

# Policies for dynamic stack composition

*Ioana Şora  
Sam Michiels  
Frank Matthijs*

*Report CW 313, February 2001*



**Katholieke Universiteit Leuven**  
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Policies for dynamic stack composition

*Ioana Şora*  
*Sam Michiels*  
*Frank Matthijs*

*Report CW 313, February 2001*

Department of Computer Science, K.U.Leuven

## **Abstract**

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. The stack has to be dynamically built up from components, based on the requirements and the momentary situation. The first step in building the stack is to determine the component types to be used and the stack architecture that has to define the way building blocks are connected. In this document we report on how to describe protocol building blocks, external requirements and how to build a stack from these descriptions. We give a detailed description of our composition algorithm.

# POLICIES FOR DYNAMIC STACK COMPOSITION

Ioana Şora, Sam Michiels and Frank Matthijs  
Department of Computer Science,  
Katholieke Universiteit Leuven.  
Celestijnenlaan 200A, 3001 Heverlee, Belgium  
{Ioana.Sora, Sam.Michiels, Frank.Matthijs}@cs.kuleuven.ac.be

## Abstract

*Currently, 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. The stack has to be dynamically built up from components, based on the requirements and the momentary situation.*

*The first step in building the stack is to determine the component types to be used and the stack architecture that has to define the way building blocks are connected. In this document we report on how to describe protocol building blocks, external requirements and how to build a stack from these descriptions. We give a detailed description of our composition algorithm.*

## 1. Introduction

### 1.1 Goals

In the context of a generic service platform where the services are distributed over a number of tiers and where access to the services is to be provided in a flexible way, the Universal Access to Services supposes specification and implementation of a set of basic protocol and service components, that can be distributed over different access network elements and that can flexibly adapt to various environments and terminals[Pepita99].

An important element for the universal access is the protocol stack. A protocol stack with hard coded functionality can not provide the current required degree of flexibility. Currently, 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. The stack has to be dynamically built up from components, based on the requirements and the momentary situation.

The first step in building the stack is to determine the component types to be used and the stack architecture that has to define the way building blocks are connected. In this document we report on how to describe protocol building blocks and how to build a stack from these descriptions.

---

This research was supported by the Flemish Institute for the advancement of scientific-technological research in industry (IWT) # 990219(Pepita)

## 1.2 Report overview

The following section describes the algorithm for stack composition. Section 3 outlines the building blocks of the stack composer. Section 4 lists the infrastructure support that is needed in order to support dynamic stack composition. Section 5 points the possibilities for enhancement and further research.

## 2. The algorithm for stack composition

### 2.1 Problem description

The goal is to automatically build a protocol stack from building blocks that are available as protocol components, so that the resulting stack is the best response to the current requirements imposed by the application and environment at the moment.

There are two distinct problems involved:

1. Identifying the complete information that is needed to describe each component so that it can be well deployed in the stack composition
2. Determining the policies by which appropriate components are to be integrated into the stack, so that application and context requirements are satisfied. Designing an algorithm for automatic stack composition based on these policies.

Recently, the construction of software systems and applications through composition of modular software is imposing as an easy and reliable technology. In the component technology, the units of packaging, distribution and deployment are the components. An 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. Also they must be able to react to the changes in their environment and dynamically reconfigure themselves to best fit the current conditions.

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.

The realization of our goal of automatic stack composition can benefit from experience accumulated in the previous and ongoing research in the general domain of dynamic configuration of component-based systems, but it raises some specific problems that have to be addressed in a more dedicated approach.

In our case, within the stack composition problem, it is not at all obvious from the start which component depends on which exact component list. One component (in our case a protocol building block) does explicitly depend only on a set of properties that has to be provided by the environment (including also other components). One problem in the stack 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. Anonymous dependencies, expressed indirectly by means of required/provided properties and established on the fly have to be taken into account. This leads to more complex composition algorithms.

## **2.2 Solution starting points**

This paragraph gives an overview of the most important strategy issues in the stack composition algorithm. Also it establishes first a common terminology used in the rest of this text.

### **2.2.1. Context and terminology**

#### **2.2.1.1. Layered stack architecture**

We start with the assumption that we use a *layered stack architecture* since this is till now the dominant architecture for protocol stacks. Each *layer* is a unit of modularity which provides a specific set of services which can be used by other layers on top of it.

The stack composition process will exploit the layer property of incremental enhancement of communication services. If the current layer does not provide enough functionality for a given application, the communication service may be enhanced step by step by adding other layers to the protocol stack.

#### **2.2.1.2. Components**

We use layers as the units of composition in the stack composition algorithm. These will be referred to as the protocol building blocks or *components*. We use the term components in the description of the composition algorithm since this is more general. Similar composition principles and rules may be used not only for protocol stack composition, but also in the context of a wider problem domain, the construction and automatic configuration of software systems through composition.

### 2.2.1.3. Component descriptions

Within the frame of the composition algorithm, only lightweight component representations are needed. These correspond to simplified *component descriptions* that include only that part of the component specification which is relevant during the decisional process. This comprises the functionality that can be provided by the component and the requirements that it imposes.

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.

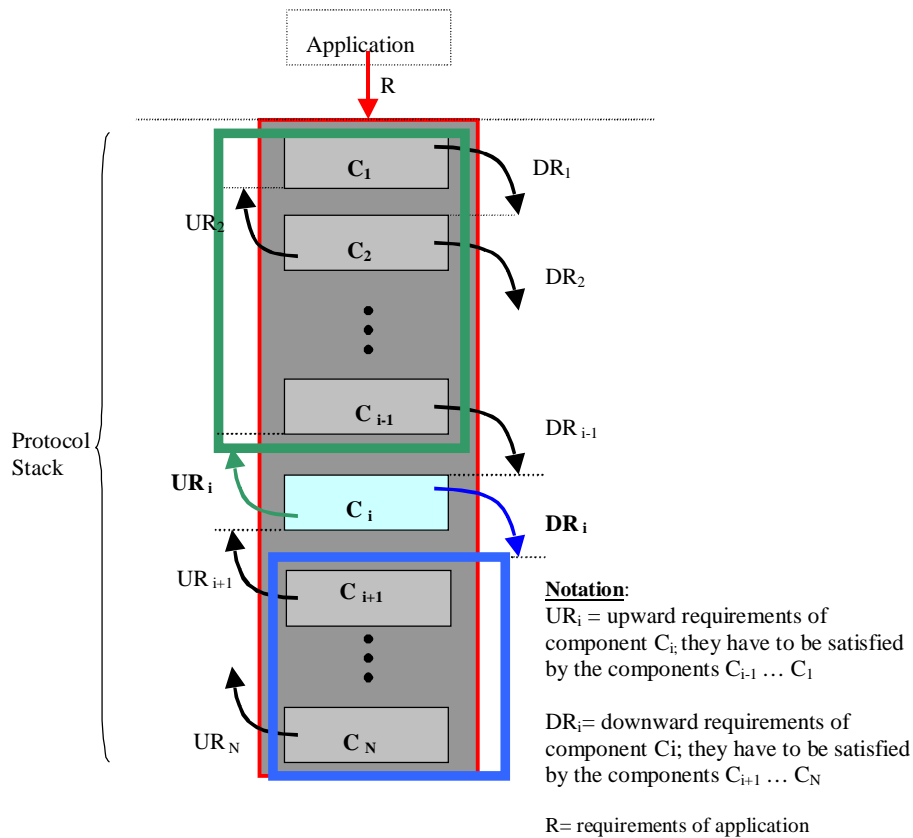
In order to be able to provide those properties, each component requires certain conditions to be satisfied by others. As a representation of the requirements of each component we use lists of *required properties*. The solving strategy of our composition algorithm is “matching requested properties with provided properties”.

We classify the requirements with regard to the direction of their target (target that is anonymous), and according to their strictness. 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).

We deploy the following types of requirements:

- *Strong downward requirements*, further called *downward requirements* in the algorithm description. This category of requirements is that which guides the solution searching for the right protocol stack, since in the layered stack architecture, a layer at level n makes use of services of layers at level n-1 (requests something to be provided by them) and offers its services to layers at level n+1 (provides services for the upper layers).
- *Weak upward requirements*: they are optional requirements, the compliance with this category of requirements is checked, but they are not used as a driving factor of the solution searching.
- *Strong upward requirements* are less important as the downward requirements for the process of searching (which is driven mainly by downward requirements).

By default, no order is assumed in meeting a list of required properties. An ordering preference may be specified, stating whether a property from a requirement list has to be met strictly before or after another property.



**Figure 1.** Upward and downward requirements of the components in a protocol stack

The components may be complex stack building blocks which can have different functionalities, depending on the context of their usage. To better handle this situation, a component is considered as being a set of *roles*. Each role groups a list of related provided properties, that imposes its requirements toward the environment. At each moment, the component may play a different role according to the context of its usage. The basic functionality of the component is defined by a so called “basic role”, while the context-dependent roles are described by a list of alternative roles. Usually, a different implementation will correspond to each different alternative role. Only one alternative role may be active at one time. The full behavior of the component is given by the basic role together with the active alternative role as an extension, like depicted in figure 2.

For example, SIP (Session Initiation Protocol) is a complex protocol with different context-dependent behaviour. It generally provides session initiation and requires to be used over a transport layer. But SIP may be used in different contexts (like as a SIP server or as a SIP UA). It also may require reliable transport or it may provide itself reliability. For this example, the component description of SIP uses 4 alternative roles like depicted in Figure 3. One alternative role (*rel\_sip\_server*) is active at one moment and defines the component behavior together with the basic role. Two of the alternatives roles of SIP have additional downward requirements (notation REQD).

