

Automatic composition of software systems
from components with anonymous
dependencies

Ioana Şora
Frank Matthijs

Report CW 314, May 2001



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Automatic composition of software systems from components with anonymous dependencies

Ioana Şora
Frank Matthijs

Report CW 314, May 2001

Department of Computer Science, K.U.Leuven

Abstract

We present a mechanism for automatically composing systems based on finding a layered architecture of components to satisfy system-level requirements. We consider components with anonymous dependencies between them, expressed through required and provided properties. We propose an application-domain independent formalism for describing the client-specific configuration requests in terms of desired properties, and a composition algorithm that works well in these conditions.

Secondly, we argue that the composition method is independent from the application domain, only architecture-style dependent, which allows a larger reuse of the composition method. Configuration knowledge that is specific to a certain application domain may be incorporated in domain-specific front-end tools that accept client requirements expressed at a higher abstraction and translate them in the terms of a domain-unaware description language.

We illustrate by giving an example of automatic composition of protocol stacks within DiPS, our Distrinet Protocol Stack framework for building network subsystems. The integration of an automatic composition module into DiPS has validated our approach as a simple but powerful tool for customizing software to support client-specific requirements.

Automatic composition of software systems from components with anonymous dependencies

Ioana Şora, Frank Matthijs
DistriNet labs, Department of Computer Science
Katholieke Universiteit Leuven, Belgium
E-mail: {Ioana.Sora, Frank.Matthijs}@cs.kuleuven.ac.be

Abstract

We present a mechanism for automatically composing systems based on finding a layered architecture of components to satisfy system-level requirements. We consider components with anonymous dependencies between them, expressed through required and provided properties. We propose an application-domain independent formalism for describing the client-specific configuration requests in terms of desired properties, and a composition algorithm that works well in these conditions.

Secondly, we argue that the composition method is independent from the application domain, only architecture-style dependent, which allows a larger reuse of the composition method. Configuration knowledge that is specific to a certain application domain may be incorporated in domain-specific front-end tools that accept client requirements expressed at a higher abstraction and translate them in the terms of a domain-unaware description language.

We illustrate by giving an example of automatic composition of protocol stacks within DiPS [Mat99], our DistriNet Protocol Stack framework for building network subsystems. The integration of an automatic composition module into DiPS has validated our approach as a simple but powerful tool for customizing software to support client-specific requirements.

1 Motivation

The goal of software composition is to find a good combination of components that leads to a software system that responds to client-specific requirements.

Earlier research in the domain of composition has addressed the issue within the domain of software architectures, ADL's [SDZ96] and composition languages [SN99]. In these cases, deciding a good component combination is done statically and relies on the application programmer. An important research issue is to automate this configuration process.

Another important challenge in current computer systems is that they must be able to configure themselves dynamically, adapting to the environment in which they are executing. In the case of dynamic configuration, the issue of automatic vs. manual configuration is even more important.

There is previous and ongoing research in the domain of dynamic and automatic configuration of component-based systems ([Kon00], [BCRP98] [HF98], [AW99], [TJJ00]). An essential step towards the possibility of implementing services that support automatic configuration is a good explicit representation of dependencies. In [KC00], a model for representing dependencies among components and mechanisms for dealing with these dependencies is proposed. Prerequisite specifications (including the nature

and capacity of needed hardware as well as required software services) reify static dependencies of components towards their environment, while component configurators reify dynamic, runtime dependencies. The software requirements are directly expressed by means of explicit references to components from a component repository. Some of them may also be optional, but the task of the automatic configuration service is only to load the indicated components. Also the component configurators, which are responsible for reifying the runtime dependencies for a certain component, deal with explicit dependencies among components.

One problem in the application composition is the determination itself of the component dependencies. Often, dependencies can only be expressed indirectly, in terms of a set of properties that have to be provided by an unknown provider from the environment, including also other components. We argue that *anonymous dependencies*, expressed indirectly by means of *required/provided properties* and established on the fly should be taken into account. This leads to more complex composition algorithms.

In this paper we propose a model for specifying the component properties which are relevant for the purpose of composition, as well as a method for determining dependencies between components dynamically during composition and for achieving a good composition of components that satisfies the client-specific requirements.

