A Pi-Calculus based Framework for the Composition and Replacement of Components

Claus Pahl
Dublin City University, School of Computer Applications
Dublin 9, Ireland
cpahl@compapp.dcu.ie

Abstract

Evolution in component systems is critical with respect to the maintainability of these systems. Systems evolve due to changes in the environment or due to improvements of individual components. Even though component technology aims at reducing the dependencies between components, necessary replacements of components might affect the composition of systems. We introduce a composition and replacement calculus based on the π-calculus, allowing us to specify composition and to reason about replacements and their effects.

1. Introduction

A formal semantics for components and component composition is essential if rigorous analysis and reasoning in the development and maintenance of component systems shall be deployed. Our objectives are a formalisation of basic composition principles, similar to [1, 2, 3], and the provision of a framework for change and evolution analysis. The motivation to use the π-calculus lies in a similarity between the notions of *mobility* in the π-calculus and *evolution* in component technology. Mobility is defined as the capacity to change the connectivity of a network, i.e. to change the spatial configuration. Evolution in large component system is also about the change of connections between components. Therefore, the π-calculus seems to be a suitable formal notation to develop a framework for specification and reasoning about component composition and, in particular, evolution in these systems. As we will see later on, we will propose some change to basic π-calculus semantics and develop a type system reflecting component technology principles. The result is a mixed calculus, based on π-calculus basics and concepts from component technology. The type system plays the role of the integrator. Types govern how names can be used in process calculi [4]. They classify patterns of behaviour, and they can also reflect connectivity and the control mobility and evolution.

We essentially define a composition protocol with different phases: matching and connector establishment, invocation and execution of the service, an invocation reply, and, later on, dynamic replacements. The different phases will be described by different transition rules. We argue that a process-oriented look at component composition is of major importance for the reliability and maintainability of evolving systems. We will complement this process view on compositions with methods to reason about replacements of components in changing environments. The evolution of systems is often neglected in formal approaches to software engineering problems. Some papers have addressed dynamic reconfiguration and replacement, see for example [5] or [6], but most of these papers have addressed the problem from a pragmatic or not fully rigorous point of view. We devise a consequent approach in this direction, overcoming these deficiencies, by formulating a formal framework of change and reconfiguration for component composition.

Section 2 illustrates component composition. In Section 3 we introduce some basics of our composition and replacement calculus. Section 4 focusses on contract-based matching and connector establishment. The type system is introduced in Section 5. Section 6 specifies the life cycle of component composition, without looking at replacements. Replacements and how to reason about their effect is the subject of Section 7. Finally, we discuss related work and end with some conclusions.

2. Components and Composition

Our component model is based on port-based components, where ports represent services, with export and import interfaces for provided and requested services. Two component interfaces are presented below - `Interface` requesting services and `DocServer` providing services - both part of a document manipulation and storage system.

Component `Interface`
import services
`servReqDoc(uri:URI):Doc`
`servModDoc(doc:Doc,upd:Txt)`
export services
`openDoc(uri:URI)`
`saveDoc(uri:URI, upd:Txt)`

The interfaces uses services from the server component to request (load) and modify (store) documents.

Component `DocServer`
import services
...
export services
`reqDoc(uri:URI):Doc`
`modDoc(doc:Doc,upd:Txt)`

Documents are identified by URIs – uniform resource identifiers. The request service `reqDoc` returns a document, but does not change the state of the server component, whereas the modification service `modDoc` updates a document on the servicer side without returning a value.

Service ports are points of access described in component interfaces. We assume these ports to be specified in some kind of logic that allows to express pre- and postconditions as abstractions for ports [7, 8], enabling the design-by-contract approach [8, 9]. Hoare logic or modal logics are suitable frameworks [10, 11]. A requirements specification of the service user `servModDoc` could look like:

```
servModDoc ( myDoc:Doc, myUpd:Txt )
     pre   valid()
     post  updated()
```

Documents shall be XML-documents here, which can be well-formed (correct tag nesting) or valid (well-formed and conform to a document type definition DTD). A service provider specification could look like:

```
modDoc ( doc:Doc, upd:Txt )
     pre   wellFormed()
     post  updated() ∧ acknowledged()
```

A contract can be formed between interface and document server. The service `modDoc` of the document server matches the requirements of `servModDoc` - a service that might be called in methods provided by the interface. `modDoc` has a weaker, less restricted precondition - `valid()` implies `wellFormed()` - and a stronger postcondition - the conjunction `updated()` ∧ `acknowledged()` implies `updated()`. This means that the provided service satisfied the requirements; it is even better than requested.

We will neglect a detailled presentation of component semantics in this presentation; our focus is on the *composition semantics*. The composition of components [12, 13] will be defined using the π-calculus [14, 15, 16]. Interaction is the composition principle. Ports represent services of a component. We will distinguish different roles of ports, e.g. whether a service is provided or requested. The semantics of service execution is reflected in the type system through pre- and postconditions. The type system is the link between the component and the composition semantics. However, the pre- and postconditions, forming a contract, are also important for the composition process. A conformance condition expresses that two ports match based on whether a provided service satisfies the needs of a requested service. Technically, the matching process is facilitated through contract ports $\texttt{servModDoc}_C$ and $\texttt{modDoc}_C$. The user interface requires a service (annotation REQ) and the document server provides a service (annotation PRO).

$$\texttt{Interface} \stackrel{\text{def}}{=} \text{REQ } \overline{\texttt{servModDoc}_C}\langle\texttt{servModDoc}_I\rangle.\texttt{Interface}'$$
$$\texttt{DocServer} \stackrel{\text{def}}{=} \text{PRO } \texttt{modDoc}_C(\texttt{modDoc}_I).\texttt{DocServer}'$$

The names are not relevant for components to match - only the pre- and postconditions represented through types are. Successful matching results in the establishment of a connector - a private channel between the components that allows one component to use services provided by the other.

$$\texttt{Interface}' \stackrel{\text{def}}{=} \text{INV } \overline{\texttt{servModDoc}_I}\langle\texttt{doc}, \texttt{upd}\rangle.\texttt{Interface}''$$
$$\texttt{DocServer}' \stackrel{\text{def}}{=} \text{EXE } \overline{\texttt{modDoc}_I}\langle\texttt{x}_1, \texttt{x}_2\rangle.\texttt{DocServer}''$$

The user interface can invoke the service (INV) through the interaction port $\texttt{servModDoc}_I$, which will trigger the execution (EXE) of $\texttt{modDoc}_I$ by the server. The composition, i.e. establishment of a connector, is one of the key activities; the other is replacement. Systems evolve over time, components are replaced by improved or modified versions. Methods to answer whether for example the user interface or the document server can be replaced shall be introduced.

3. Calculus - Syntax and Type Basics

This section shall introduce basics of our calculus, such as syntax of the notation and some type issues, before we look at rules for component composition.