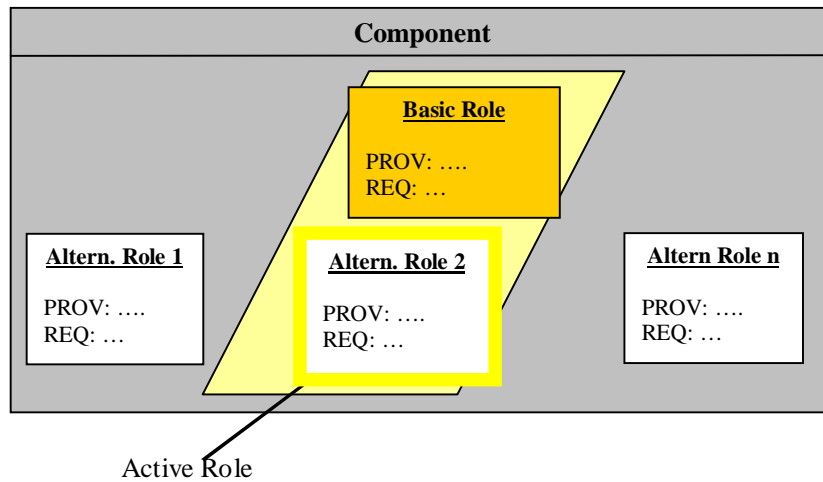


Figure 2. Roles of a component

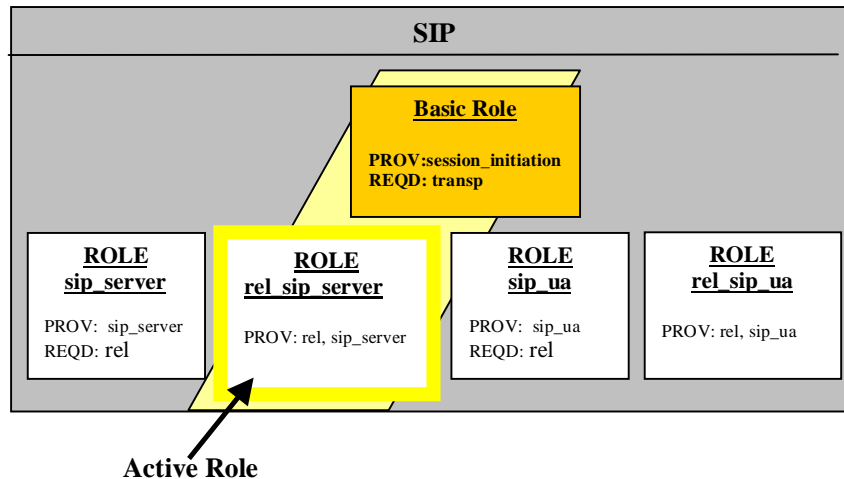


Figure 3. Example of component with multiple roles

#### 2.2.1.4. Defining a “good solution”

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

For example, if an application requests reliable transport, a suitable protocol stack can include an UDP and a REL layer (where REL is a special reliability layer). UDP provides unreliable transport, so the inclusion of the reliability layer into the protocol stack is necessary. Since there are no other requirements imposed by the application, the sequence REL UDP fulfills the first condition of the above list. If none of the two components, UDP and REL have other requirements, the second condition is also fulfilled. We have to observe that both UDP on REL and REL on UDP are valid solutions that comply with conditions 1 and 2. If one of these two solutions should be preferred, for example REL on UDP, then on these occasions an additional ordering restriction may be imposed by the user. If this is rather a permanent requirement then the description of component UDP must include reliability to its list of weak upward requirements. If UDP has reliability as a weak upward requirement, that means that a reliability layer is not strictly necessary above the UDP layer, but, if a reliability providing layer is to be added to the stack it is preferable to insert it on top of the UDP layer.

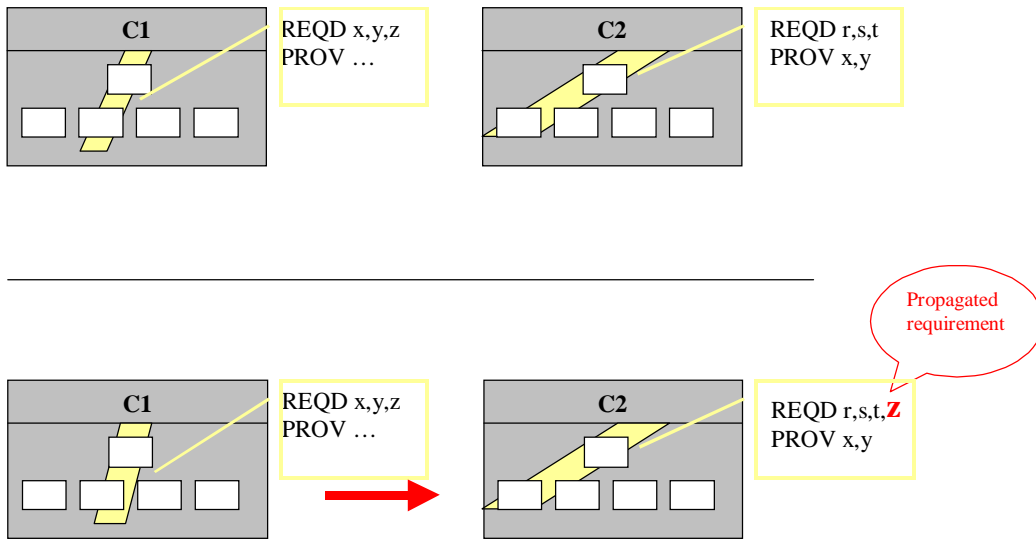
### 2.2.2. Solution searching

As said before, a simple “matching requested properties with provided properties” approach is used as a solving strategy.

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).

The starting point of the stack-building 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 in order to function well. The requirements of the application, that have not yet been accomplished, together with the own requirements of the partial-solution component, form a new problem for the stack 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.

We define the mechanism of *propagation of requirements* during the stack-finding process as an essential element in our strategy for stack composition. This is a mechanism that implements the principle of delegating the responsibility for solving certain requirements posed to a component, to other components, a principle which is naturally applicable to certain of the properties required by the components involved in the stack composition.



**Figure 4.** The mechanism of propagation of requirements

Figure 4 illustrates the mechanism of propagation of requirements by an example. If a component C2 provides some of the properties required by the active role of the component C1, and C2 is chosen to be a component part of the stack below C1, then all the properties that are required by C1 and are **not** provided by C2 are *propagated* to C2. 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.

## 2.3 Description of our stack composition algorithm

### 2.3.1 Structures used

The composition algorithm operates with component descriptions. A component description contains information regarding the roles of the component, with their provided and requested properties. As described in 2.2.1, the downward requirements are strong requirements and they drive the solution searching process, while the upward requirements may also be weak (optional), in the sense mentioned in 2.2.1, and used for validation of a solution.

The relations between components naturally form a graph structure. Each node is a component description with only one active role. The edges are dynamically updated during the solution-searching process. A directed edge from a node with a component description of  $C_i$  to a node of  $C_j$  exists when the active role of  $C_j$  provides at least one property that is requested as a downward requirement by the active role of  $C_i$ , either as its own or propagated requirement. Since new requirements appear through the mechanism of propagation, this implies the dynamic addition of new edges to the graph.

The application can be added as a special “component” and represented by a node in this graph. The application node has no incoming edges and will be referred to as *root-node*. The network interfaces are also special components, represented by special nodes with no outgoing edges (no downward requirements). Those nodes will be referred to as *leaf-nodes*.

A solution of the stack composition algorithm is a path from the root-node to one leaf-node, where at the end no propagated requirements are left and there is no conflict or contradiction with the upward requirements of the components involved along this path.

### 2.3.2. Algorithm details

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 components as nodes, while the edges represent the dependencies between them. A directed edge from the node with the component description of  $C_i$  in the active role  $RC_i$  to the node of  $C_j$  in the active role  $RC_j$  exists when the role  $RC_j$  provides at least one property that is requested as a downward requirement by  $RC_i$ , 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.

A modified Breadth-First Search-like [GoTa] algorithm has been used as backbone for the traversal of the graph in search for the solutions.

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.

An important observation is that the quality of the solutions produced by the stack composition algorithm is strongly conditioned by the accuracy of the specifications of the components descriptions, since that determines the graph-structure. The specification for each protocol building block should be done by the protocol designer.

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.

### 2.3.3. Algorithm implementation

A queue is used for holding the nodes that have been already examined but are still subject to further processing. In this queue, the visited nodes are stored with their momentary state. The state is given by the path coming from the root by which this component has been attained. This defines also the current propagated requirements for the component.

While the searching process, it is possible for a component description to have multiple presences in the queue, since it is possible that it is attained by going on different paths.

The pseudo-code for the algorithm is shown below.

```
Algorithm Search ( StartNode )

• do specific actions for search initialization
• initializes CurrentPathToStart <- void
• initializes Queue with ( StartNode, CurrentPathToStart )

while ( ! ConditionStopSearching() ) do
• dequeue ( CurrentNode, CurrentPathToStart ) from Queue
• solve upward requirements
• if ( CurrentNode is a leaf-node ) then
• solve the strong upward requirements of CurrentNode
• a path to a leaf-node has been found -> do strategy
  specific actions
• determine list UnresolvedRequirements along the
  CurrentPathToStart
• if ( list UnresolvedRequirements contains no properties )
  then
• CurrentPathToStart is a possible stack
• verify upward requirements along CurrentPathToStart
• do strategy specific actions if a good solution has
  been found
• else
• computes LPA = ListOfAdjacencies ( CurrentNode )
• for ( each node CurrentPANode in LPA ) do
• if ( CurrentPANode is not already in
  CurrentPathToStart ) then
• CurrentPathToStart <- CurrentPathToStart +
  { CurrentPANode }
• Propagate requirements from CurrentNode to
  CurrentPANode ( modify graph structure )
• enqueue ( CurrentPANode, CurrentPathToStart )
```

The start node for the algorithm is the root-node (corresponding to the application). The end condition may be set in function of the desired strategy. The search process may end after the finding of the first solution, the finding of a certain number of solutions, or when no other solutions may be constructed. Also when a path to a leaf node has been found (not necessarily a solution that satisfies all requirements), different actions may be performed, whether the current used strategy is interested in partial solutions or not.

In order to get optimized solutions, in terms of using only the necessary components, minimal stacks that correspond to the requirements should be proposed. Adding semantic redundant components has to be avoided (like UDP on TCP or REL on TCP). External ordering preferences have also to be taken into account. In order to achieve this, the initial adjacencies list is reduced by applying exclusion criteria like: semantic redundancy (don't add components that repeat some of the properties already provided) and good ordering.

## 2.4 Discussion of algorithm on examples

We use a simple case study for illustrating the algorithm used to determine the configuration of a stack that corresponds to a set of application and environment requirements. In our example there are also ordering preferences and components with multiple alternative roles involved.