Apart from establishing mechanisms and policies for automatic composition, one should address the usability and user friendliness of automatic composition. Often, application programmers who wish to customize the way their applications are executed are confronted with a domain that is different from their familiar application domain. The solution to this problem is to permit specifying the requirements in a sufficiently high-level style. [CE99] introduces the notion of *configuration knowledge* to fill in the gap between problem and solution space. Considering the solution space consisting of the components with all their possible combinations and the problem space consisting of application-oriented concepts and features that application programmers would like to use to express their needs, the configuration knowledge consists of illegal feature combinations, default settings, default dependencies and construction rules. A second contribution of this paper is that we propose a generic way of taking the configuration knowledge into account during the configuration process.

This paper is structured as follows. The next section introduces our component model with the component descriptors and our automatic composition strategy. In section 3 we illustrate the approach using our DIPS framework for protocol stack composition. We propose a generic way to take in account and separate domain-independent and domain-dependent configuration knowledge during the composition process in section 4.

2 Component descriptors and composition strategy

Component-based software development is often driven by an underlying component framework. A component framework offers a set of predefined plug-compatible components and sets the rules of how components can be instantiated and composed.

We consider an application as being a number of components that are connected by connectors. Our current work on composition considers layered architectures. In a simplified model, each layer is a component that provides a specific set of services, which can be used by other layers on top of it. The composition process, which is intrinsically architecture dependent, exploits the layer property of incremental

enhancement of services. If the current layer does not provide enough functionality for a given application, the service may be enhanced step by step by adding other layers.

During the decisional process of the composition, only *lightweight component representations* are needed. These correspond to simplified *component descriptions* that include only the part of the component specification that is relevant during the decisional process. This simplified component description specifies the features that can be provided by the component and the requirements that it imposes towards other components or the environment.

Each component is described by a list of *provided properties*. A *property* is a rather abstract feature (a name). We do not make differentiation about the different semantic categories of properties and argue that their semantics is not relevant in the composition process. In order to be able to provide those properties, each component requires certain conditions to be satisfied by others. As a representation of the requirements of each component we use lists of *required properties*. The solving strategy of our composition algorithm is “matching requested properties with provided properties”.

We classify the requirements with regard to the direction of their target (target that is anonymous), and according to their strictness. Figure 1 gives a generic overview of the main concepts that we assume for our component description model.

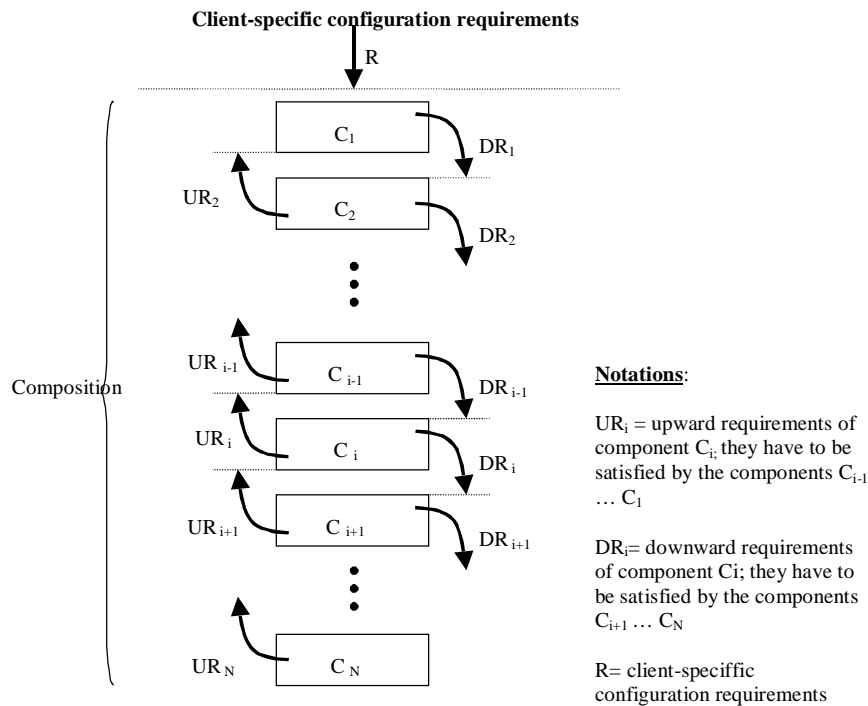


Figure 1. Upward and downward requirements of the components in a layered composition

We distinguish *downward requirements*, imposed by a component towards components that are in layers below it. We consider also *upward requirements*, imposed by a component towards components that are in layers on top of it.