3.1 Syntax of Component Composition

The basic element describing activity in the π-calculus are actions. Actions are combined to process expressions. Actions are expressed as prefixes to these process expressions:

$$\pi ::= \text{PTYPE } \overline{x}\langle y\rangle \mid \text{PTYPE } x(y) \mid \tau$$

Actions can be divided into output actions $\overline{x}\langle y\rangle$ - the name y is sent along channel (or port) x -, input $x(y)$ - y is received along x, and a silent non-observable action τ. We have annotated these action prefixes by port types, which will explain the role of the port with respect to life cycle activities such as service request or service invocation. Here is the full list of action prefixes, their port types and polarities (types and polarities will be explained soon):

π	::=	REQ $\overline{m_C}\langle m_I\rangle$	+	Request
		PRO $n_C(n_I)$	−	Provide
		INV $\overline{m_I}\langle a_1, \ldots a_l, m_R\rangle$	+	Invoke
		EXE $n_I(x_1, \ldots, x_k, n_R)$	−	Execute
		REP $\overline{n_R}\langle b\rangle$	+	Reply
		RES $m_R(y)$	−	Result

The syntax of composition expressions involving the action prefixes is the following:

P	::=	$\nu m\ P$	Restriction
		$P_1 \mid P_2$	Parallel composition
		$!P$	Iteration
		$\Sigma_{i\in I}\ \pi_i.P_i$	Summation
		0	Inaction

Restriction means that m is only visible in P. Summation $\pi_i.P_i$ means that one π_i is chosen and the process tranfers to state P_i. Iteration $!P$ means that the process is executed an arbitray number of times. This follows the presentation of the π-calculus in [15]. We also need abstractions, i.e. defining equations of the form $A(a) = P_A$. Even though the polyadic π-calculus is intended to be used, we often use the monadic variant here in order to keep the notation simple. The substitution $\{b/a\}$P means that b replaces a in P.

3.2 Ports and their Types

The entities in our composition system are values, ports and components. Values are characterised by the usual value domains as types. The list of basic types $t_1, t_2, \ldots$ shall be assumed, but not explicitly specified. Components are syntactically characterised by an interface with service signatures, separated into import and export elements.

The most important entities are the ports. Each port p is essentially a family of ports $p = (p_C, p_I, p_R)$. The first port p_C is the *contract port*, essentially an abstract interface described by a signature, a precondition and a postcondition. p_I is the connector activation (or interaction) port. This port is used to invoke a service. The port p_R carried

m_C : $\textsc{Ctr}(\textsc{Sig}(T_1, \ldots, T_n, +\textsc{Cre}(T)), \textsc{Prd}(\textsc{pre}), \textsc{Prd}(\textsc{post}))$
n_C : $\textsc{Ctr}(\textsc{Sig}(T'_1, \ldots, T'_n, \textsc{Cre}(T')), \textsc{Prd}(\textsc{pre'}), \textsc{Prd}(\textsc{post'}))$
m_I : $\textsc{Cac}(T_1, \ldots, T_n, \textsc{Cre}(T))$
n_I : $\textsc{Cac}(T_1, \ldots, T_n, \textsc{Cre}(T))$
m_R : $\textsc{Cre}(T)$
n_R : $\textsc{Cre}(T)$

Figure 1: Ports and their channel types.

the reply from the service invocation. We distinguish a port type and a channel type for each port.

Port types describe the functionality of the port within the component (e.g. contract or connector ports) and its orientation (in- and out-ports). Port types $\mathcal{T}_p(p)$ or $p :_p t$ for port p are denoted by 3-letter lower case abbreveations. The annotations in the action prefix syntax (see Section 3.1) denote port types, e.g. $\overline{m_C}$ is a request port; n_C is the dual provide-port: $\mathcal{T}_p(\overline{m_C}) = \textsc{Req}$ and $\mathcal{T}_p(n_C) = \textsc{Pro}$. Each port has also an orientation - called the *polarity*; all ports of one port family follow always the same orientation pattern: $+, +, -$ for requested (imported) services, saying that contract and connector ports are output ports ('+') and the reply port is an input port ('−'), and $-, -, +$ for provided (exported) services. Each '+' stands for an output capability (the port can only send); '−' stands for an input capability (the port can only receive).

Channel types describe what kind of entities can be transported: a contract port $p_C :_c \textsc{Ctr}(\textsc{sig}, \textsc{pre}, \textsc{post})$, a connector activation or interaction port $p_I :_c \textsc{Cac}(T_1, \ldots, T_n, \textsc{Cre}(T))$, and a connector reply port $p_R :_c \textsc{Cre}(T)$. This characterises the channel by specifying the expected capacity - what data can be transported. It will constrain the composition and interaction between components. Contract ports can transport connectors, which are characterised by a contract type. Connectors provide the connection between components to invoke a service. Channel types $\mathcal{T}_c(p)$ or $p :_c t$ for port p are denoted by 3-letter abbreveations starting with an upper case character, see Figure 1.

A contract consists of a service signature, a pre- and a postcondition. Connectors when transfered on channels have to satisfy a contract type. On connector activation ports, data values and a reply channel can be transfered; on connector reply ports, data can be transfered. The key criterion for matching, i.e. the succesful connection of two components through a connector, are are contracts (this will be explained in the next subsection). Opposite orientations also have to match in a successful composition of component ports. The signature for a remote method execution is: $\textsc{Sig}(T_1, \ldots, T_n, \textsc{Cre}(T))$. This is an adequate representation, reflecting the fact that parameters are passed, and possibly a result has to be transfered back on a channel with a different capacity (type). For local method executions, the usual notation $T_1 \times \ldots \times T_n \rightarrow T$ apply. Pre- and postconditions are formed using the predicate type constructor $\textsc{Prd}$.

Connector ports represent services. Connectors are channels that can carry data elements - for the connector activation additionally a reply channel. Connector ports and connector reply ports are only used as restricted (private) channels between components that match based on contracts. This means that these channels are only available to these two components.

3.3 Subtypes

In principle, the definition of a subtype relation is possible for all kinds of entities in our notation. However, we will focus on ports here. The subtype relation will help us to determine whether two ports, representing services, match and whether they can be composed. Later on, this concept will also be used to determine consistent (effect-free) replacements.

We have already seen that the channel types of contract ports are contracts consisting of a service signature, a precondition and a postcondition. For a required service $m_C :_c \textsc{Ctr}(\textsc{sig},\textsc{pre},\textsc{post})$ and a provided service $n_C :_c \textsc{Ctr}(\textsc{sig'},\textsc{pre'},\textsc{post'})$, we say that n_C matches m_C if

$$\textsc{sig} = \textsc{sig'} \wedge \textsc{pre} \rightarrow \textsc{pre'} \wedge \textsc{post'} \rightarrow \textsc{post}$$

This is the combination of two classical refinement relations (weaken the precondition and strengthen the postcondition) from the Refinement Calculus [17, 18], see also [19] for other matching approaches.

4. Contract Matching and Connectors

We define the operational semantics of component composition in this section. m_C, n_C are contract ports, m_I, n_I are connector activation or interaction ports, and m_R, n_R are connector reply ports. (m_C, m_I, m_R) is an output port, i.e. actively requests services from other components, and (n_C, n_I, n_R) is an input port, providing services to other components. Ports are typed. The connector ports (activation and reply) are restricted to components matched based on contracts. We will introduce a number of transition rules describing state changes, including activities such as contract matching, connector establishment, interaction and interaction reply. The rules will define the *transition semantics*. They will replace some of the transition rules for the classical π-calculus, in particular the reaction rules - see e.g. [16] Table 1.5. Other rules are still valid.