We consider the requirements : session initiation ( in the context of a SIP server) and non local communication. As ordering preferences, transport should be provided above non local and reliability above or in the same layer with transport.

Starting from those application requirements, 2 components are found to partially fulfill the requirements: IP (provides non\_local), and SIP with 2 of its roles (SIP as rel\_sip\_server and sip\_server provides session\_initiation and sip\_server). The situation is depicted in figure 5, together with the propagated requirements.

If IP is considered as a possible top of the stack, the propagated requirements session\_initiation and sip\_server have to be fulfilled, which is possible by using either the sip\_server or rel\_sip\_server role of SIP component (figure 6). However, after this addition of SIP, the downward requirement for transport appears, which is not possible to be solved, due to the ordering preference that states transp should be above non\_local. Those paths will be abandoned.

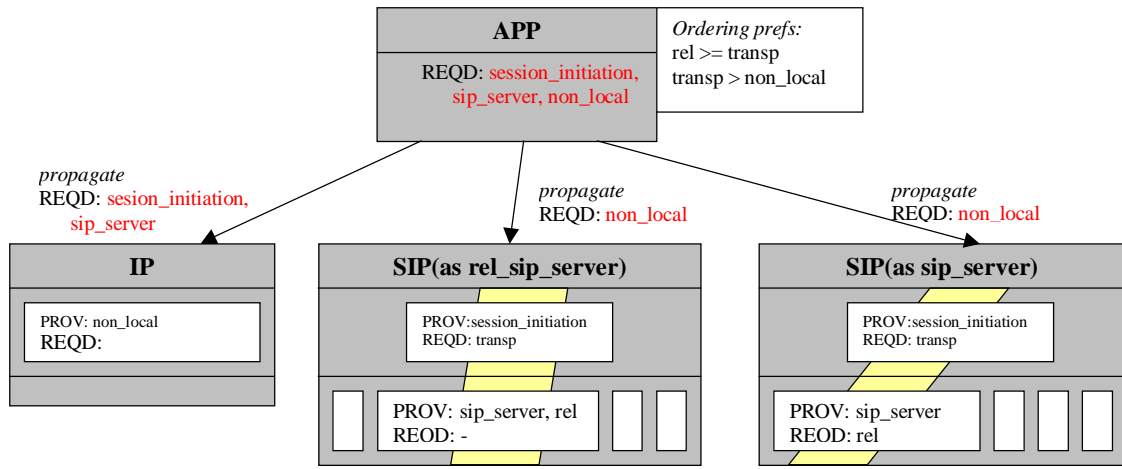


Figure 5. Example: step in finding a protocol stack composition according to client-specific requirements

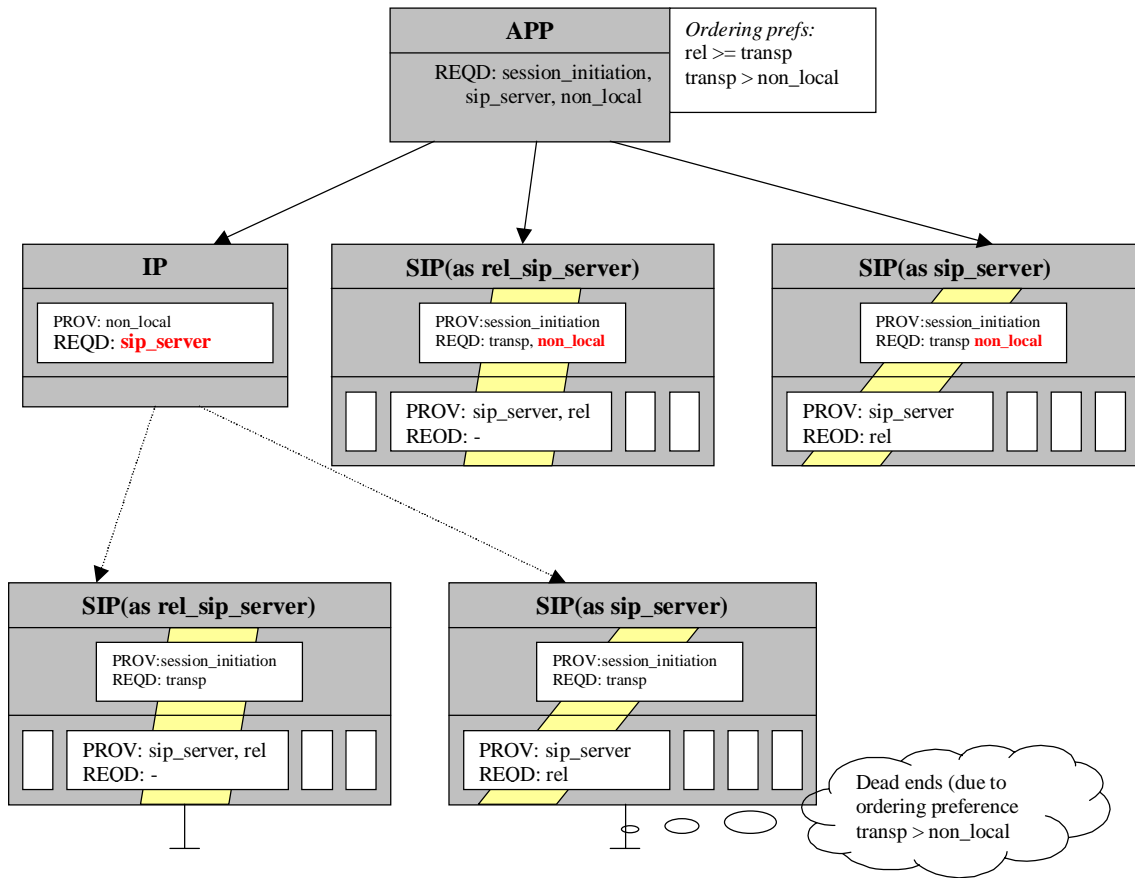
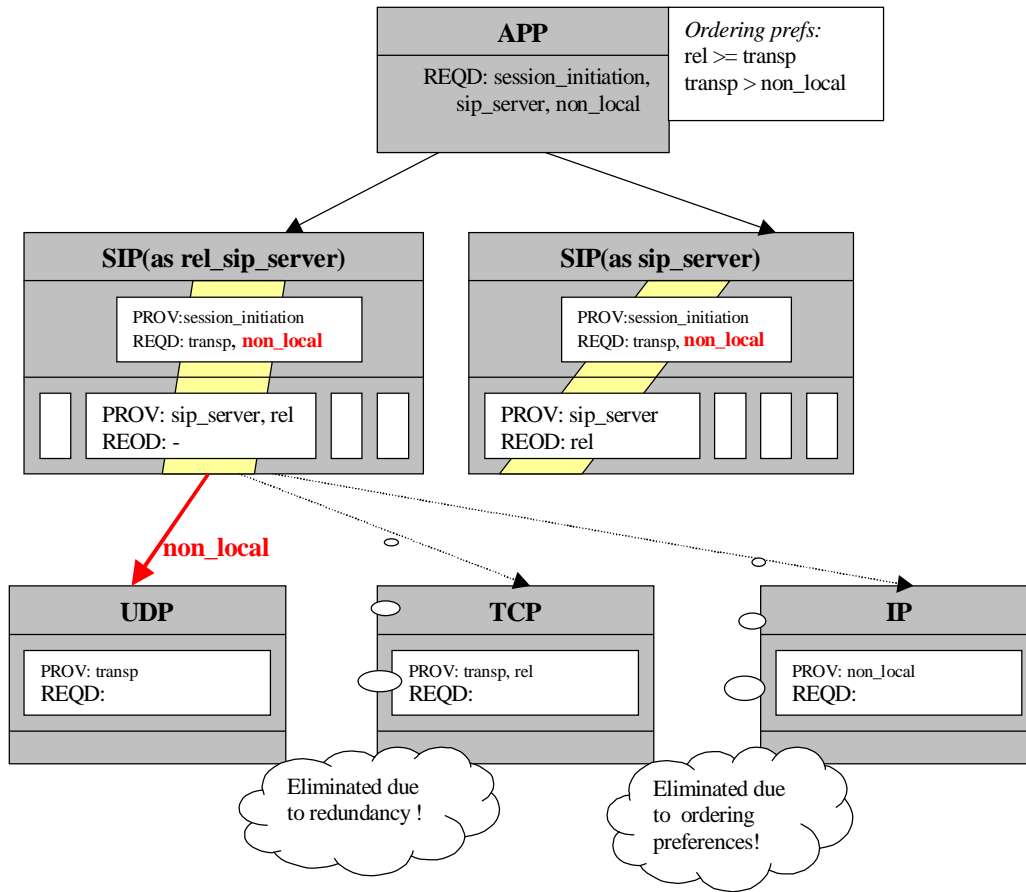
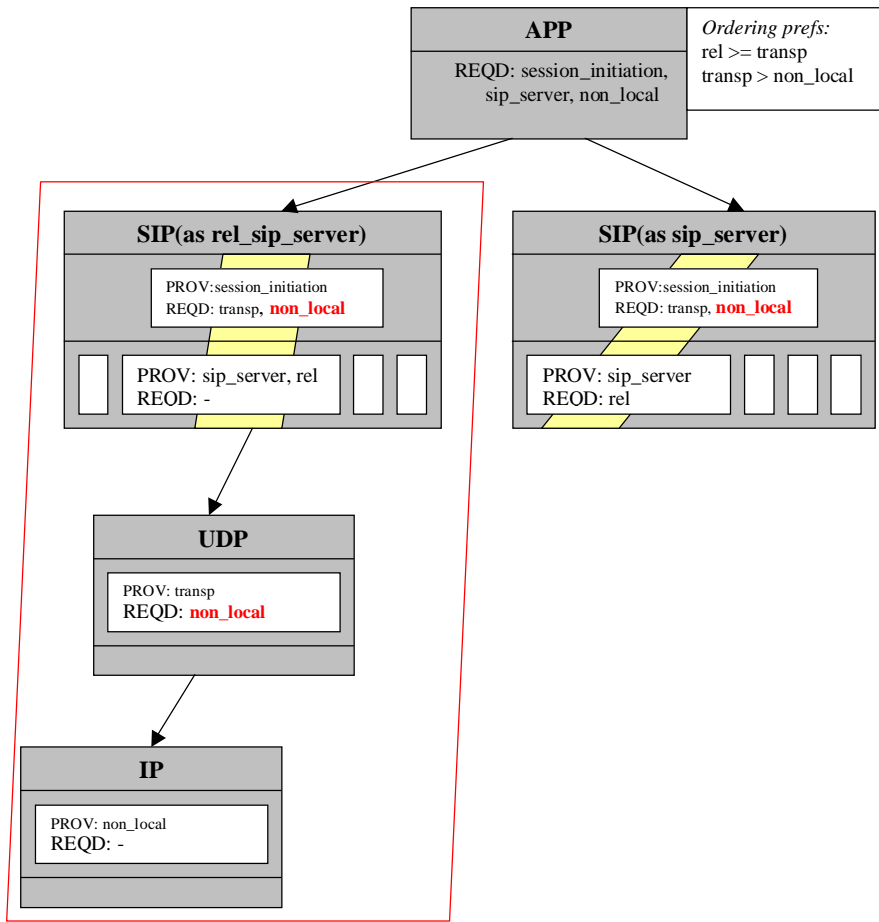