Depending on their strictness, we can consider *strong* and *weak* requirements. The strong requirements must be fulfilled in order to yield a correct composition. The weak requirements should only not be contradicted by the composition solution (which means, for example, if a component C states property r as an weak upward requirement, then it is not allowed to have r provided by a component below C, but it is not necessary

to have r present above C). For the majority of the requirements, it is sufficient that they are met by some component that is present somewhere in the path above (for upward requirements) or below it (for downward requirements). Specifying *immediate* requirements is also possible, meaning that those requirements apply only to the immediate adjacent layer. By default, no order is assumed in meeting a list of required properties. A succession of components C_1, C_2, C_3 may be a solution as well as permutations of this sequence, if the additional requirements of each individual component do not eliminate permutations. An ordering preference may be specified, stating whether a property from a requirement list has to be met strictly before or after another property. Such ordering preferences are specified only in the case that the requirements of the candidate components do not impose a unique ordering as desired by a specific application.

The components are described in a component repository. Each component repository entry is a component description which contains information about the functionality provided by the component and its requirements. We don't imply a particular specification formalism, but as an example, Figure 3 presents the way how components are described in our implementation.

As said before, we use an approach of matching requested properties with provided properties as solving strategy. The composition algorithm produces solutions as sequences of component descriptions. Assuming the set of client-specific configuration requirements $R = \{R_1, R_2, R_3, \dots, R_r\}$, a succession of components C_1, C_2, \dots, C_N represents a good composition, if:

1. all requirements R_i are met, being present in the provided properties list of a component C_j , for all components C_j
2. each component C_j has its own downward requirements list DR_j accomplished by some components $C_i, i=j+1 \dots n, i > j$ (components C_i that follow C_j in the succession)
3. each component C_j has its upward requirements list UR_j accomplished by some component $C_i, i=1 \dots j-1, i < j$ (components C_i that precede C_j in the succession)
4. additional imposed ordering restrictions are met
5. there are no contradictions with respect to the weak requirements

Our approach for finding the suitable composition is a top-down searching. The client-specific requirements R are associated with a component C_0 . Components are added to the sequence representing the good composition by starting from C_0 and the client-specific requirements. The searching of the solution is driven by the downward requirements, since in the layered architecture, a layer at level i makes use of services of layers at level $i+1$ (requests something to be provided by them) and offers its services to layers at level $i-1$ (provides services for the upper layers).

We introduce the mechanism of *propagation of requirements* during the solution searching process as an essential element in our strategy. This is a mechanism that implements the principle of delegating the responsibility for solving certain requirements posed on a component, to other components.

Figure 2 illustrates the mechanism of propagation of requirements by an example. If a component C_2 provides some of the properties required by a component C_1 , and C_2 is chosen to be a component below C_1 , then all the properties that are required by C_1 and **not** provided by C_2 are *propagated* to C_2 . The requirements that are propagated to a component are added to the own requirements of this component and treated in a non-discriminatory way in further propagation steps.

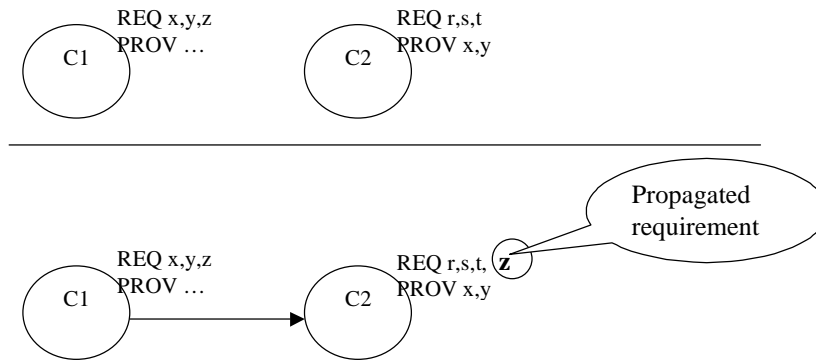


Figure 2. The mechanism of propagation of requirements

The category of *immediate requirements* is the exception in the sense that it is not subject to the rule of requirement propagation. Those immediate requirements are strictly bound to a certain component, and their resolving can not be delegated to other layers.

The starting point of the composition strategy consists in searching at least one component that provides some of the properties requested by the application. If such a component is found, in most of the cases there will still remain unsatisfied requirements. Also, it is most likely that the component found to (partially) fulfill the application requirements has its own set of requirements. The requirements of the application that have not yet been met, together with the own requirements of the new component, form a new problem for the composer. This problem has to be approached in a similar manner: step by step new components, that solve some of the dynamically updated requirements list, will be added.