4.1 Contracts

Two components can react, i.e. can be connected, if their contract types (end points of possible channels) form a subtype relationship. This changes the original π-calculus reaction rule which requires channel names to be the same. Here we only require a subtype relationship between the ports. The receiver can *accept* an input based on the type, not the name. We assume the following channel types $t_{m_C} = \textsc{Ctr}(\textsc{sig}, \textsc{pre}, \textsc{post})$ and $t_{n_C} = \textsc{Ctr}(\textsc{sig'}, \textsc{pre'}, \textsc{post'})$ for contract ports m_C and n_C, respectively. The contract type t_{n_C} is a subtype of t_{m_C}, if the precondition is weakened and the postcondition is strengthened. Signatures are assumed to be equal. The following transition rule – the **contract rule** [T-Ctr] – describes matching of services:

$$\frac{\textsc{Req}\ \overline{m_C}\langle m_I\rangle.C \xrightarrow{\overline{m_C}\langle m_I\rangle} C \qquad \textsc{Pro}\ n_C(n_I).P \xrightarrow{n_C(n_I)} P}{\textsc{Req}\ \overline{m_C}\langle m_I\rangle.C | \textsc{Pro}\ n_C(n_I).P \xrightarrow{\tau} C \frown P} \langle\ \Phi$$

where side condition is $\Phi \equiv t_{n_C} \leq t_{m_C}$. The connection $C \frown P \stackrel{\text{def}}{=} \nu c(\{c/m_I\}C | \{c/n_I\}P)$ introduces a fresh variable c - free in C and P - creating a private (restricted) channel c called the connector. This rule expresses the connector

establishment. Ultimately, we will chain together several components that will import services from others.

A second typing constraint is hidden. REQ and PRO denote port types, i.e. $\overline{m_C} :_p \textsc{Req}$ and $n_C :_p \textsc{Pro}$. These are type annotations to the ports. Here, the port types match: REQ is the complement of PRO, and the polarities are opposite. We write $\mathcal{T}(\overline{m_C}) \simeq \mathcal{T}(n_C)$ in this case.

Type systems for the π-calculus usually constrain data that is sent, here we constraint reaction (the interaction between agents). The contract rule cannot be translated to the match-rule found in some π-calculus variants. Our contract rule is similar to transition rules describing reaction that are based on bounded output $\overline{x}(z)$ where z is introduced as a bound variable forming a restricted channel [16]. We have chosen to introduce a fresh variable c instead.

4.2 Connectors

We assume that a private channel c - the connector - has been established between client and provider. This channel is used by the client to invoke a service n_I at the server side. Parameter data $a : t_a$ with $t_a \leq t_x$ and a reply channel $m_R : t_{m_R}$ are send to the provider. The **connector activation rule** [T-CAC] is defined as follows:

$$\frac{\textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle.C \xrightarrow{\overline{m_I}\langle a,m_R\rangle} C \quad \textsc{Exe}\ n_I(x, n_R).P \xrightarrow{n_I(x,n_R)} P}{\textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle.C | \textsc{Exe}\ n_I(x, n_R).P \xrightarrow{\tau} C \frown \{a/x\}P} \langle\ \Phi$$

where $\Phi \equiv t_{n_I} \leq t_{m_I}, a : pre$. The types t_{m_I} and t_{n_I} are the connector activation types $\textsc{CAc}(t_1, \ldots, t_m, \textsc{CRe}(t))$ and $\textsc{CAc}(t'_1, \ldots, t'_n, \textsc{CRe}(t'))$, respectively. The reply channel is again a private channel between the two components that replaces m_R and n_R. Type equality (or a subtype relation) for m_I and n_I is not required if we can guarantee that the connector types satisfy the contract types and that the contract matching has successfully been executed. A protocol - specified in form of a component life cycle - can guarantee this. We will discuss the side condition $a : pre$ shortly.

The last rules is the **connector reply rule** [T-CRE]:

$$\frac{\textsc{Res}\ m_R(y).C \xrightarrow{m_R(y)} C \quad \textsc{Rep}\ \overline{n_R}\langle b\rangle.P \xrightarrow{\overline{n_R}\langle b\rangle} P}{\textsc{Res}\ m_R(y).C | \textsc{Rep}\ \overline{n_R}\langle b\rangle.P \xrightarrow{\tau} \{b/y\}C \frown P} \langle\ \Phi$$

where $\Phi \equiv t_{n_R} \leq t_{m_R}, b : post$. We assume $t_b \leq t_y$. b is the result of the internal computation, i.e. b is a function of x.

The contract with pre- and postconditions can be reflected at the connector level. We associate pre- and postconditions with the in- and out-ports. Input a is required to satisfy the precondition pre and output b is required to satisfy the postcondition $post$. These assertions are obligations, formalised by contracts, to be satisfied by client (pre) and provider (post) at runtime. This attachment of obligations to the connectors results in more symmetry and links contracts and connectors. Pre- and postconditions are formulas, but here they are evaluted at runtime when the corresponding method is invoked and executed.

5. Types and Subtypes

We use the type system to control the correct establishment, use and replacement of connections between components. Especially subtypes are important for this purpose.

We use typing rules to describe our type system. Syntactial aspects of our notation have been dealt with in previous sections. We will address the relation between the type system and the transition semantics. The type safety property guarantees that well-typed expressions (expressions whose types can be infered using the type system) do not fail under transition. We show that the well-typedness is preserved.

T	::=	B	Basic type
	|	L	Link type
	|	$\textsc{Sig}(T \times \ldots \times T \times L)$	Signature
	|	$\textsc{Prd}(T)$	Predicate
L	::=	$P\ C$	Port and channel type
P	::=	$\pm$ (REQ | PRO | INV | EXE | REC | REP)	Port type
C	::=	$\textsc{Ctr}(T \times T \times T)$	Contract
	|	$\textsc{CAc}(T \times \ldots \times T \times L)$	Connector activation
	|	$\textsc{CRe}(T)$	Connector reply

Figure 2: The syntax of the type language.

5.1 Typing Rules

A typing context Γ is a finite set of bindings - mappings from names to types. Three types of judgments are used:

$\Gamma \vdash x : T$	name x has type T
$\Gamma \vdash S \leq T$	type S is subtype of T
$\Gamma \vdash P$	expression P is well-typed

The type language syntax is defined in Figure 2. The constructors CTR, CAC, CRE are the link-type constructors. Their purpose is to classify channels based on the data that is transfered along them. We leave the set of value types unspecified. We assume that there is at least one basic type. SIG and PRD are standard constructors for service signatures and predicates, the other type constructors are application-specific to the component context.

The semantics of the type system will be defined by typing rules for basic types, type constructors, subtypes and process expressions. We will now address these different kinds of rules, see Figure 5.1. Transition rules based on these typing rules have been given in the previous section.