Figure 6 Example: step in finding a protocol stack composition according to client-specific requirements



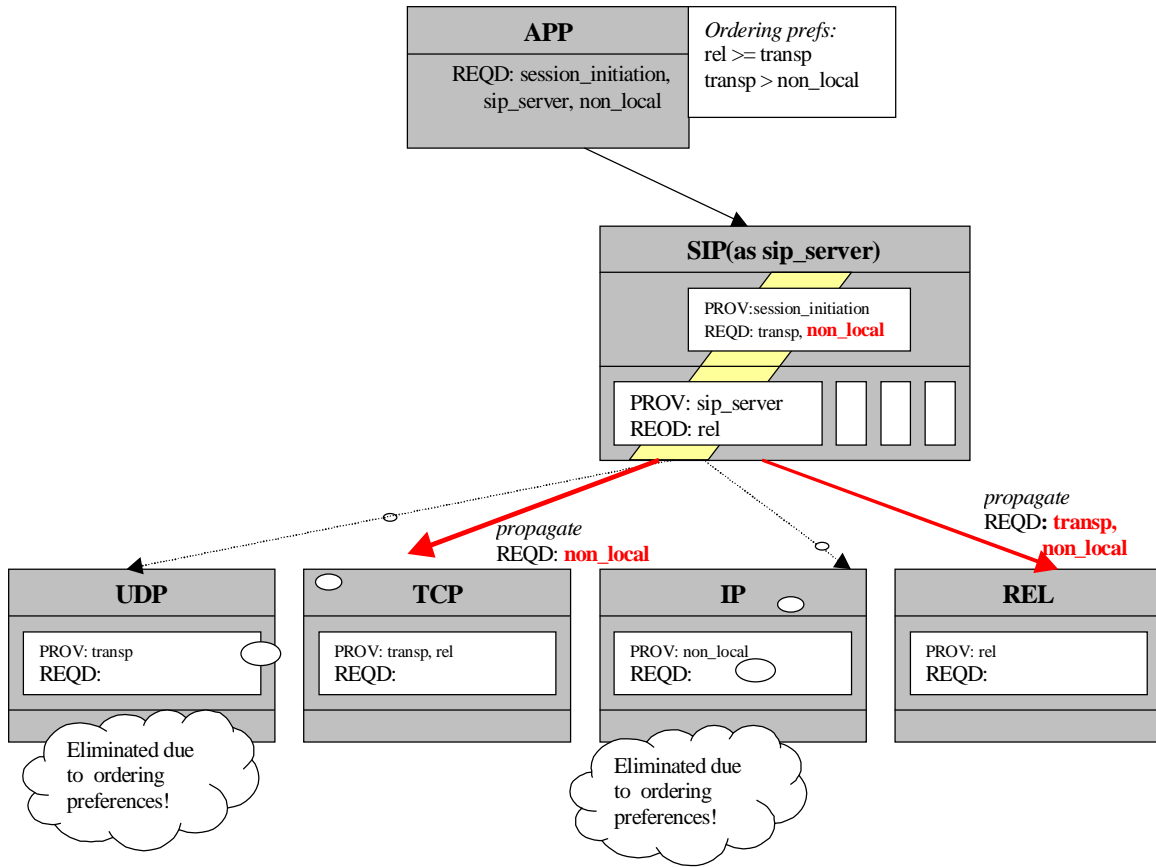
**Figure 7** Example: step in finding a protocol stack composition according to client-specific requirements

The next possibility investigated will be to use SIP in the role of a `rel_sip_server` as a stack top, like depicted in figure 7. The cumulated requirements (own and propagated) are `transp` and `non_local`. Possible successors (components that provide something) are UDP, TCP and IP. TCP is eliminated due to redundancy (it provides `rel`, which is a property already provided by SIP in its current active role). IP is eliminated due to ordering preferences which states that `transp` should be above `non_local`. The only valid successor of SIP as `rel_sip_server` remains UDP.



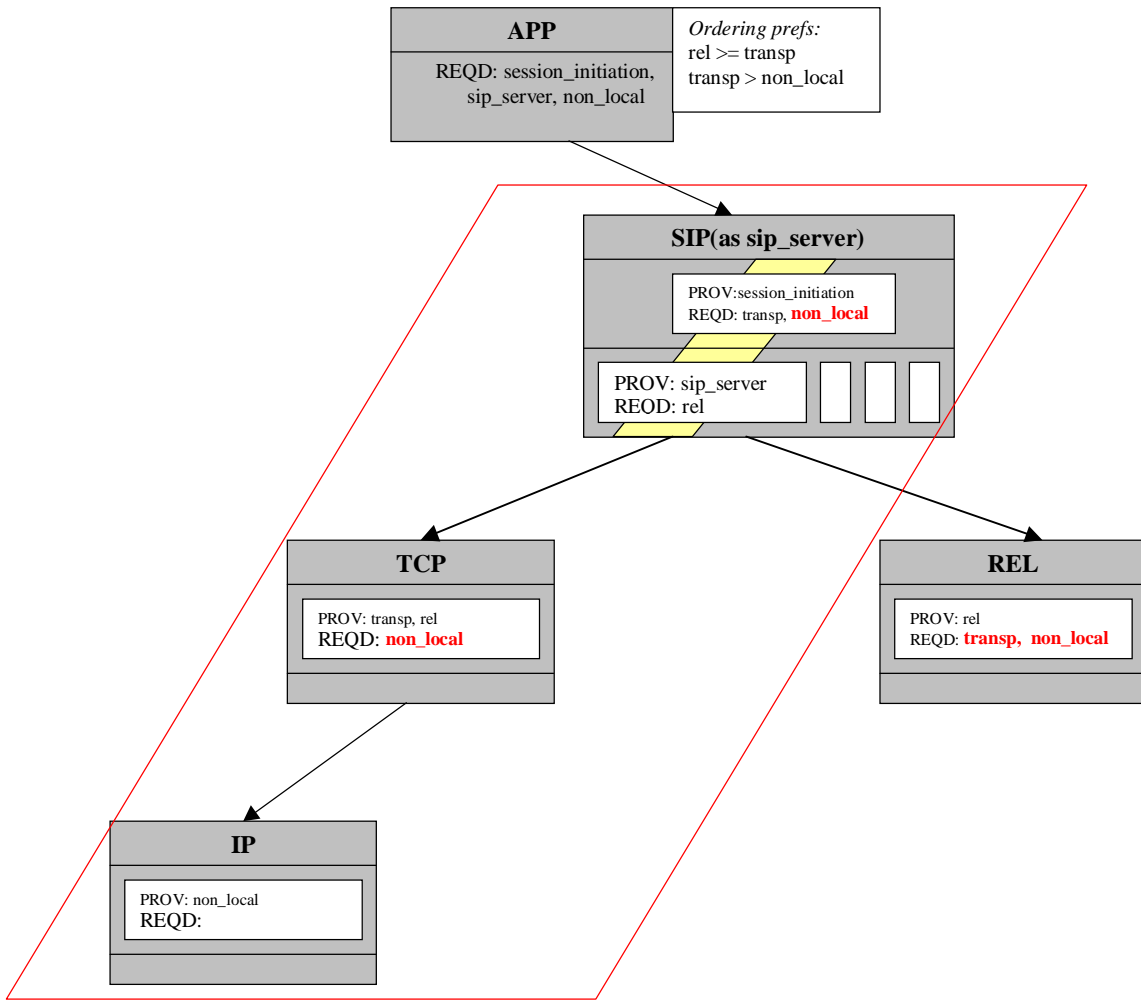
**Figure 8** Example: step in finding a protocol stack composition according to client-specific requirements

To UDP, the requirement `non_local` has been propagated and IP is found as a good successor. No other requirements are left, so a good solution has been found in the stack SIP as `rel_sip_server` on UDP on IP. The searching may stop here or proceed further to look for other solutions as well.