The composition algorithm exploits the fact that the dependencies between components naturally form a graph structure. This graph has component descriptions as nodes, while the edges represent the dependencies between them. A directed edge from the node with the component description of C_i to the node of C_j exists when C_j provides at least one property that is requested as a downward requirement by C_i , either as its own or propagated requirement. The graph has a very dynamic structure due to the fact that new requirements may appear at every moment through the mechanism of propagation. Edges are added and removed, as requirements are propagated and solved.

During the composition process, if the component described in a node N_1 has been just added to the solution sequence, the problem is to determine its successor, a component that may be added below it. The list of potential successors of a node N_1 contains all nodes N that provide at least one property that is currently required as downward requirement (directly or by propagation) by N_1 . The list of potential successors of N_1 is found in the graph in the neighborhood of that node (its adjacencies list). This initial list is reduced by applying a few exclusion criteria, like: semantic redundancy (don't add components that repeat some of the properties already provided); good ordering (if property "p1" is still in the requirements list, with the ordering option "p1>p2", don't add yet a component that provides property "p2"); no contradiction of weak requirements (don't add a component that provides a property "p1" if there is already on top in the sequence a component requesting "p1" as weak upward requirement). The downward requirements represent the motor of the searching, as stated above, but also the upward requirements are evaluated. If a component from the list of potential successors has strong upward requirements, then components that provide these properties have to be inserted in the sequence. The right place for

insertion in the sequence has to be found so that no contradictions with existing properties are created. These components, inserted in the sequence as consequence of upward requirements may yet introduce new downward requirements, that update the whole propagated requirements list, yielding a new graph structure.

It is possible that for certain client-specific requirements no solution can be given using components as found in the component repository. We can choose what strategy to adopt in this case, to cancel the composition process or to continue and find solutions that only partially fulfill the client requirements. Also, as always when things are generated automatically, the composed system may not be the best solution in all cases.

3 Composition example

We have validated the automatic composition approach described above by implementing automatic configuration of protocol stacks. Since currently protocol stacks operate in various contexts and it is therefore not possible to know the required properties of a stack (both functional and non-functional) in advance, stack configuration mechanisms are needed. The configuration of the stack should be as automatic as possible and the user shouldn't need to be bothered with the communication subsystem that should just work in any environment.

The run-time support for coping with dynamic protocol stack changes is assured by DiPS (Distrinet Protocol Stack) [Mat99], our framework for building network subsystems. The DiPS framework provides the necessary support for the plumbing of layers and components, and run-time support for the dynamic behavior. The DiPS framework is able to build stacks with different degrees of customizability.

DiPS provides the needed infrastructure support, that comprises:

- a stack building framework, that contains a special stack builder which is able to interpret at run-time a sequence of component descriptions and build a protocol stack from it
- a component repository
- support for loading of components and creation of the corresponding stack layers
- a generic application endpoint(socket) that completely hides the structure of the stack's underlying composition from an application

We consider as a simple intuitive example an application that needs reliable non-local transport.

Suppose that available protocol building blocks are UDP, REL (Reliability layer), TCP, IP, ETH (Ethernet driver) and that an Ethernet network interface (EthNI) is available in the current configuration. These building blocks are retrieved from a component repository which contains information like described in Figure 3.

The network interface imposes the strict, immediate upward requirement to have the appropriate driver in the layer immediately above. The "application component" has as downward requirements "transp", "non-local", "rel", "interf". As ordering preferences, the network interface has to be at the bottom of the stack, so every other required property has to be met before "interf", we express this ordering requirement in our language for requirements description as "`* > interf`". As an additional option, reliability can be requested to be provided at an upper layer, "`rel >= transp`".

For a uniform treatment, component descriptions for the application and the network interface are added in our approach.

```

COMPONENT REL
PROV rel
END

COMPONENT UDP
PROV transp
END

COMPONENT IP
REQU(WEAK) transp
PROV non_local
END

COMPONENT TCP
PROV rel transp
END

COMPONENT ETH
PROV datalink
END

INTERF EthNI
REQU(IMMEDIATE) eth
END

```

Figure 3. Information contained in a Component repository

At the beginning, a component repository is interrogated, and every component descriptor that could provide some of the requirements is inspected. Further, the remaining requirements from the initial list together with the consequent new requirements introduced by the added components form the new problem that has to be addressed. The stack composition will be discovered step by step, each step adding a new component to the stack, in a top-down manner (the stack “grows” starting with the upper layers and advances toward the network interfaces).

