO_12\}$.

Component Y in this example has NI_Y=3 inputs and NO_Y=4 outputs. The set of properties provided by Y is $YP = \{YP_1\}$ and it is known that property YP_1 is a match with property CO_11. Component Y has four internal flows and they are: $In_1 \rightarrow Out_1, In_1 \rightarrow Out_4, In_2 \rightarrow Out_2$ and $In_3 \rightarrow Out_3$. If port $C.Out_1$ is connected to port $Y.In_1$, this fulfills requirement CO_11 of port $C.Out_1$. The requirement CO_12 of $C.Out_1$ remains not provided yet and will be propagated to ports $Y.Out_1$ and $Y.Out_4$ (the ports that are connected with input $Y.In_1$).

4 Composition Strategy

We formulate the automatic composition problem as: *given a set of requirements describing the properties of the desired system (the composable target), and a component repository that contains descriptions of available components, the composition process has to find a set of components and their interactions to realize the desired system.* This composition decision occurs through the composition strategy implemented in a *Composer* tool.

We address the requirements driven composition of a multi-flow system by dividing it into subproblems of linear compositions on each flow of the system. Achieving fine-tuned compositions and managing the complexity of the system are possible in our approach by deploying hierarchical composable components. This leads to hierarchically recursive compositions. The driving force of the composition search are the requirements and the propagation of requirements.

First we present the strategy for linear composition on a flow. In the linear case, the composition problem is to determine an ordered sequence of components aligned on a single flow. For presentation terminology, we consider this flow to have a descending orientation. The components align in a layered form on this flow. The client level is the first layer (the "highest" one) and expresses the requirements set REQ imposed for the composable target as its downward requirements. The requirements in REQ are expressed as sets of required properties defined using the same vocabulary as that used for the component descriptions, and possibly as ordering restrictions between properties. Ordering restrictions are generated in most of the cases by the structural constraints. The set of requirements REQ results from the set of requirements CR directly imposed by the client and from the set of requirements SCR that emerge from the structural constraints of the target. Figure 3 depicts the start assumptions for linear composition.

A dummy start component C_0, having REQ as its downward requirements, is created, as Figure 3 shows. The set of requirements REQ is the current driving force for the composition. The search begins looking for components that provide at least a part of the required properties from REQ. If such a component C_x, providing part of REQ, is found, it will be connected below C_0. Component C_x has also its own requirements, upward and downward. The new downward composition driving requirements are now the downward requirements of C_x, together with the propagated part of the initial requirements and the search continues. A component is selected for the solution if it matches at least a subset of the current driving requirements. Similarly, the upward requirements of C_x become the upward composition driving set.

A solution is considered complete when the current composition driving requirements set becomes empty. It is possible that for certain sets of requirements no exact solution can be found. The *Composer* can be configured to respond to this problem in alternative ways, either to relax the client requirements and produce a solution, or to abort composition.

The general case addresses complex systems with multi-flow architecture. An example of how composition results through stepwise refinements is depicted in

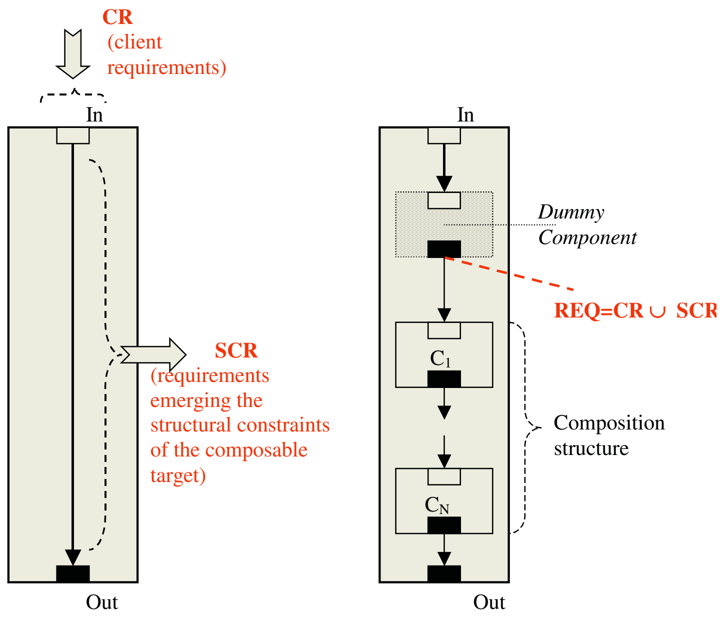

Fig. 3. Composition strategy - the linear case

Figure 4, where the composition target is the internal structure of the composable component C. The set REQ of requirements for the target results by uniting the direct client requirements and the own structural constraints of C.