Typing rules for the type constructors (contract, connector, signature, predicate) shall be omitted, except for the one for contracts, I-CTR. If the three names s, p_1 and p_2 are of type signature, predicate, and predicate, respectively, then the contract $\textsc{Ctr}(s, p_1, p_2)$ is of contract type:

$$\textsc{Ctr}(\textsc{Sig}(T_1, \ldots, T_n, \textsc{CRe}(T), \textsc{Prd}(F_1), \textsc{Prd}(F_2))$$

Subtype relations are in principle possible between types constructed with the same constructor. Two structural rules contribute to the definition of the subtype relation $\leq$: the reflexivity rule S-REFL and the transitivity rule S-TRANS:

$$[\text{S-REFL}]\ \frac{S =_\beta T}{\Gamma \vdash S \leq T}$$

$$[\text{S-TRANS}]\ \frac{\Gamma \vdash S \leq T \quad \Gamma \vdash T \leq U}{\Gamma \vdash S \leq U}$$

They show that $\leq$ is a preorder. The subtyping rules for signatures and predicates are S-SIG and S-PRD - see Figure 5.1. The names COND, PRE, POST, SIGN and their primed variants are type variables. A condition is subtype of another if it implies it: $\textsc{cond} \leq \textsc{cond}'$ if $\textsc{cond} \rightarrow \textsc{cond}'$. A

$$\text{[I-Ctr]}\quad \frac{\Gamma \vdash s :_c \textsc{Sig}(T_1, \ldots, T_n, \textsc{CRe}(T))\quad \Gamma \vdash p_1 :_c \textsc{Prd}(T)\quad \Gamma \vdash p_2 :_c \textsc{Prd}(T)}{\Gamma \vdash \textsc{Ctr}(s, p_1, p_2) :_c \textsc{Ctr}(\textsc{Sig}(T_1, \ldots, T_n, \textsc{CRe}(T)), \textsc{Prd}(T), \textsc{Prd}(T))}$$

$$\text{[S-Sig]}\quad \frac{\Gamma \vdash T_1' \leq T_1 \quad \ldots \quad \Gamma \vdash T_k' \leq T_k \quad \Gamma \vdash \textsc{CRe}(T) \leq \textsc{CRe}(T')}{\Gamma \vdash \textsc{Sig}(T_1', \ldots, T_n', \textsc{CRe}(T')) \leq \textsc{Sig}(T_1, \ldots, T_n, \textsc{CRe}(T))}$$

$$\text{[S-Prd]}\quad \frac{\textsc{cond}' \rightarrow \textsc{cond}}{\Gamma \vdash \textsc{Prd}(\textsc{cond}') \leq \textsc{Prd}(\textsc{cond})}$$

$$\text{[S-Ctr]}\quad \frac{\Gamma \vdash \textsc{pre} \leq \textsc{pre'}\quad \Gamma \vdash \textsc{post'} \leq \textsc{post}\quad \Gamma \vdash \textsc{sig'} \leq \textsc{sig}}{\Gamma \vdash \textsc{Ctr}(\textsc{sig'}, \textsc{pre'}, \textsc{post'}) \leq \textsc{Ctr}(\textsc{sig}, \textsc{pre}, \textsc{post})}$$

$$\text{[S-CAc]}\quad \frac{\Gamma \vdash T_1' \leq T_1 \quad \ldots \quad \Gamma \vdash T_k' \leq T_k \quad \Gamma \vdash \textsc{CRe}(T) \leq \textsc{CRe}(T')}{\Gamma \vdash \textsc{CAc}(T_1', \ldots, T_k', \textsc{CRe}(T')) \leq \textsc{CAc}(T_1, \ldots, T_k, \textsc{CRe}(T))}$$

$$\text{[S-CRe]}\quad \frac{\Gamma \vdash T' \leq T}{\Gamma \vdash \textsc{CRe}(T') \leq \textsc{CRe}(T)}$$

Figure 3: Typing rules.

contract forms a subtype of another if its precondition is weakened and its postcondition is strengthened - see S-Ctr - where sig, pre, post, sig', pre', and post' denote signature and predicate types. The port orientation also has to be considered. We assume that ports do not change their orientation. For connector activations we expect subtype relations for the value types to hold - see S-CAc. This definition is - similar to the signature subtypes - contravariant on the reply channel. A connector reply channel is a subtype of another if the value types that can be carried form a subtype - see S-CRe. Subtypes for the value kind shall be neglected for the rest of the paper - which has as a consequence that there are no proper subtypes between signatures and connector activations and replies.

5.2 Type Safety

Type safety concerns the relation between the type system and the operational semantics. The operational semantics are defined as transition semantics, specified by rules such as contract matching and connector establishment. Type safety comprises two issues. Firstly, evaluation should not fail in well-typed programs - we will introduce a notion of well-typedness shortly. Secondly, transitions should preserve typing. The judgment $\Gamma \vdash C$ denotes the well-typedness of composition expression C. This will be the construct to investigate type preservation under transition.

We need to define a notion of satisfaction before we can define well-typedness. A connector type satisfies a contract type if the signatures correspond and, if the precondition holds, the execution of the service attached to the connector port establishes the postcondition.

Definition 5.1. *A connector type* $T_I = \textsc{CAc}(T_1, \ldots, T_n, \textsc{CRe}(T))$ *satisfies a contract type* $T_C = \textsc{Ctr}(\textsc{sig}, \textsc{pre}, \textsc{post})$, *or* $T_I \models T_C$, *if for a service port* p *the connector port* p_I *satisfies the following constraints:* $\textsc{Sig}(T_1, \ldots, T_n, \textsc{CRe}(T)) = \textsc{sig}$ *and, if* pre *holds, then the execution of* p_I, *if it terminates, establishes* post.

We assume an analogous definition of satisfaction between data types and connector reply types and their connector activation type.

Definition 5.2. *We define well-typedness for simple actions as follows:*

- $\Gamma \vdash \textsc{Req}\ \overline{m_C}\langle m_I\rangle$ *if* $\mathcal{T}_c(m_I) \models \mathcal{T}_c(m_C)$ – *otherwise* $\textsc{Req}\ \overline{m_C}\langle m_I\rangle$ *fails.*
- $\Gamma \vdash \textsc{Pro}\ n_C(n_I)$ *if* $\mathcal{T}_c(n_I) \models \mathcal{T}_c(n_C)$ – *otherwise* $\textsc{Pro}\ n_C(n_I)$ *fails.*
- $\Gamma \vdash \textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle$ *if* $type(a), \mathcal{T}_c(m_R) \models \mathcal{T}_c(m_I)$ – *otherwise* $\textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle$ *fails.*
- $\Gamma \vdash \textsc{Exe}\ n_I(y, n_R)$ *if* $type(y), \mathcal{T}_c(n_R) \models \mathcal{T}_c(n_I)$ – *otherwise* $\textsc{Exe}\ n_I(y, n_R)$ *fails.*

The execution of an action fails, if the data sent along the channel does not satisfy the channel constraint. A reaction fails if both participating actions are well-typed, but the type constraint is not satisfied.