In Figures 4 and 5, two initial steps during the evolution of the dependency graph are depicted. As a graphical notation used in this figure, the arrows point to the nodes that are adjacent with the current node, meaning that they will be considered as potential successors in the composition sequence. Each arrow is labeled with the corresponding propagated requirements.

In the first step, according to the application requirements (“reliable”, “non-local”, “transport”, “interf”), five components (UDP, REL, IP, TCP, and ETHNI) are found to each partially fulfill those requirements. But since the ordering preferences stated that “interf” should be in the last layer (and there are still unresolved requirements), EthNI is excluded from the list of potential successors. Also UDP is excluded, since it provides “transp” but “rel” is still in the requirements list and has to be above “transp”. The components that remain in the list, REL, IP and TCP, do not satisfy alone, none of them, all the application requirements. REL satisfies only the reliability property, so the “non-local”, “transp” and “interf” requirements will be propagated. TCP satisfies the reliability and transport properties, so the “non-local” and “interf” requirements will be propagated. To IP, “transp”, “rel” and “interf” are propagated.

The second step supposes picking one of the components selected in the first round and trying to resolve its requirement list. When trying a continuation of the stack started with REL, the components UDP, IP, EthNI and TCP are found to provide some of the requirements. EthNI will be eliminated again as a possible successor due to the ordering preference and TCP will be eliminated due to redundancy (it provides again the reliability property).

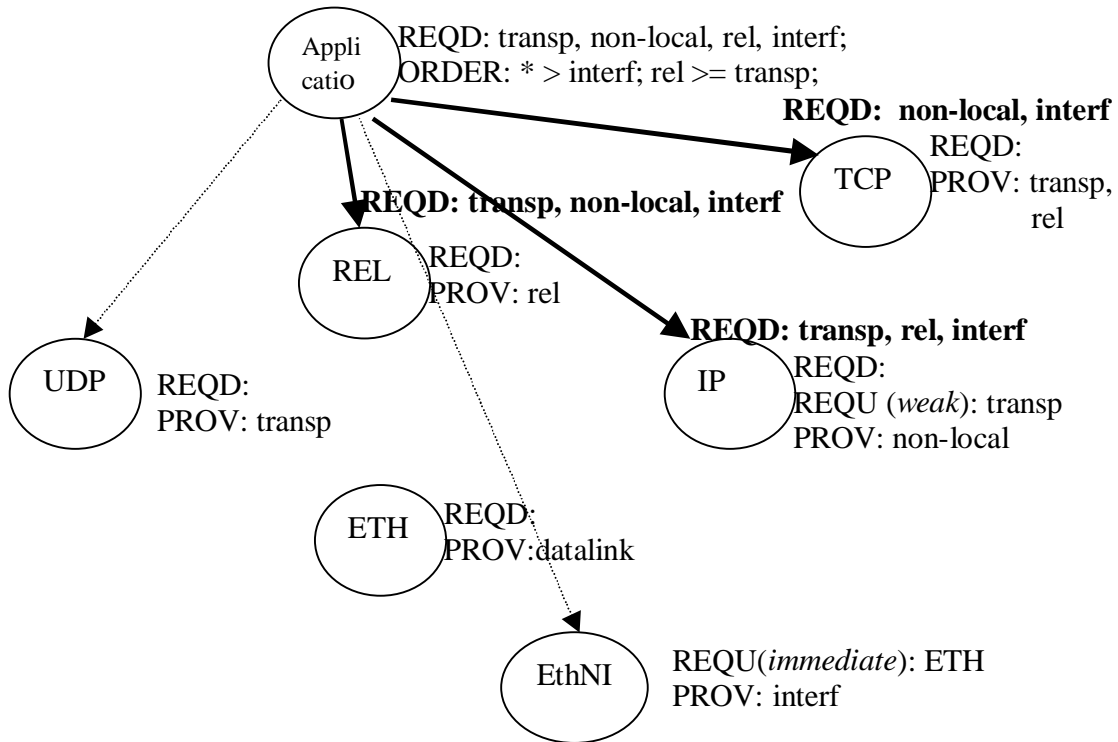


Figure 4 Example - finding a protocol stack composition according to client-specific requirements; start step