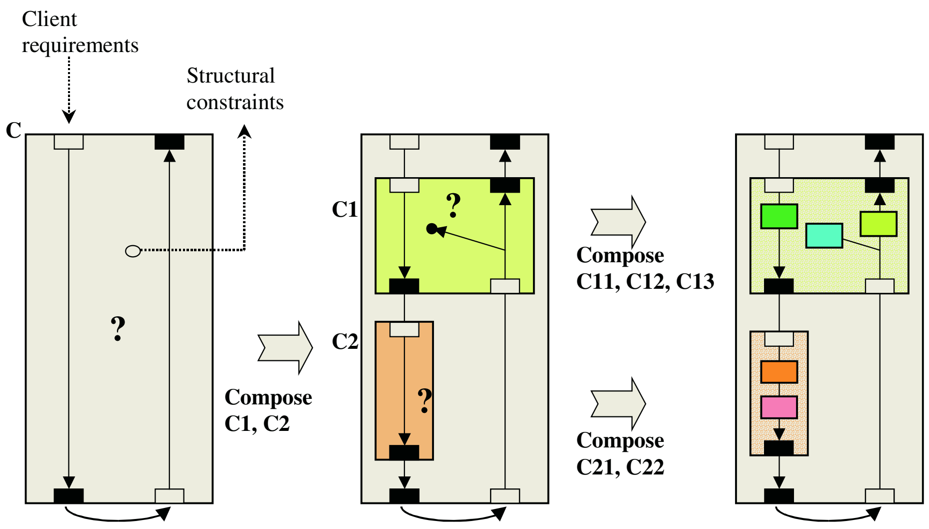

Fig. 4. Composition through stepwise refinements

After a composition search has determined that it wants certain component types ($C1$ and $C2$ in the example in Figure 4) in place to fill in the structure, a new search may be launched for composing the internal structure of these components. Such hierarchically recursive compositions will occur especially if it is necessary to satisfy subproperties of the required properties. Let the original requirements set containe a property $p1$ with the subproperties $p11$ and $p12$ and component $C1$ provide property $p1$. Component $C1$, found to provide $p1$, will have to be fine-tuned so that its internal structure is compliant to the set of subrequirements $p11$, $p12$. The set of required properties $p11$, $p12$ represent direct client requirements for the composition of target $C1$. Together with the structural constraints for $C1$, these requirements lead to the composition of the internal structure of $C1$ from components $C11$, $C12$, $C13$.