Definition 5.3. *The well-typedness of parallel compositions is defined by rule* [W-ParComp]*:*

$$\frac{\Gamma \vdash \textsc{Req}\ \overline{m_C}\langle m_I\rangle\quad \Gamma \vdash \textsc{Pro}\ n_C(n_I)\quad \Gamma \vdash \mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)}{\Gamma \vdash \textsc{Req}\ \overline{m_C}\langle m_I\rangle | \textsc{Pro}\ n_C(n_I)}$$

If $\Gamma \vdash \textsc{Req}\ \overline{m_C}\langle m_I\rangle$ *and* $\Gamma \vdash \textsc{Pro}\ n_C(n_I)$, *but not* $\Gamma \vdash \mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)$, *then* $\textsc{Req}\ \overline{m_C}\langle m_I\rangle | \textsc{Pro}\ n_C(n_I)$ *fails.*

Well-typedness guarantees correct composition and interaction behaviour according to the specifications given through the type system (pre- and postconditions) constraining behaviour and matching. The objective later on will be to show whether replacements preserve well-typedness; for example to show that if $\Gamma \vdash \textsc{Req}\ \overline{m_C}\langle m_I\rangle | \textsc{Pro}\ n_C(n_I)$ and if port m is replaced by m', then to show whether $\Gamma \vdash \textsc{Req}\ \overline{m_C'}\langle m_I'\rangle | \textsc{Pro}\ n_C(n_I)$ holds, i.e. whether correct behaviour is preserved by replacements.

We shall note type safety properties as conjectures only, without a formal proof.

Conjecture 5.1.

1. *Substitution lemma: if* $\Gamma \vdash C$ *and* $\Gamma \vdash x : T, v : T$, *then* $\Gamma \vdash \{v/x\}C$.
2. *Evaluation cannot fail in well-typed programs: if* $\Gamma \vdash C$ *then the execution of* C *does not fail.*
3. *Transition preserves typing: if* $\Gamma \vdash C_1$ *and* $C_1 \to C_2$ *then* $\Gamma \vdash C_2$.

5.3 Types as Formulas

There is a relationship between the contracts and connector types. Contract types can be see as Hoare logic or dynamic (modal) logic formulas consisting of a precondition and a postcondition, complemented by a signature. We have a two-layered type system with a layer of contract types and a layer of connector types with a notion of satisfaction between them. These types correspond to the distinction of specification and implementation for a component. The contract type CTR(SIG, PRE, POST) corresponds to the formula PRE $\to [n(a_1, \ldots a_k)]$ POST in dynamic logic where n : SIG. This refers to the specification of services of a component. The lower type layer corresponds to the implementation. Types for parameters are value types.

6. Client and Provider Life Cycles

In the previous sections, we have seen several stages in the life cycle of a component such as service matching and connector establishment, or service invocation. The full life cycle of clients, providers and systems consisting of both clients and providers shall now be specified in a standard form. The client, parameterised by a list of required services, can be specified as follows:

$$
\begin{array}{l}
C_i(m_1, \ldots, m_l) \stackrel{\text{def}}{=} \\
\quad \text{REQ } m_C^1\langle m_I^1\rangle.!(\text{INV } \overline{m_I^1}\langle a^1, m_R^1\rangle.\text{RES } m_R^1(y^1).0) \\
\quad | \\
\quad \ldots \\
\quad | \\
\quad \text{REQ } m_C^l\langle m_I^l\rangle.!(\text{INV } \overline{m_I^l}\langle a^l, m_R^l\rangle.\text{RES } m_R^l(y^l).0)
\end{array}
$$

Requests have to be satisfied before any interaction can happen. Once a connection is established, a service can be used several times. In order to function properly all service requests need to be satisfied - expressed by the parallel composition of all individual ports.

Service providers need to be replicated $!P$ in order to deal with several clients at the same time. Otherwise their behaviour is the dual to that of the clients.

$$
\begin{array}{l}
P(n_1, \ldots, n_k) \stackrel{\text{def}}{=} \\
\quad !(\ \text{PRO } n_C^1(n_I^1).!(\text{EXE } n_I^1(y^1, n_R^1).\text{REP } \overline{n_R^1}\langle b\rangle.0) \\
\quad\quad + \\
\quad\quad \ldots \\
\quad\quad + \\
\quad\quad \text{PRO } n_C^k(n_I^k).!(\text{EXE } n_I^k(y^k, n_R^k).\text{REP } \overline{n_R^k}\langle b\rangle.0)\ )
\end{array}
$$

A provider does not need to engage in interactions with all its ports, which is modelled by using the choice operator instead of the parallel composition.

Clients and a server are composed in parallel $CompSys \stackrel{\text{def}}{=} P(n_1, \ldots, n_k) \mid C_1(m_{1_1}, \ldots, m_{1_{m_1}}) \mid \ldots \mid C_j(m_{j_1}, \ldots m_{j_{m_j}})$ to form a composed system. Another case which also needs to be considered is that a component can be both client and provider, i.e. can both import and export services.

$$
\begin{array}{rl}
Comp \stackrel{\text{def}}{=} & (\text{REQ } m_C^1\langle m_I^1\rangle.0 | \ldots | \text{REQ } m_C^l\langle m_I^l\rangle.0). \\
& !(\ !(\ \text{INV } \overline{m_I^1}\langle \ldots\rangle.\text{REC } m_I^{R^1}(\ldots).0 \\
& \quad + \\
& \quad \ldots \\
& \quad + \\
& \quad \text{INV } \overline{m_I^l}\langle \ldots\rangle.\text{REC } m_I^{R^l}(\ldots).0\ ) \\
& + \\
& P(n_1, \ldots, n_l)\ )
\end{array}
$$

The requirements have to be satisfied, i.e. connectors have to be established, before any service can be provided. A service which is provided and actually invoked can then trigger the invocation of imported services. The specification of composed systems does not involve the possibility for evolution - through the replacement of components - so far. This will be looked at in the next section.

7. Replacements and Evolution

In evolving systems, components might change in their specification or implementation, or are replaced by other components with different specification and implementation. Two questions arise. Firstly, can a component be replaced by another component without affecting the behaviour and the overall consistency of the system? This can be answered using a static analysis based on the component contracts. Secondly, what are the consequences if a replacement fails? This can affect a running system. The analysis has to be carried out based on the actual connectors between components in a running system. We assume that only a single component is replaced by another at a time. Components are the unit of change.

We will address replacements based on the type system – statically and dynamically – and the determination of effects if such a replacement results in inconsistencies. Types are explicit in our notation. That will allow us to change the type (and implementation) of a component, see Section 7.1. This can even be done dynamically for a running system, see Section 7.2. In case the types cannot be preserved, the effects of a change need to be determined, see Section 7.3. We shall look at replacements firstly as a meta-construct, then we will introduce it into the notation. We assume that replacing a component means replacing existing ports, possibly adding new ones. We discuss the replacement of a single port only in order to illustrate the issue. The following definitions formalise a consistent (effect-free) replacement.

Definition 7.1. *A* **context** X *is obtained when a hole* [.] *replaces an occurrence of* 0 *in a process expression. We write* $X[P]$ *for the replacement of* [.] *by* P *in* X.

Definition 7.2. *Given an arbitrary context* X, *a component (a process expression)* C *can be* **consistently replaced** *by a component* C', *if* $\Gamma \vdash X[C]$ *implies* $\Gamma \vdash X[C']$.

This describes the preservation of well-typedness under replacement. It guarantees that replacements do not affect the composition behaviour.

Proposition 7.1. *If* $\Gamma \vdash C$ *implies* $\Gamma \vdash C'$, *then* $\Gamma \vdash X[C]$ *implies* $\Gamma \vdash X[C']$ *for all contexts* X.