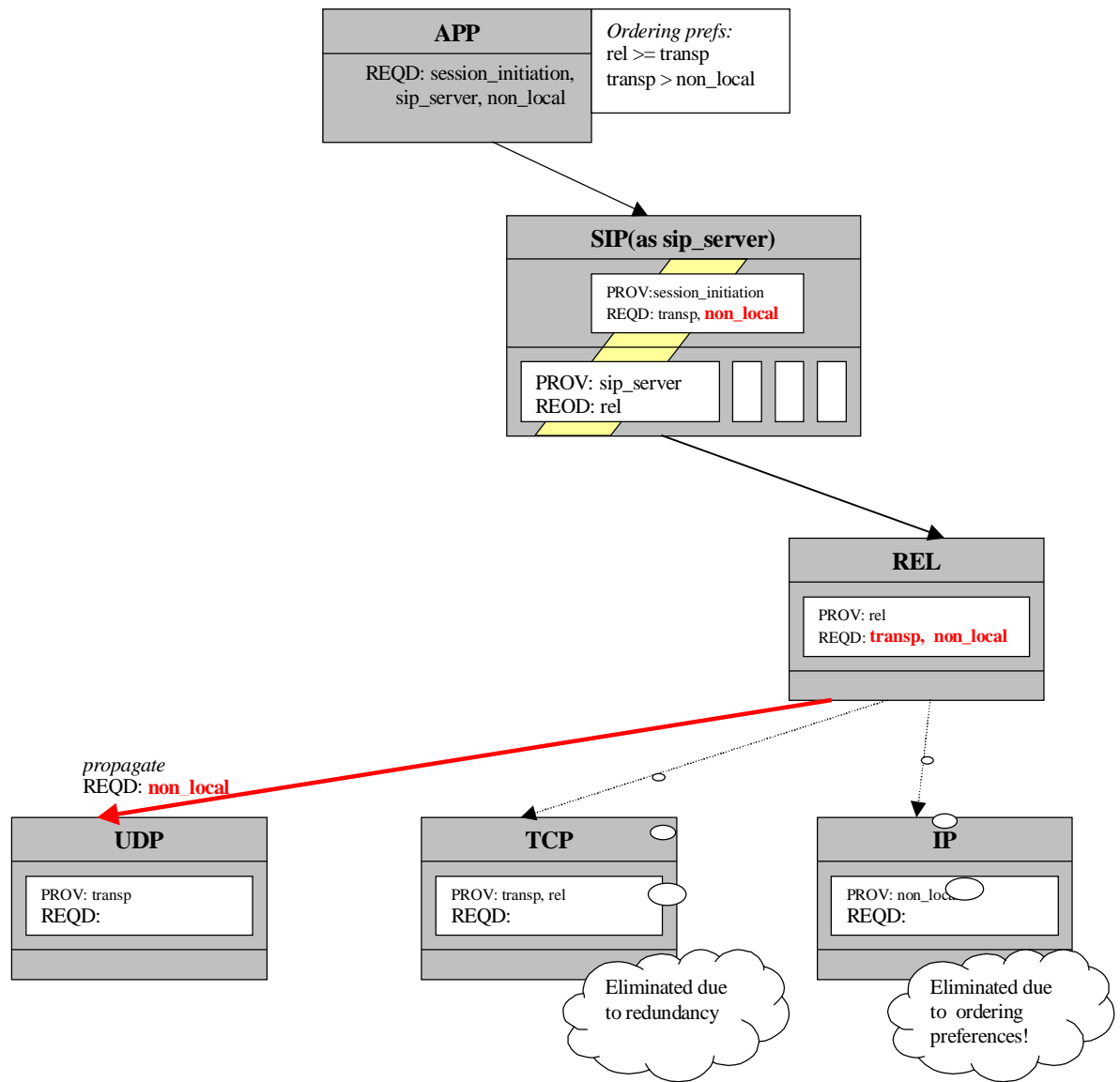
**Figure 9** Example: step in finding a protocol stack composition according to client-specific requirements

The next possibility investigated will be to use SIP in the role of a sip\_server as a stack top, like depicted in figure 9. The cumulated requirements (own and propagated) are rel, transp and non\_local. Possible successors (components that provide something) are UDP, TCP, IP and REL. UDP is eliminated due to the ordering preference that states rel should be above transp. IP is eliminated due to the ordering preference which states that transp should be above non\_local. Valid potential successors stay TCP and REL.



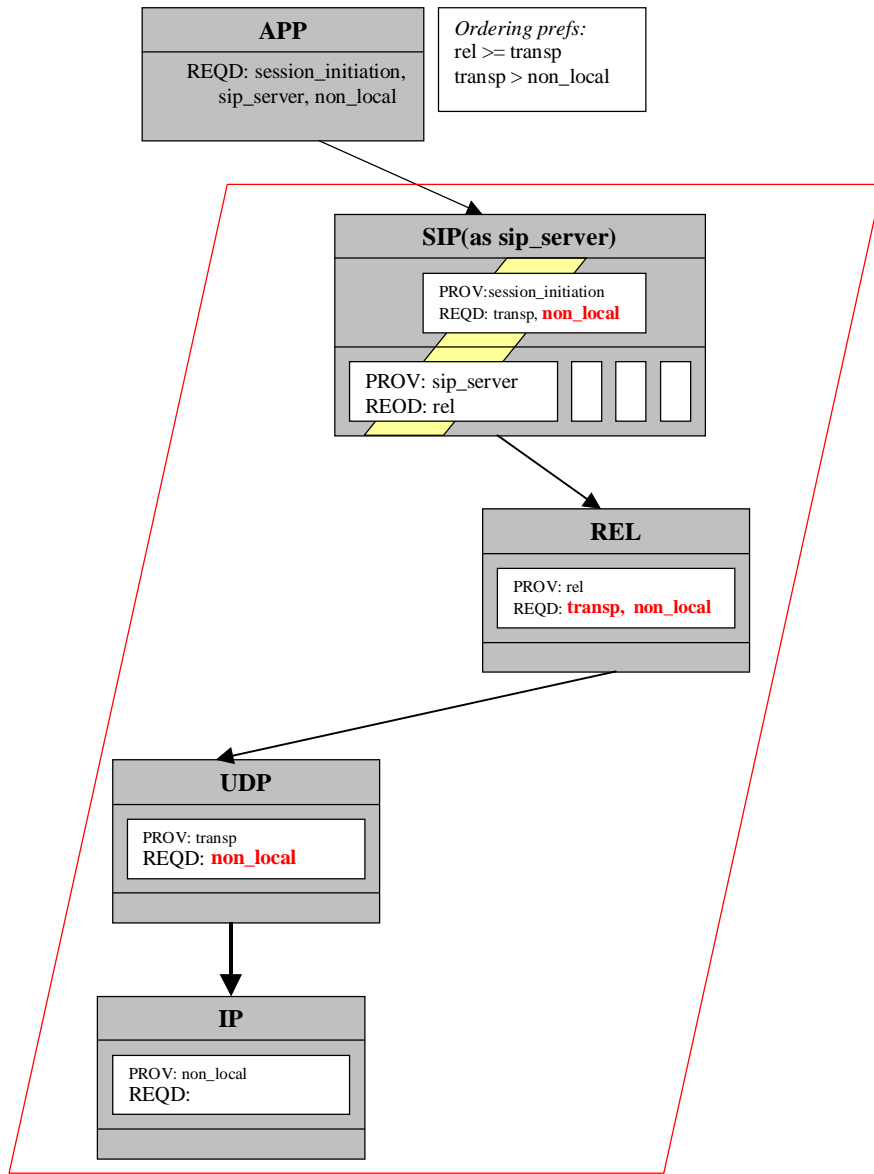
**Figure 10** Example: step in finding a protocol stack composition according to client-specific requirements

If TCP is chosen as a successor of SIP as sip\_server, the only requirement that remains is non\_local, and it will be fulfilled by adding IP. Another solution has been found, as the stack with the configuration SIP (as sip\_server) on TCP on IP.



**Figure 11** Example: step in finding a protocol stack composition according to client-specific requirements

If REL is chosen as a successor of SIP as sip\_server, the requirements that remain are transp and non\_local. Potential successors of REL are UDP, TCP and IP. TCP is eliminated due to redundancy (provides reliability, which is already present in rel\_sip\_server). IP is eliminated due to ordering preferences (non\_local should be under transp). UDP remains the only valid successor.



**Figure 12** Example: step in finding a protocol stack composition according to client-specific requirements

UDP is chosen as a successor of SIP as sip\_server, the only requirement that remains is non\_local, and it will be fulfilled by adding IP. Another solution has been found, as the stack with the configuration SIP (as sip\_server) on REL on UDP on IP.

### 3 Design of the stack composition package

#### 3.1 Overview of core architecture

We now give a short overview of the building blocks of the architecture of the stack composer. This overview includes also the interactions with the additional infrastructural support that is needed and will be described in section 4. The role of the stack composer package is to automatically provide a list of potential stacks that respond to the given application requirements and that are built using components from a component repository. A stack composer package needs access to a component repository that must be able to be interrogated for component descriptions. An effective stack-builder is also needed, as part of the infrastructural support of the framework for stack composition.

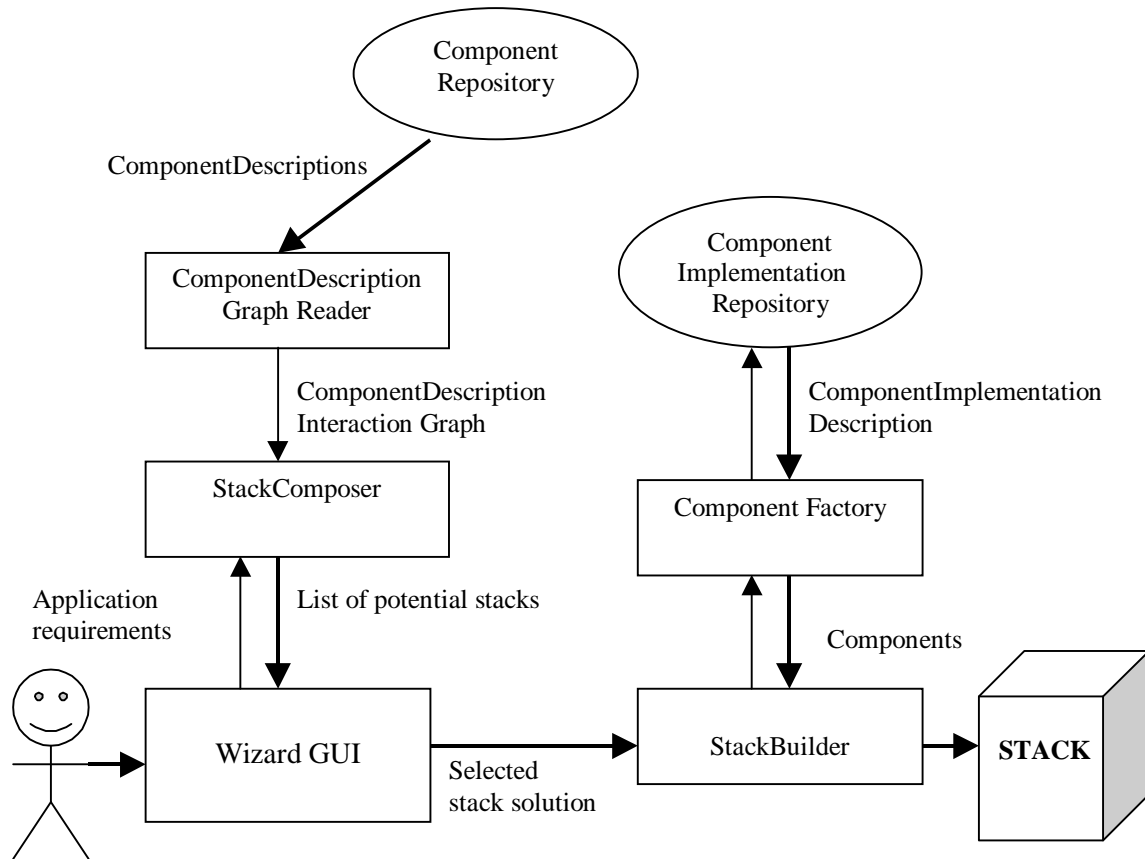


Figure 13. Overview of the architectural building blocks

**Component repository.** A component repository must be part of the system and should be accessible for interrogation to the stack composer. The stack composer requires that descriptions of all protocol building blocks (components) may be obtained from this repository. The description of a component must specify its own properties as well as requirements towards other components.