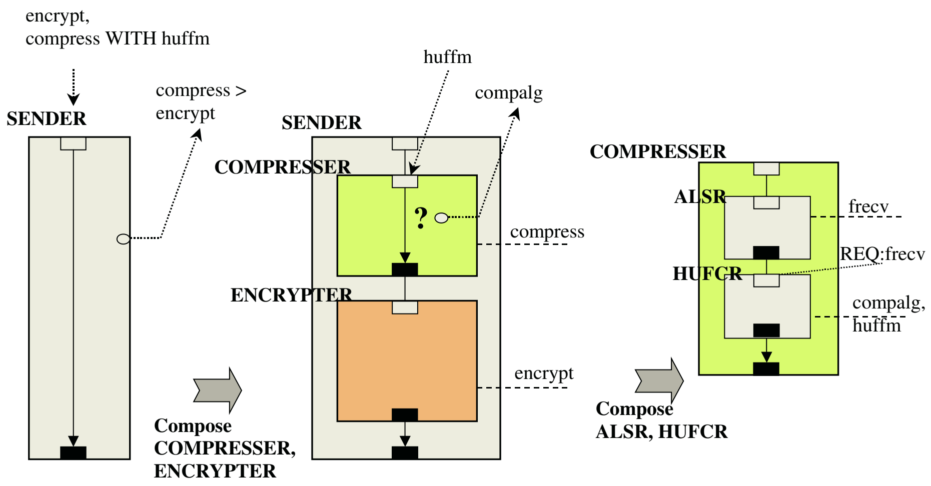

Fig. 5. Composition example

As an example illustrating the concepts presented, we consider the simple scenario depicted in figure 5. The compositional target is a $SENDER$ component that can be customized according to different client requirements. The client requirements in this example contain both encryption and compression of sent data with a particular compression algorithm. These requirements are expressed as properties from the universal properties vocabulary, *encrypt* and *compress*. Property *compress* is specified with a subproperty $huffm$ (compression with the Huffmann algorithm is required). The structural constraints of the $SENDER$ in this example state the ordering restriction that if the property *compress* is present, it should be on the internal flow above property *encrypt*. Through propagation of all these requirements results immediately the solution of composing $SENDER$ from the components $COMPRESSER$ and $ENCRYPTER$ as in the figure. (Without the ordering restriction, both sequences $COMPRESSER$ before $ENCRYPTER$ and $ENCRYPTER$ before

COMPRESSER are possible.) Further, the component *COMPRESSER* is also a composable one and its internal structure must be composed according to the current client requirements, specified by property *huffm*. The structural constraints of the *COMPRESSER* component specify that it must contain the property *compalg* on its internal flow (it must contain the implementation of an arbitrary compression algorithm). Starting from these requirements the solution search on the internal flow of *COMPRESSER* begins. Component *HUFCR* provides properties *huffm* and *compalg* and will be deployed inside *COMPRESSER*. *HUFCR* requires property *frecv* at its input port (as Huffmann compression requires a static analysis of the data), thus the search continues in the upward direction of the flow and adds the analyzer component *ALSR* which provides *frecv*. Since no requirements are left unfulfilled, component *COMPRESSER* is ready composed from components *ALSR* and *HUFCR*.

5 Discussion and Related Work

A distinctive characteristic of our composition approach is that it works with abstractions of the architectural level. This is according to our insight that architectural style dependent composition models, independent from the application domains, are needed. This permits a generic solution, avoiding coding of specific solutions for each application domain. The approach is to build a system by assuming a certain defined architectural style. Treating component composition in the context of the software architecture is a largely accepted approach ([Ham02], [Wil03], [IT03],[BG97], [KI00], as it makes the problem manageable and eliminates the problems of architectural mismatch. Also, we argue that at the architectural level compositional decisions are made with knowledge of the architectural style, but ignoring technological details of the underlying component model, as long as this provides the infrastructural support needed for runtime assembly of components. Components are described through their properties, seen as facts known about them – in a way similar to Shaw's credentials ([Sha96]). The composition strategy interprets properties in a semantic-unaware way by having a general matching criteria, thus no application-domain specific code occurs in the *Composer*. The general composition strategy described in this paper emerges from our experience with automating composition in two different application domains where systems have multi-flow architectures – self customizable network protocols ([ŞMBV03], [ŞVB02]) and an intelligent environment for virtual instrumentation in measurements and control.

The mechanism of propagation of requirements used in our approach is a generalization rooted in Perry's mechanism of propagation introduced in [Per87], [Per89]. Perry defines a semantic interconnection model for the verification of program semantics, at the level of procedural programming. It extends Hoare's specification of program semantics with pre- and postconditions, proposing another category of clauses, the obligations. Preconditions must be satisfied by the postcondition of an operation that follows on the control flow. Obligations are conditions that must be satisfied by postconditions of operations that precede them on the control flow. In Perry's mechanism, preconditions and obligations are propagated to the interface of the containing module. Our upward requirements may be similar to preconditions, obligations to downward requirements and postconditions to provided properties.

Perry's model deals with the composition of small-grained entities: procedures and functions. Batory et. al. ([BG97], [BCRW00]) propose a similar model for the composition of components in *GenVoca* architectures (layered systems). The entities subject to composition are components, implemented as classes, and are used in layered compositions that can be seen as components being put on top of each other. From the compositional point of view, these components can distinguish only between two interaction points, one upper and one lower interaction point. A particularity of their approach is that a layer provides different properties for the layer on top of it as it provides for the layer below it. This leads to two kinds of pre- and postconditions. Postconditions are named the properties that are provided to the components below it and postrestrictions the properties provided to the components on top. Preconditions are requirements that are directed toward components on top while prerestrictions are requirements directed to components below. In [BG97] an algorithm for the verification of the correctness is given, verification done by downward propagation of postconditions and upward propagation of postrestrictions.