PROOF. Obvious. □

The dynamic replacement analysis based on types gives more flexibility. However, the result might be an effect on other components in form of a change of connections (mobility).

7.1 Replacement and Subtypes

Changes in structure - reflected by changes in connector types - are usually difficult to deal with, but changes in behaviour - here reflected by changes in contract types -, do not always affect the overall consistency of a composition (the consistency is affected if the specified behaviour is not preserved). Preservation of well-typedness is the technical criterion for this kind of analysis.

Since our aim is to determine whether one component can replace another, we can consider the type system and its subtypes. We will look at bound names in providers and free names in clients in particular. We do not consider type equivalence here; our concern is the replacement of one component by another relying on the subtype relation.

Clients, or service requestors, shall be addressed first. A port $m = (m_C : t_C, m_I : t_I, m_R : t_R)$ shall be replaced by $m' = (m'_C : t'_C, m'_I : t'_I, m_R' : t_R')$. Later on, we will assume that names do not change, only their types will.

In some situations, replacement preserves well-typedness: for $\Gamma \vdash X[C]$ and C' replaces C, we get $\Gamma \vdash X[C']$ for any context X. This shall now be investigated - firstly for a single component.

Proposition 7.2. *A requested service port $m_C :_p$ REQ can be consistently replaced by a port $m'_C :_p$ REQ if*

$$\mathcal{T}_p(m_C) = \mathcal{T}_p(m'_C) \ \wedge \ \mathcal{T}_c(m_C) \leq \mathcal{T}_c(m'_C)$$

PROOF. m is a refinement of m'. m' has consequently a stronger (more restrictive) precondition and a weaker (less specific) postcondition. $\Gamma \vdash \overline{m'_C}\langle m_I \rangle$ if $\Gamma \vdash \overline{m_C}\langle m_I \rangle$ and $\mathcal{T}_c(m'_C) \leq \mathcal{T}_c(m_C)$ and we assume that $\mathcal{T}_c(m'_I) \leq \mathcal{T}_c(m_I)$ for connectors. Therefore, well-typedness is preserved. □

We shall look at this issue considering one particular context: that of a parallel composition where a client and a provider match. In this particular context, we can loosen the constraint for well-typedness.

Proposition 7.3. *A component C' can replace a client component C in a composition $C|P$ preserving well-typedness, i.e. $\Gamma \vdash C|P \rightarrow \Gamma \vdash C'|P$, if $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m'_C)$ for a service n provided by P, a service m requested by C and replacement m' for m.*

PROOF. Suppose a composition $C|P$ exists where n of P is connected to m of C, i.e. $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)$. As long as $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m'_C)$ the provider satisfies the requirements. This means that C' can replace C without affecting the behaviour of the composition. Well-typedness as formulated in the well-typedness rule for compositions (Definition 5.3) in Section 5.2 is preserved. □

Strengthening the client specification might be acceptable. A refinement m'_C of m_C is acceptable as long as $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)$ is guaranteed. The condition $\mathcal{T}_c(m_C) \leq \mathcal{T}_c(m'_C)$ does not need to be satisfied, but would, if true, guarantee the well-typedness of the replacement. Proposition 7.3 is more flexible than Proposition 7.2, but Proposition 7.3 can only be checked dynamically for a composed system. This condition would have to be checked for all connections in a running system.

We have a similar situation for the service provider.

Proposition 7.4. *A provided service port $n_C :_c$ PRO can be consistently replaced by a port $n'_C :_c$ PRO if*

$$\mathcal{T}_p(n_C) = \mathcal{T}_p(n'_C) \wedge \mathcal{T}_c(n'_C) \leq \mathcal{T}_c(n_C)$$

PROOF. Analogously to Proposition 7.2. □

Here, refinements are always permitted as replacements. Analogously to clients, replacements are consistent (effect-free) as long as the connector remains intact.

Proposition 7.5. *A component P' can replace a server component P in a composition $C|P$ preserving well-typedness if $\mathcal{T}_c(n'_C) \leq \mathcal{T}_c(m_C)$ for a provided service n connected to m and the replacement n'.*

PROOF. Analogously to Proposition 7.3. □

For this form of analysis we have looked at contract ports and their types only. Firstly, because the contract matching is the crucial activity; we essentially consider only contract port related activities as observable. Secondly, their channel types involve the contracts - which the connectors are expected to oblige to. The client is expected to guarantee the precondition and the provider is expected to guarantee the postcondition if the precondition is satisfied. The types of the connector and connector reply ports have therefore been neglected.

7.2 Dynamic Replacement

In order to allow dynamic compositions and replacements, we introduce a new feature into our notation. We introduce an explicit configuration CFG $s(p : t_p)$ for a port that allows components to change their specification dynamically:

$$\begin{array}{lcl} Client_i & \stackrel{\text{def}}{=} & !(\text{CFG } s(m : t_m).\overline{C_i}\langle m \rangle) \\ Provider & \stackrel{\text{def}}{=} & !(\text{CFG } s(n : t_n).\overline{P}\langle n \rangle) \end{array}$$

where C_i and P are defined as in Section 6. The port specification and implementation, i.e. contract and connectors, are provided by some external process.

Ignoring name changes - they can always be introduced easily via renamings - this construct essentially allows us to change the type of a port dynamically. Changes in contract types can be dealt with. The type system shall therefore be revisited. Ports are associated with types, e.g. the typing context Γ can contain a binding $m_C \mapsto$ CTR(SIG,PRE,POST). These associations in the typing context can change through the execution of for example

$$\text{CFG } s(m_C : \text{CTR(SIG',PRE',POST')})$$

where SIG,PRE,POST and SIG',PRE',POST' denote signature and predicate types. We assume that the connector types do not change. The configuration has an effect on the type context. The semantics of CFG $s(p : t_p)$ is that of a dynamic declaration on Γ; we write $\Gamma[\text{CFG } s(p : t_p)]$. We need an initial configuration for a port, but a replacement can

essentially happen at any time:

$$
\begin{array}{lcl}
C & \stackrel{\text{def}}{=} & !\textsc{Cfg}\ s(m_C : t_{m_C}).\overline{C'}\langle m_C\rangle \\
C' & \stackrel{\text{def}}{=} & (\ \textsc{Cfg}\ s(m_C : t_{m_C}).\overline{C'}\langle m_C\rangle \\
 & & \quad + \\
 & & \quad \textsc{Req}\ \overline{m_C}\langle m_I\rangle.(\\
 & & \quad\quad \textsc{Cfg}\ s(m_C : t_{m_C}).\overline{C'}\langle m_C\rangle \\
 & & \quad\quad + \\
 & & \quad\quad !(\textsc{Inv}\ \overline{m_I}\langle\ldots\rangle.(\\
 & & \quad\quad\quad \textsc{Cfg}\ s(m_C : t_{m_C}).\overline{C'}\langle m_C\rangle \\
 & & \quad\quad\quad + \\
 & & \quad\quad\quad \textsc{Rec}\ m_R(\ldots).0)\ \ )\ \ )
\end{array}
$$

This describes that a replacement will always result in a re-establishment of the connector, i.e. the request for a service will be made again. If we want to make use of the results of well-typedness preserving replacements (a re-establishment of connectors is not necessary in that case), we need to make this explicit in the notation. We introduce a type-based guard, remotely similar to the match-operator found in some π-calculus variants: $[TJ]\,\pi.P$ where TJ is a boolean expression based on a type judgment. The type judgment acts as a simple condition making the typing context explicit in the notation. In our situation, we could specify

$$\ldots\ .\textsc{Cfg}\ s(m_C : t_{m_C}).[\neg(\Gamma \vdash C)]\overline{C'}\langle m_C\rangle.\ \ldots$$

expressing that only if typing is not preserved, the re-establishment of a connection is necessary after a replacement.