**ComponentDescription-Graph Reader.** The nature of the component repository may be diverse. The role of this Reader is to provide, in a repository-independent manner, a list of the components (actually the component descriptions) and the interdependencies between them.

**StackComposer.** This is the main part, where the actual composition of the stack is determined. Different strategies of solution searching may be deployed as composition algorithms by the stack composer. The input for the stack composer consists of the application requirements and an initial graph of component descriptions. A list of potential stacks is produced. A potential stack, as produced by the Composer, is a sequence of component descriptions.

**StackBuilder.** Part of the framework for dynamic protocol composition. A protocol stack is built based on a potential stack produced by the Composer in form of a sequence of component descriptions.

**ComponentFactory.** During stack building, the necessary components are instantiated through a Component Factory that uses the Implementation Repository.

**Implementation Repository.** For each component found in the Component Repository, several implementations may be available, which have different implementation characteristics. Descriptions of the available implementations are contained in the Implementation Repository.

**WizardGUI.** A wizard-like user-interface may be used at the junction of the Composer with the effective StackBuilder, for interactive selection of one stack out of several automatically detected candidates.

## 3.2 Description of stack composition modules

### 3.2.1 Classes used for component description

A component description, as used in the stack composition, is a partial specification of a component, including only that part of the component specification information which is relevant during the decisional process of the stack composition. This information comprises the functionality that can be provided by the component and the requirements that it imposes towards other components, in its different roles, as well as the type of component (layer component or network interface).

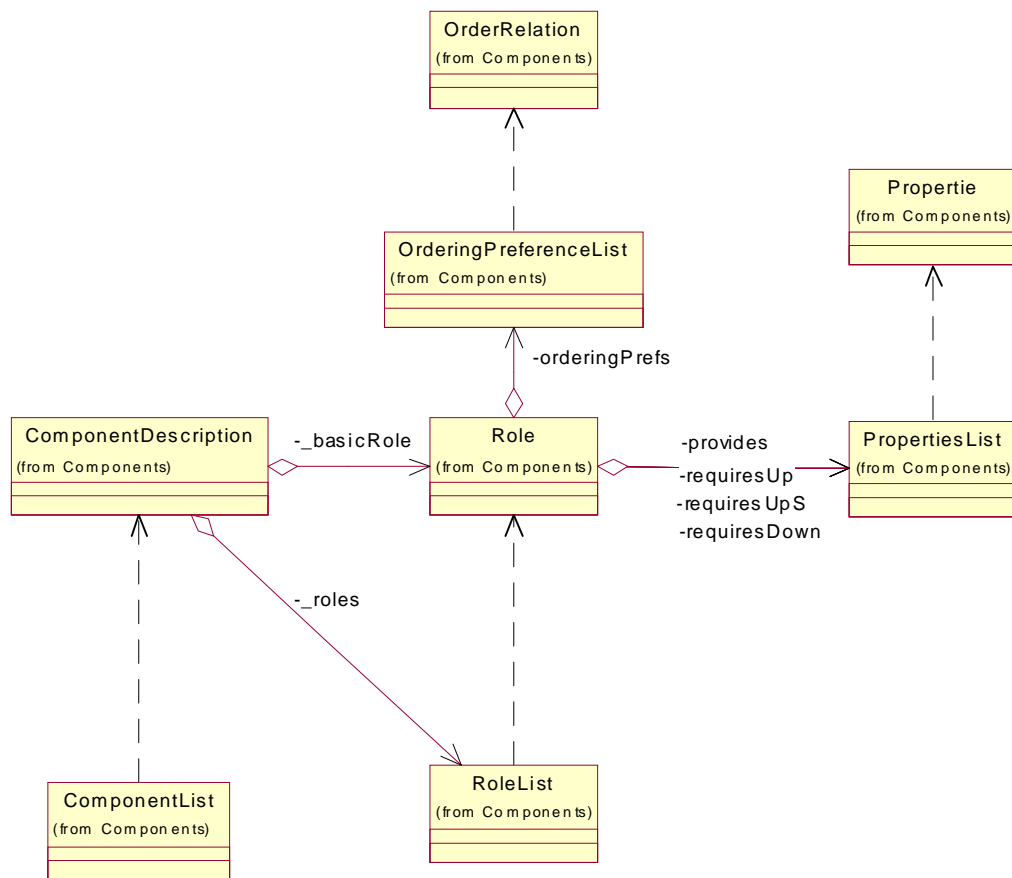
ComponentDescription is the abstract way for describing a protocol building block. Objects of ComponentDescription are not instances of that component type. Those objects are only representations of a subset of the properties of that component type, subset that is relevant for the stack composer. For creating the real component (protocol stack layer) corresponding to a ComponentDescription, a ComponentFactory has to be used.

A component description may have many roles, like discussed in 2.2.1.3. A role specifies a set of provided properties and the sets of corresponding requirements (upward and downward) that have to be satisfied for that role. Also a profile may be active or not. A component has one basic role, which defines the main properties, and may have several alternative roles for defining complementary or optional properties. The basic role is always active and only one of the alternative roles, if these exist, is active.

The provided properties and also the requirements are expressed as lists of properties. A property is a rather abstract feature. In our current implementation a property is a name. No differentiation is made about semantically different categories of properties. A requirement is satisfied by a property if there exists a name-matching between them.

Class ComponentDescription implements a lightweight component representation within the frame of the composition algorithm. It also implements operation for:

- activating a role
- testing the presence of a specified property among the provided properties of this component
- testing the presence of a set of provided properties among the active role of this component
- testing if a property is a requirement of the active role of this component
- testing if the component requires some properties provided by another component
- testing lack of semantic redundancy of this component with a list of other components
- propagating requirements from a specified component to this



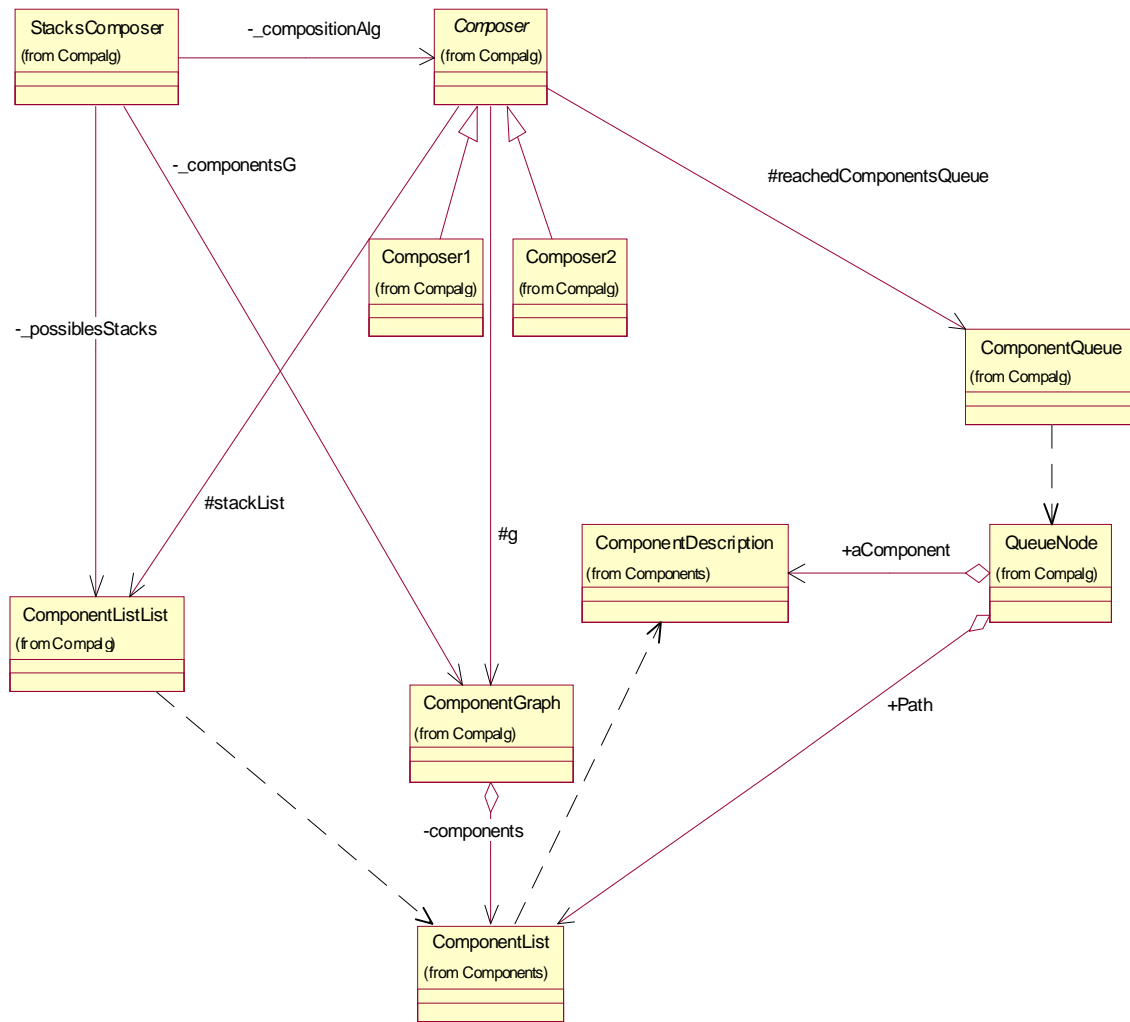
**Figure 14.** Classes used for component description

### 3.2.2 Classes used for stack composition

The stack composition algorithm uses a graph structure for representing the dependence relations between the components.