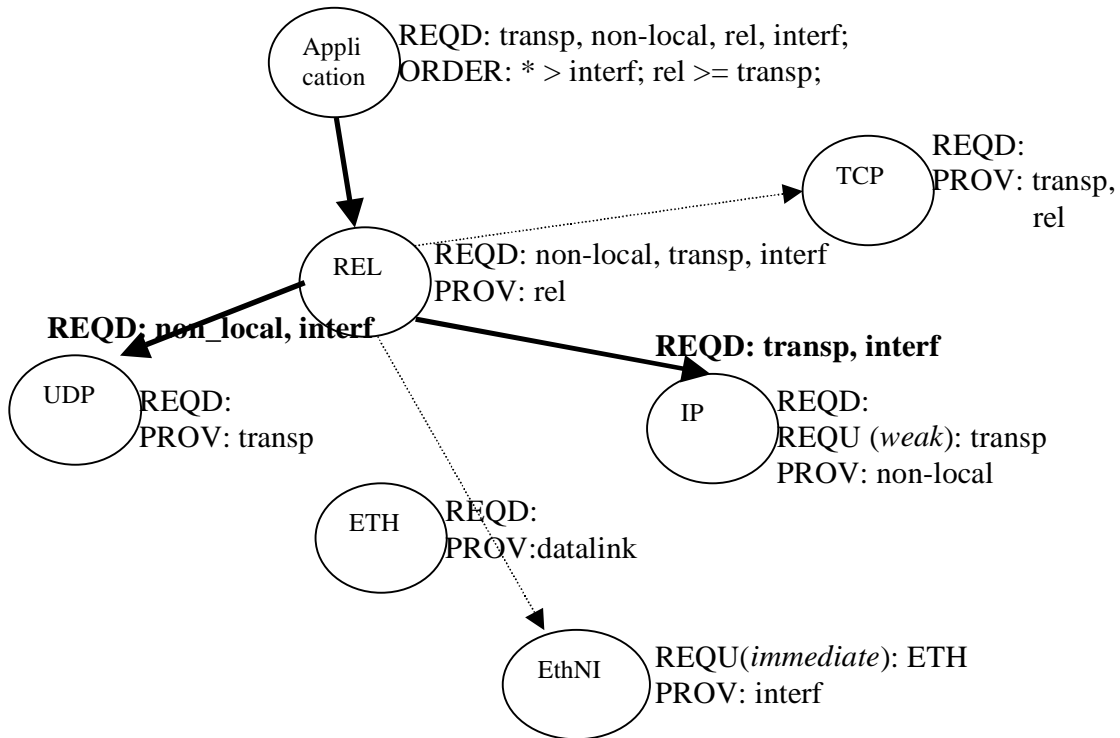


Figure 5. Example - finding a protocol stack composition according to client-specific requirements; second step

The searching continues until finding REL – UDP – IP- ETH – EthNI as a good solution. The path that was started with TCP will also lead to a good solution, TCP-IP-ETH-EthNI. The paths that were started with IP and REL-IP do get rejected later, when trying to add a component providing “transp”, since IP had as weak upward requirement “transp”. This weak upward requirement obstructs from having the network protocol on top of the transport protocol, but does not actually enforce the presence of a transport protocol on top of the network layer.

4 Domain-specific description of client-requirements

The strategies used for configuration and composition are not dependent on the application domain. These are policies that apply generally to systems that are of the same architectural style, but do not interfere with the application domain. On the one hand, we want to use the same composition strategy for a whole family of composition problems sharing the same architectural style, on the other hand applications should not be confronted with the problem of stating their requirements in a form that matches the underlying architecture style formalism. Applications should be able to specify their requirements in a sufficiently high-level style so that application programmers who wish to customize the way their applications are executed are not confronted with a domain that is different from their familiar application domain.

Therefore we argue that the problem of finding a good composition that suits client-specific configuration requirements may and should be split in two parts, like depicted in Figure 6.