The substitution of the type context can be formulated in two *dynamic typing rules* based on results from Section 7.1. Proposition 7.2 proves the soundness of the following **client replacement rule** [R-CRPL] for the type system:

$$\frac{\Gamma \vdash m_C :_c t_C \qquad \Gamma \vdash m_C :_p \textsc{Req}}{\Gamma[\textsc{Cfg}\ s(m'_C : t'_C)] \vdash m'_C :_p t'_C}\langle\ t_C \leq t'_C$$

where we replace $m_C : t_C$ by $m_C : t'_C$. We assume that the signature of the port m does not change. We also assume that t'_C is a contract type. The rule expresses a substitution in the type context.

Corresponding to the replacement rule for clients, we can formulate a **provider replacement rule** [R-PRPL] - based on Proposition 7.4.

$$\frac{\Gamma \vdash n_C :_c t_C \qquad \Gamma \vdash n_C :_p \textsc{Req}}{\Gamma[\textsc{Cfg}\ s(n'_C : t'_C)] \vdash n'_C :_p t'_C}\langle\ t'_C \leq t_C$$

Note that only the type requirement for the contract has changed. The execution of the CFG-action does neither affect the composition nor the component state. It only changes the type context dynamically.

Proposition 7.6. *Replacement based on the rules client replacement* [R-CRPL] *and provider replacement* [R-PRPL] *preserves well-typedness. For $\Gamma \vdash C$ and C' replaces C by one of the rules, we get $\Gamma \vdash C'$.*

PROOF. Follows from Propositions 7.2 and 7.4. □

Similar rules for dynamic replacements can be defined based on Propositions 7.3 and 7.5. Methods to analyse and reason about replacements will be addressed next.

7.3 Non-preserving Replacements

The first step in replacing components is always an analysis of types. Based on these criteria a component might be replaced consistently. In case the replacement has to take place - for example due to changes in the execution environment (technical, legal, etc.) - but does not result in a consistent replacement, then the effect of the replacement on other connected components has to be determined. This could again be done statically by looking at all potential compositions based on the overall system specification (all clients and providers composed in parallel), but should rather be limited to those components actually connected to a component in form of connectors (and components connected to those) in a composed system.

Starting point for this dynamic analysis is the network of connected components at a certain moment of time. This is the *flowgraph* of the system, which describes the structure of the system in terms of its linkages between components - the concept of flowgraphs to describe the spatial structure of connected processes is introduced in [15] Chapters 4.1 and 9.3. Flowgraphs and dependency analysis based on contract types shall now be looked at. A component export depends on the component's import; the import depends on another component's export. This dependency relation is transitive and allows us to determine which other components are potentially affected if one component has to be changed. The simple dependency relation needs to be refined. The reason is that a change in the export interface of one component (to the worse) might still satisfy the requirements of a service request. In this case there is no further effect. In case a second component is affected, then it can be tried to replace this component as well (using static analysis).

Definition 7.3. *Two kinds of graphs shall be introduced. A* **flowgraph** *is a graph where nodes are ports m_C, m_I, m_R, n_C, n_I, n_R, ... and edges are connections (m_C, n_C), (m_I, n_I), (m_R, n_R), ... The edges are directed and express dependencies. (m_I, n_I) expresses that a request m_I depends on a provided service n_I. A* **dependency graph** *is a flowgraph extended by component internal dependencies, e.g. (n_I, m_I) saying that export n_I depends on import m_I. Thus, a dependency graph has two kinds of edges: edges (m_I, n_I) are connectors between components and edges (n_I, m_I) are internal dependencies. If (p_1, p_2) and (p_2, p_3) then (p_1, p_3). Types (port type and channel type) shall be associated with each port node.*

Semantically the dependency graph is a bipartite graph defined on two different relations: the *subtype relation* between components and component-internal dependencies between service exports and services imports. A component needs to satisfy its imports in order to provide services to other components. By default, all combinations between input and output services are included in the dependency graph. A more precise account of internal dependencies can be derived from a component life cycle specification (see specification of *Comp* on page 6), which describes the extenally visible interaction behaviour of a component. Requests that occur before a service provision indicate a potential dependency. This could be refined explicitly by the component developer, but this would require to consider the actual service implementations.

The well-typedness of a composition shall be expressed by a consistency notion for dependency graphs.

Definition 7.4. *A dependency graph (or a flowgraph) is* **consistent** *if for all edges (m, n) of the connector kind:*

$$\mathcal{T}_c(n) \leq \mathcal{T}_c(m)\ \wedge\ \mathcal{T}_p(m) \simeq \mathcal{T}_p(n)$$

A graph update is an update of type associations for nodes (ports). This corresponds to the replacement of type bindings $\Gamma[\text{CFG } s(p : t_p)]$ in type context Γ.

Definition 7.5. *If a dependency graph is not consistent for a connector edge (p_2, p_1), then the* **effect** *of the inconsistency is the collection of all (p_i, p_j) with $i \geq 2, j \geq 1$ such that p_i depends on p_1.*

The effect of an inconsistency can be calculated based on the closure of the dependency relation. Classical algorithms can be used here. Therefore, this shall be neglected.

8. Related Work

A composition language for components which is also based on the π-calculus is presented in [20, 21]. A variation of the π-calculus is used to realise a composition language, called PICCOLA, which supports various forms of components, and, thus, various composition mechanisms. The basis is a formalisation of interacting objects as processes. Key concepts are glue code for component compositions and adaptation, and a scripting language to express this glue code.

Catalysis is a development approach building up on the UML incorporating formal aspects such as the pre- and postcondition technique [22]. Catalysis uses ideas from formal languages such as OBJ, CLEAR or EML. The concept of the connector that we have used here is motivated by the Catalysis approach. There, connectors allow the communication between ports of two objects. A connector defines a protocol between the ports. Several other authors also address contracts based on pre- and postconditions for the UML, see e.g. [9]. The combination of the pre- and postcondition technique and refinement calculi is explored in e.g. [23].

KobrA [24] is another approach which combines the UML with the component paradigm. The basic structuring mechanism is the *is-component-of* hierarchy, forming a tree-structured hierarchy of components, i.e. sub-components. Each component is described by a suite of UML diagrams. A component consists of a specification (an abstract export interface) and a realisation.