The ComponentGraph has a ComponentList (see figure 15) that stores the graph nodes. The StackComposer determines the list of all possible stacks that could be built from the components that are in a component graph which defines their dependencies. Different strategies for composition algorithms may be used, by employing a different Composer. The algorithm of stack composition is run by the go operation of the composer. The basic mechanism of the composition algorithm has been described in 2.3.3. A BFS-like algorithm is used as a backbone for the traversal of the graph in search for solutions. On this common pattern, the user may control what strategy to use with regard to whether accepting or not a component sequence as a good solution (dealing with partially fulfilled requirements), including or eliminating some searching criteria (redundancy, strong interpretation of requirements, trying to cope with insufficient requirements). The variable strategy elements are defined by the operations actionsInitSearching(), actionArrivedLeaf(), actionGoodSolution(), buildListPartialAdjacencies(). Specific Composer may implement differently these methods.

The implementation of the algorithm uses a queue for storing partial results during the unfolding of the solution searching. A QueueNode contains the component and a component list representing the succession of components that precede it on a possible solution path.



**Figure 15.** Classes used for stack composition

## 4 Description of needed infrastructure support

Besides the policies for dynamic stack composition (comprising the component specifications and composition algorithm), also specific infrastructure support is needed for achieving the dynamic stack building. The composition algorithm describes a sequence of components. Infrastructure support is needed in order to build the protocol stack that corresponds to this sequence of component descriptions.

The needed infrastructure support 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 builds 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

## 4.1 DiPS Stack Composition Framework (SCF)

Support at infrastructure level is needed for really building a stack that suits a specific purpose. The first requirement to support building a stack that suits a specific purpose, is infrastructure support. This includes framework support for the necessary plumbing of layers and components, and run-time support for the dynamic behaviour.

Having a way to compose stacks with selected functionality and non-functional properties is not enough. The protocols a stack is composed of also have an impact on the addressing used inside the stack. Since addresses are used by application programs, the impact of modifying a stack is not automatically transparent to the application. In addition, addresses have to be carefully managed in a world with dynamically changing protocol stacks. For example, adding a reliability layer to an existing UDP/IP/Ethernet stack should be possible, but this may impact the addresses used in the stack. AddressManagement is responsible for hiding as much as possible of the impact from the applications.

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, that has been extended with a Stack Composition Framework (SCF).

A DiPS stack has several properties that have to be constructed: the Structure property, the SCF property and the interface configuration property.

- The structure property: defines the structure of a stack. It knows about layers, glue, interfaces and socket types of a stack. This property should always be present.
- The ScfProperty: sets up all SCF-related items. Most importantly, it installs a StackAddressManager and it adds LayerAddressManagers. Only present for SCF stacks.
- The IfConfigProperty: handles interface configuration. It basically reads configuration info and uses the standard DiPS Configurator system to configure the interface.

The framework class which allows a complete stack to be built dynamically is the StackBuilder [Mat99]. It can interpret declarative statements and construct a stack, interpretation can be done at run-time. This only covers the mechanisms for building stacks (and in a similar manner also layers) at run-time. A complete adaptive system needs, besides this mechanism, a policies-based mechanism which can decide when a protocol stack change is necessary, based on information about varying requirements from the environment (network condition, application) and also determine the configuration of the stack that would best fit the situation.

A StackBuilder should provide the following capabilities:

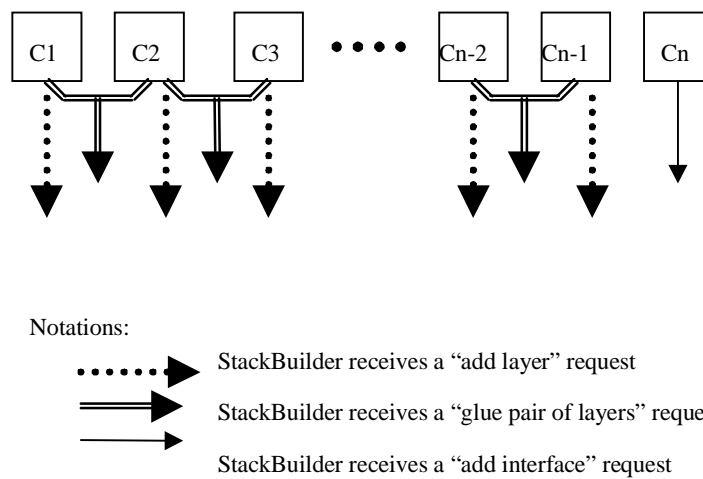
- Capability of adding a new layer to the stack. The StackBuilder should be able to receive an "add layer" type of request and respond to it by adding the new layer, which has to be specified in the request, to the stack.
- Capability of connecting two layers in the stack, specifying the upper and lower layer. The concept of "glue" is used for the connection of two layers. Different types of "glue" have to be deployed, depending on the type of the connected layers. The

StackBuilder should be able to receive a “glue layer1 on top of layer2” type of request and respond to it by connecting layer1 on top of layer2, with help of the specific glue, which has to be specified in the request.

- Capability of introducing a network interface to the stack, specifying also the layer the interface belongs to. The interfaces should be able to be later configured using appropriate configuration information that depends on the specific interface and the protocols that are used on top of it. The StackBuilder should be able to receive “add interface” and “configure interface” type of requests.

Each StackBuilder will be responsible to build one protocol stack. An empty stack is created initially and with help of the above mentioned capabilities that each StackBuilder should provide, one can introduce, at any time, layers, glue or interfaces, and modify the current configuration of the stack.

Our stack composer decides the sequence of layers of the stack that would correspond best to a current situation, finding potential stacks in form of sequences of component descriptions. The presence of the above mentioned capabilities of a StackBuilder is essential for transforming the input given as a sequence of component descriptions into the corresponding protocol stack.



**Figure 16.** Transforming a sequence of component description into a structure property of a protocol stack, by making use of StackBuilder capabilities

Figure 16 shows the principles for transforming a sequence of component descriptions into a protocol stack, by making use of StackBuilder capabilities.

For each component description of the sequence, a layer has to be generated, using the StackBuilders capability of adding a new layer. Exceptions make only component descriptions of network interfaces, for those an interface is generated through use of StackBuilder interface adding capability. Each pair of consecutive components found in the given succession has to be connected by a specific “glue”.

Integrating the composition module into DIPS requires mainly a modified StructureProperty class, that uses the StackBuilder in order to translate a sequence of

component descriptions into a stack structure. Also a modified ScfProperty, that is able to add the needed address managers for the given component sequence that forms a stack structure, is needed(see figure 17).

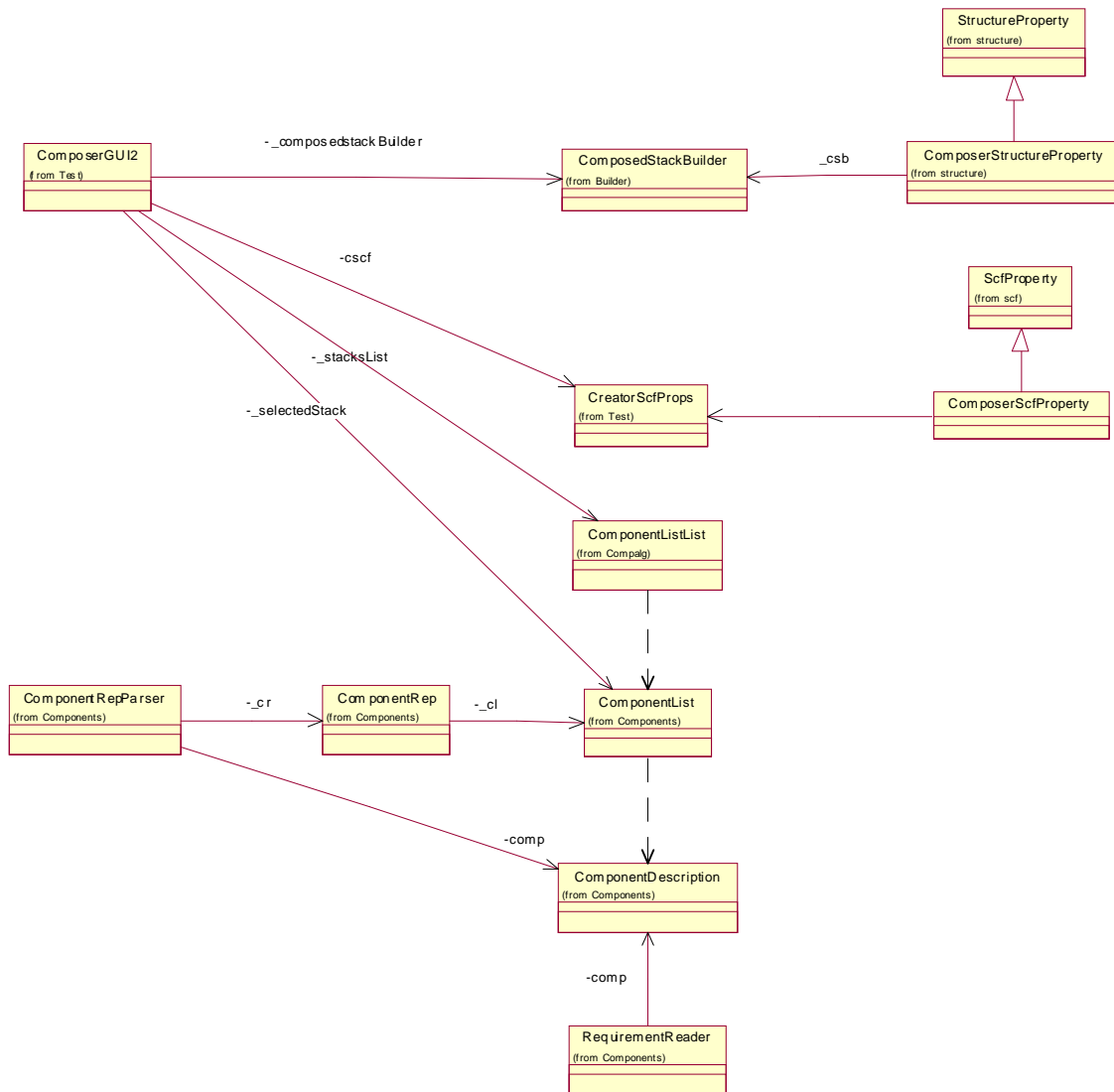


Figure 17. Integrating the composition module with DiPS

## 4.2 Component repository

We distinguish the

- Component Repository
- Implementation Repository

The Component Repository contains component descriptions and their roles.