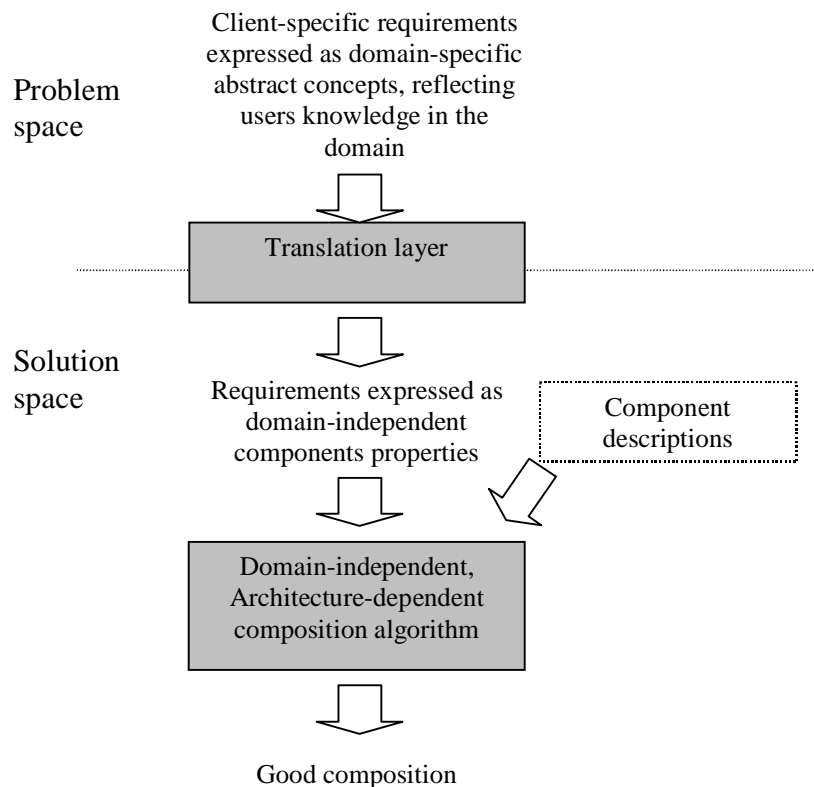


Figure 6. Solving automatic composition based on application-domain specific requirements in an application-domain independent way

One part is to define application-domain independent and only architectural-specific composition policies. This makes possible that the same composition method is reused for different application domains. Reuse is approached not by domain, but by specific architectural style. The other part is to define translation layers that may be used for encapsulating the application domain knowledge, assuring a translation of requirements expressed as domain-specific abstract concepts, reflecting user’s knowledge in the domain, into requirements expressed as domain-independent component properties.

In our example, the networking application domain, the protocol stack user who is also the application developer must make a mapping between the end user understandable configuration settings and the more technical configuration settings which implement the user requirements on the protocol level. The problem here is that typical application programmers are not network specialists and will find it difficult to express application customizations in terms of an unfamiliar domain.

Our solution, to deploy a translation layer, would enable the stack user to express requirements on a higher, more abstract level, depending on the user expertise. The translation layer would translate these high level requirements into the requirement directives understandable to the protocol stack composition algorithm. In our example regarding protocol stack composition, the expression of preferences regarding the network service from the viewpoint of the end user may be not technical at all. The main issue that arises is the specification complexity of the requirements. For a user who doesn’t know anything about the possibilities of the underlying protocol stack framework, the type of quality of service settings this kind of user expects would for instance be to specify “send an urgent message” or “send a message with an as low cost as possible”. For application developers who have a better understanding of the stack framework, the possibility of specifying the wanted requirements in a more specific way (like “an unreliable multimedia communication over a wireless network”) should also be possible. It is necessary to be able to give the programmer (or end user) more control over the use of the protocol stack. We considered a diversification of users, each of them having a different amount of technical expertise with respect to networking, like indicated in Figure 7.

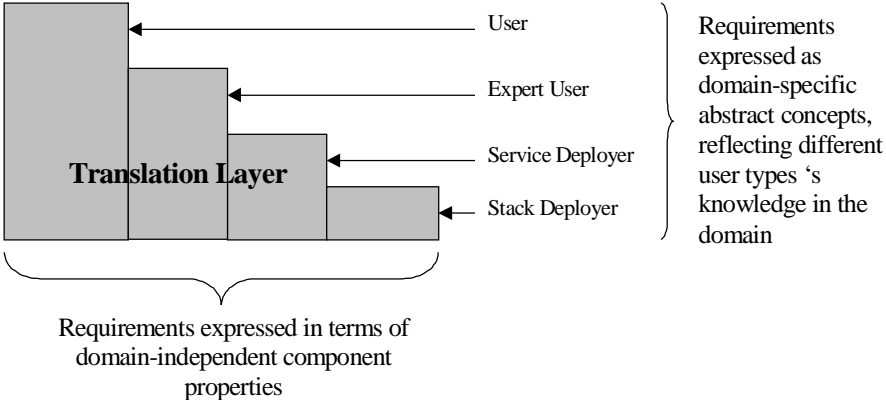


Figure 7. Domain-specific translation layer

The same composition module may be reused in different application domains, by deploying domain-specific front-end tools (translation layers) that accept client requirements expressed in a description language with a higher, domain-specific abstraction level and translate them in the terms of a domain-unaware description language.