An early version of this paper has appeared in [25]. There we also looked at the connection between the π-calculus for component composition semantics and modal logics. In [25], we addressed the application of the framework to the Unified Modelling Language UML [26]. The connector idea is taken from [22]. In another paper [27], we have used a variant of the λ-calculus to define service requests and provisions using reduction as the mechanism for matching between services. The variant is called $\lambda\pi$-calculus [28] adding a flexible parameter (matching) concept to the λ-calculus. This $\lambda\pi$-calculus can be interpreted in object structures. We have used a π-calculus variant here, because it offers multiple (concurrent) connections and it allows to model two layers: contracts and connectors.

Walker [29] introduces object intercommunication into the π-calculus. In our approach the user (the client) is the active entity which initiates the establishment of the connections. In Walker's formalisation, the service provider also provides the communication channels. The service user acquires the contract channel, then acquires the interaction channels via the appropriate contract channels and finally uses the interaction channels to invoke methods of the service providers.

9. Conclusions

The suitability of the π-calculus for the definition of a component composition framework has recently been demonstrated, see for example the PhD-theses [30] and [31]. The idea of using a process calculus to model component composition has here been carried further by exploiting the similarity of mobility and evolution. We have developed a simple framework for the determination of effects of changes in composed systems. Our framework addresses in particular the problem of replacing single components in a system.

Mobility - the change of connection between components - is the key feature in the π-calculus. We have introduced evolution through replacement into our variant. This concept is somewhat different from mobility. Replacement is more fundamental since it is a meta-operation affecting the definition of the system (in particular the type system) under consideration. Replacement can cause mobility - the change of connections - as the result of the change in types. Replacement - as we have seen - can be introduced into the calculus, resulting in a dynamically typed calculus.

Transitions in the system based on the establishment of new connections occur in several variants. Reaction between two components is constrained by port types and channel types. Ports of different types match under different circumstances, formalised by a subtype relation. The reactions are transitions in a transition system, whose states reflect the state of composition that ports are in. Each port can pass through different composition stages - expressed by the life cycle for clients and providers.

Bisimulations and similar relations of equivalence between processes are essential concepts in the π-calculus for comparing processes. This theory could be applied in our context, if we would consider the replacement of systems of composed components. In that case, we would only be interested in the externally observable behaviour of these systems. Then, weak bisimilarity could be the tool to define and analyse replacements of compositions. However, our assumption here has been that single components are the unit of change. Consequently, we have based our replacement analysis mainly on the type system.

Some component technologies provide features for the discovery of services. This is an implicit process here, described by the contract rule. The overall model does not involve an intermediary. The CORBA framework for the interaction of remote objects is based on an object request broker. Here, this functionality is implicit. In an extension of the approach, an explicit broker could be considered. Another element of future work includes the exploration of the relationship of modal logics and the type system for our composition and replacement calculus.

References

[1] E.K. Nordhagen. *A Computational Framework for Verifying Object Component Substitutability.* PhD thesis, University of Oslo, November 1998.

[2] W. Weck. Inheritance Using Contracts & Object Composition. In *Proceedings 2nd International Workshop on Component-Oriented Programming WCOP'97.* Turku Center for Computer Science, General Publication No.5-97, Turku University, Finland, 1997.

[3] S. Cimato and P. Ciancarini. A Formal Approach to the Specification of Java Components. In B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Tech. Rep. 251, University of Hagen, 1999.

[4] B. Pierce and D. Sangiorgi. Typing and Subtyping in Mobile Processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[5] N. De Palma, P. Laumay, and L. Bellissard. Ensuring Dynamic Reconfiguration Consistency. In *Proceedings 6th Int. Workshop on Component-Oriented Programming WCOP2001*. http://research.microsoft.com/users/cszypers/events/, 2001.

[6] L. Tan, B. Esfandiari, and B. Pagurek. The SwapBox: A Test Container and a Framework for Hotswappable JavaBeans. In *Proceedings 6th Int. Workshop on Component-Oriented Programming WCOP2001*. http://research.microsoft.com/users/cszypers/events/, 2001.

[7] J.B. Warmer and A.G. Kleppe. *The Object Constraint Language – Precise Modeling With UML*. Addison-Wesley, 1998.

[8] G.T. Leavens and A.L. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[9] L.F. Andrade and J.L. Fiadero. Interconnecting Objects via Contracts. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[10] K. R. Apt. Ten Years of Hoare's Logic: A Survey – Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[11] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier Science Publishers, 1990.

[12] O. Nierstrasz and T.D. Meijler. Requirements for a Composition Language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-based Models and Languages for Concurrent Systems*, pages 147–161. Springer-Verlag, 1995.

[13] G.T. Leavens and M. Sitamaran. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[14] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes - parts I/II. *Information and Computation*, 100(1):1–77, 1992.

[15] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[16] D. Sangiorgi and D. Walker. *The π-calculus - A Theory of Mobile Processes*. Cambridge University Press, 2001.

[17] C. Morgan. *Programming from Specifications 2e*. Addison-Wesley, 1994.

[18] R.J.R. Back and J. von Wright. *The Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[19] A. Moorman Zaremski and J.M. Wing. Specification Matching of Software Components. In Gail E. Kaiser, editor, *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 6–17. ACM Software Engineering Notes 20(4), October 1995.

[20] M. Lumpe, J.-G. Schneider, O. Nierstrasz, and F. Achermann. Towards a Formal Composition Language. In G.T. Leavens and M. Sitamaran, editors, *Proceedings European Conference on Software Engineering ESEC'97*, pages 178–187. Springer-Verlag, 1997.

[21] M. Lumpe, F. Achermann, and O. Nierstrasz. A Formal Language for Composition. In G.T. Leavens and M. Sitamaran, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[22] D. D'Souza and A.C. Wills. *Objects, Components and Frameworks in UML: the Catalysis approach*. Addison-Wesley, 1998.

[23] M. Büchi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In *Proceedings 2nd International Workshop on Component-Oriented Programming WCOP '97*. Turku Center for Computer Science, General Publication No.5-97, Turku University, Finland, 1997.

[24] C. Atkinson, J. Bayer, O. Laitenberger, and J. Zettel. Component-Based Software Engineering: The KobrA Approach. In *Proc. International Workshop on Component-Based Software Engineering, Limerick, Ireland*. 2000. ICSE (International Conference on Software Engineering) Workshop.

[25] C. Pahl. Components, Contracts and Connectors for the Unified Modelling Language. In *Proc. Symposium Formal Methods Europe 2001, Berlin, Germany*. Springer-Verlag, LNCS-Series, 2001.

[26] H.-E. Eriksson and M. Penker. *UML Toolkit*. John Wiley & Sons, 1998.

[27] C. Pahl. Modal Logics for Reasoning about Object-based Component Composition. In *Proc. 4rd Irish Workshop on Formal Methods, July 2000, Maynooth, Ireland*. 2000.

[28] L.M.G. Feijs. The Calculus $\lambda\pi$. In *Algebraic Methods: Theory, Tools and Applications*, pages 307–328. Springer-Verlag, 1989.

[29] D. Walker. Objects in the π-Calculus. *Information and Computation*, 115:253–271, 1995.

[30] M. Lumpe. *A π-Calculus Based Approach for Software Composition*. PhD thesis, Universität Bern, Institut für Informatik und angewandte Mathematik, 1999.

[31] J.-G. Schneider. *Components, Scripts, and Glue*. PhD thesis, Universität Bern, Institut für Informatik und angewandte Mathematik, 1999.