Our approach brings two important contributions. First, we generalize the principle of propagation to non-linear structures. Also we adapt it in the context of components. Our model considers that the provided clause is associated to the component as a whole; a component provides the same properties to all its interacting entities. Requirements are associated with individual ports of the component.

Second, the goal of our model is to serve the automatic component composition (to *generate* the structure of the target assembly) rather than only the verification of a given assembly structure as in the related works. The mechanism of propagation of requirements is the driving force of our searching algorithm. Therefore, in our model the propagated elements are the required properties and not the provided properties.

Our current implementation of a composition algorithm does exhaustive searches and thus has the disadvantage of exponential time. We foresee to implement improvements of it using a search based on heuristics. The mechanism of propagation of requirements as described in this paper will remain a central element of the search.

Our work tackles composition decisions at the semantic level. Other research in automating the composition or adaptation of components deal with the problem at the behavioral level. Different kinds of finite automata or *message sequence charts* (MSC) are used to model the behavior of components ([SR00], [SVSJ03], [IT03]). The behavioral compatibility tests for components check the matching according to syntactic and synchronization criteria. Without considering also semantic level information it is possible that a behavioral test declares as compatible semantical different components that happen to have compatible automata, so semantic checking is needed together with behavioral checking.

6 Conclusions

Our research defines a compositional model for multi-flow architectures that comprises

- a scheme and a language for the description of composable components by semantic–unaware properties and structural constraints.
- a requirements driven composition strategy capable to implement automatic composition decisions starting from the descriptions of the available components and from the requirements for the compositional target.

In this paper, we presented the principles of our composition strategy, introducing the mechanism of propagation of requirements as the central element of our composition strategy. This strategy is implemented by an automatic *Composer* tool that facilitates the building of self-customizable systems. The strengths of our strategy are its simplicity, its application domain independence, and the possibility to compose unanticipated configurations.

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.

BCRW00. Don Batory, Gang Chen, Eric Robertson, and Tao Wang. Design wizards and visual programming environments for GenVoca generators. *IEEE Transactions on Software Engineering*, 26(5), May 2000.

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

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

IT03. Paola Inverardi and Massimo Tivoli. Deadlock-free software architectures for COM/DCOM applications. *Journal of Systems and Software, Special Issue on Component-Based Software Engineering*, 65(3):173–183, March 2003.

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

Per87. Dewayne E. Perry. Software interconnection models. In *Proceedings of the 9th International Conference of Software Engineering*, pages 61–69, Monterey CA, USA, May 1987.

Per89. Dewayne E. Perry. The logic of propagation in the Inscape environment. In *Proceedings of SIGSOFT '89: Testing, Analysis and Verification Symposium*, Key West FL, USA, December 1989.

Sha96. Mary Shaw. Truth vs knowledge: The difference between what a component does and what we know it does. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 181–185, 1996.

SR00. Heinz Schmidt and Ralf Reussner. Automatic component adaptation by concurrent state machine retrofitting. Technical Report 2000/81, School of Computer Science and Software Engineering, Monash University, Melbourne, Australia, 2000.

SVSJ03. Pieter Schollaert, Wim Vanderperren, Davy Suvee, and Viviane Jonckers. Online reconfiguration of component-based applications in PacoSuite. In *Proceedings of Workshop on Software Composition, affiliated with ETAPS 2003*, volume 82 of *Electronic Notes in Theorethical Computer Science*, Warsaw, Poland, 2003. Elsevier.

SW01. J. A. Stafford and A. L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–452, August 2001.

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

ŞMBV03. 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.

ŞVB02. 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.

ŞVB03. Ioana Şora, Pierre Verbaeten, and Yolande Berbers. A description language for composable components. In Mauro Pezze, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, Proceedings*, number 2621 in Lecture Notes in Computer Science, pages 22–36. Springer Verlag, 2003.

Wil03. 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.