5 Summary

In this paper, we advocate the use of automatic composition as an intermediate between applications and component framework technology. Our insight is that the strategy for the composition process can and should be independent from the application domain, only architecture-style dependent. Configuration knowledge that has to be used during the composition and that is specific to a certain application domain may be incorporated in domain-specific front-end tools that accept client requirements expressed at a higher abstraction level and translate them in the terms of a domain-unaware description language. This allows a larger reuse of the composition method across different application domains.

Our observations are based on our experience with defining and implementing a composition mechanism for layered architectures. We propose a manner of describing the client-specific configuration requests, of specifying component descriptions and a composition algorithm that works well in those conditions. The client-specific requirements are expressed indirectly, in terms of desired properties with help of an application-domain independent descriptive language. The specifications of the components are oriented on provided/requested properties and do not express explicit dependencies. Explicit dependencies are dynamically discovered during the composition process. The approach we have presented provides a simple but powerful tool for customizing software to support client-specific requirements.

Using the networking domain as application domain example, we illustrate how our composition algorithm finds a good composition of a protocol stack that solves client-specific requirements. Developing a prototype that integrates an automatic composition module into our DiPS component framework we have validated the automatic composition approach described in this paper.

6 References

- [AW99] Noriki Amano, Takuo Watanabe, An Approach for Constructing Dynamically Adaptable Component-based Software Systems using LEAD++, OOPSLA'99 Workshop on Reflection and Software Engineering
- [BCRP98] Gordon S. Blair, Geoff Coulson, Phillippe Robin, and Michael Papathomas, An Architecture for Next Generation Middleware, Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware' 98), Lake District, UK, Editors: Davies, N., Raymond, K., Seitz, J., Springer-Verlag, 1998.
- [CE99a] Krzysztof Czarnecki, Ulrich Eisenecker, Components and Generative Programming, Proceedings of ESEC/FSE'99, LNCS 1687, pp. 2-19, 1999
- [CE99b] Krzysztof Czarnecki, Ulrich Eisenecker, Synthesizing Objects, Proceedings of ECOOP'99, LNCS 1628, pp. 18-42
- [DFY98] Sunshil Da Silva, Danilo Florissi, Yechiam Yemini, Composing Active Services in NetScript, DARPA Active Networks Workshop, 1998
- [HF98] Richard Hayton, Matthew Faupel, FlexiNet: Automating Application Deployment and Evolution, Workshop on Compositional Software Architectures, Monterey, California, January 6-8, 1998

- [Kon00] Fabio Kon, Automatic Configuration of Component-Based Distributed Systems, PhD Thesis, University of Illinois at Urbana-Champaign, 2000
- [KC00] Fabio Kon, Roy Campbell, Dependence Management in Component-Based Distributed Systems, IEEE Jan-March 2000
- [Mat99] Frank Matthijs, Component Framework Technology for Protocol Stacks, PhD Thesis, Katholieke Universiteit Leuven, December 1999
- [ML98] Mira Mezini, Karl Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development, in Proceedings of OOPSLA'98
- [PCCB] Nikos Parlavantzas, Geoff Coulson, Mike Clarke, and Gordon Blair, "Towards a Reflective Component Based Middleware Architecture", in Workshop on Reflection and Metalevel Architectures, June 13, 2000, Sophia Antipolis and Cannes, France.
- [SDZ96] Mary Shaw, Robert DeLine, Gregory Zelesnik, Abstractions and Implementations for Architectural Connections, in Proceedings of the International Conference on Configurable Distributed Systems, Annapolis, Maryland, 1996,
- [SN99] Jean-Guy Schneider, Oscar Nierstrasz, Components, Scripts and Glue, in Software Architecture- Advances and Applications, Leonor Barroca, John Hall and Patrick Hall (Eds.), Springer, 1999
- [Ter99] S. Terzis, P. Nixon, Component Trading: The basis for a Component-Oriented Development Framework, 4th International Workshop on Component-Oriented Programming (WCOP 99), at ECOOP 99, June 1999
- [TJJ00] Eddy Truyen, Bo N. Joergensen, Wouter Joosen, "Customization of Object Request Brokers through Dynamic Reconfiguration", in Proceedings of Tools Europe 2000, June 2000, Mont-St-Michel, France