Each component repository entry is a component description which contains information about the functionality provided by the component and its requirements. Figure 18 gives an example of the information that has to be contained in a component repository. An Implementation Repository Reader is able to cope with different or changing formats of the component repository.

```
Reliability
  REQD  REQUW  REQUS  PROV  rel  END

udp
  REQD  REQUW  REQUS  PROV  transp  END

ip
  REQD  REQUW  REQUS  PROV  non_local  END

tcp
  REQD  REQUW  REQUS  PROV  rel  transp  END

sip
  REQD  transp  REQUW  REQUS  PROV  session_initiation
  ROLE  rel_sip_server
        REQD  REQUW  REQUS  PROV  sip_server  rel  ENDROLE
  ROLE  sip_server
        REQD  rel  REQUW  REQUS  PROV  sip_server  ENDROLE
  ROLE  rel_sip_ua
        REQD  REQUW  REQUS  PROV  sip_ua  rel  ENDROLE
  ROLE  sip_ua
        REQD  rel  REQUW  REQUS  PROV  sip_ua  ENDROLE
  END

ethernet
  REQD  REQUW  REQUS  PROV  eth  END

STOP  veth0
  REQD  REQUW  REQUS  eth  PROV  END
```

**Figure 18.** Information contained in a Component repository – example excerpt

The Implementation Repository specifies where to find implementations of each component-role. The implementation may be specified in form of a java class-file or as the name of a layer-description file (an xml file containing the description of a layer). Several implementations with different implementation characteristics (high performance, low memory footprint, etc.) may be available for the same component-role.

```

IpImplem1 IMPLEMENTS ip
    IMLEM_CHARACTER high_performance
    CLASS dips.protocol.ip.ipv4.IPv4FastCore
    XML scfipv4perflayer.xml
    END

IpImplem2 IMPLEMENTS ip
    IMLEM_CHARACTER
    CLASS dips.protocol.ip.ipv4.IPv4Core
    XML scfipv4layer.xml
    END

SipServerImpl IMPLEMENTS sip AS sip_server
    IMLEM_CHARACTER
    CLASS dips.protocol.sip.SipServer
    XML scfsipserver.xml
    END

SipRelServerImpl IMPLEMENTS sip AS rel_sip_server
    IMLEM_CHARACTER
    CLASS dips.protocol.sip.SipRelServer
    XML scfsiprelserver.xml
    END

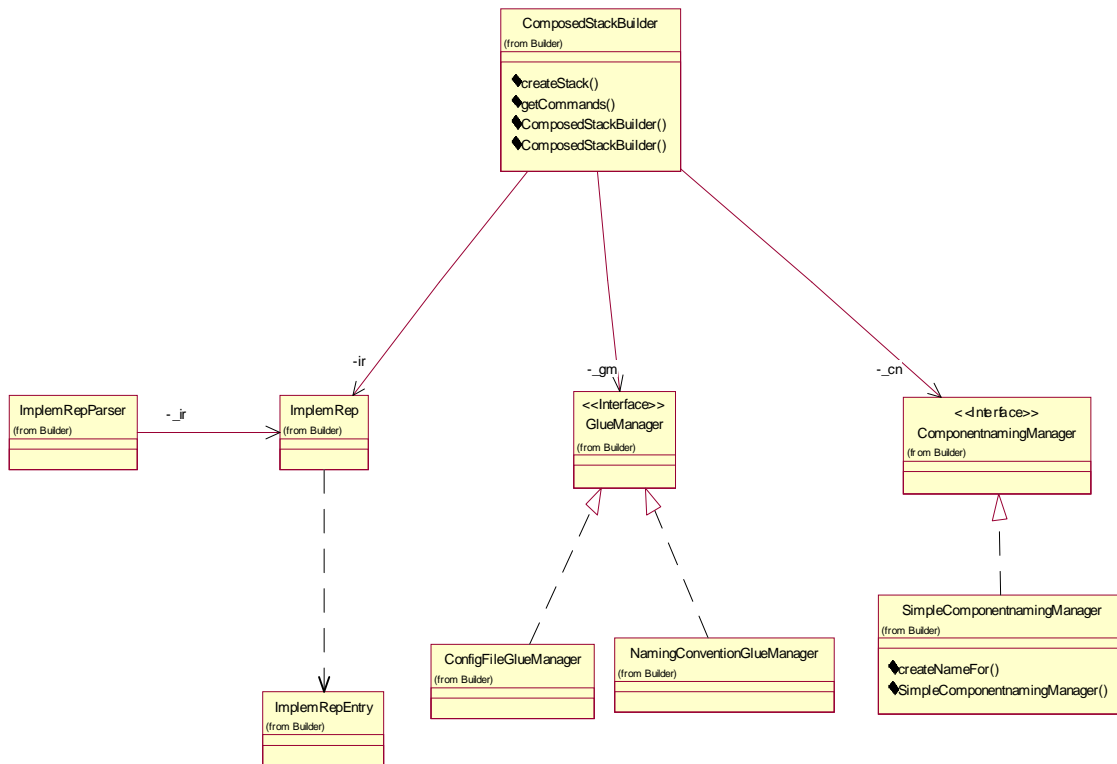
SipUAImpl IMPLEMENTS sip AS sip_ua
    IMLEM_CHARACTER
    CLASS dips.protocol.sip.SipUA
    XML scfsipua.xml
    END

```

**Figure 19.** Information contained in a Implementation Repository –example excerpt

### 4.3. Component deployment

The problems of instantiating the components and finding the corresponding glue will be solved through a Component Factory that relies on the Implementation Repository and a Glue Manager. An InstanceNamingService is used for creating unique names for the layers that are created with help of the ComponentFactory during the stack building process.



**Figure 20.** Infrastructure support needed for component deployment in stack composition

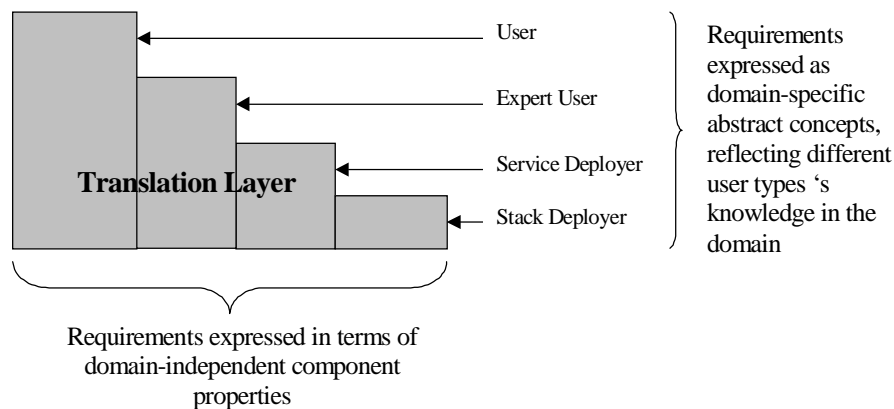
The Component Factory uses a factory method to create the adequate components according to the component descriptions. It interrogates a Implementation Repository in order to find the corresponding implementation for a component description. When multiple implementations are found for the same component and role, it uses additional decision criteria (memory footprint, performance, etc).

The Glue Manager has the role of finding out the correct glue type that has to be used between two components. The current implementation of GlueManager gives a simple solution based on a naming convention: the classname of the glue is derived from the names of the two components that are the layers to be connected. If N1 is the name of the upperlayer and N2 the name of the lower layer, then the classname of the glue will be, according to our naming convention, N1onN2. The inconveniences of relying on this naming convention for determining the glue type can be eliminated by design of a new generic glue.

## 5 Concluding remarks

We have developed a protocol stack composition tool that accepts application requirements and, based on them, finds the appropriate network protocol combination. Protocol layers provide and require specific functionality. This functionality is interpreted by the composition tool and matched with application-specific requirements to compose a protocol stack.

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. We propose the solution of deploying a translation layer, which 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 21.



**Figure 21.** Domain-specific translation layer

Open issues that we may further investigate, related to the composition policies, comprise:

- Dealing with context-aware up- and downward requirements
- Unsolvable requirements: Under certain circumstances, it is possible that the policy for stack composition does not converge to any solution. One should always take into

account the possibility of failure to build a stack. In this case, the client may retry with less strict requirements.

- "Fuzzy logic". Instead of specifying whether or not a certain property (such as "reliability") is desired, an application may specify that a certain degree of a property is considered sufficient. For example, instead of requesting "reliability", an application could consider that "80% reliability" is OK.
- Boolean expression conditions. Instead of specifying a list of requirements which must be all simultaneously accomplished, boolean expressions of requirements with AND/OR conditions between them may be used.
- Support for utility functions. An application may indicate how important each of the desired properties is and the algorithm makes a selection based on this trade-off.

We consider and will investigate the reuse of our composition approach in different application domains, as a general approach for automatically composing layer-oriented systems. This could be possible by deploying different 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.

## 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.

[CE99] Krzysztof Czarnecki, Ulrich Eisenecker, Components and Generative Programming, Proceedings of ESEC/FSE'99, LNCS 1687, pp. 2-19, 1999

[DFY98] Sunshil Da Silva, Danilo Florissi, Yechiam Yemini, Composing Active Services in NetScript, DARPA Active Networks Workshop, 1998

[GHJV] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns

[GoTa] Michael Goodrich, Roberto Tamassia, Data Structures and Algorithms in Java

[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

[Pepita99] Technical description of Pepita, June 1999

[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