A Formal Composition and Interaction Model for a Web Component Platform

Claus Pahl [1]

School of Computer Applications
Dublin City University
Dublin, Ireland

Abstract

A framework for components on the Web needs a formal model that captures essential concepts such as contractual information and service matching. We propose a typed π-calculus-based model for Web components that formalises an extension of the currently discussed Web Services framework. We address in particular activities in the stages of a component life cycle – such as matching, commitment, connection and interaction – that are part of the process that a component is involved in.

1 Introduction

The Web is evolving from a document-centred environment to a service-centred environment. The purpose of the Web Services framework [2] is to establish a distributed computing model for services on the Web. Web technologies including languages and protocols are used to provide a remote procedure call mechanism. The protocol shall be based on XML-messaging in order to achieve maximal interoperability.

We propose to extend Web Services to a formally defined Web components framework. Several framework and models exist that suggest an extension of the proposed Web services framework [4,5,11,12,14], but so far the formal aspects have been neglected. Service requests and service provision and their matching are integral aspects of component technology. Semantic description of services through contractual information is a necessity. A formal model for Web components based on a typed π-calculus [13] shall be discussed that provides clear semantics and that allows to support analysis and design tools.

[1] Email: cpahl@computing.dcu.ie

[2] We base our discussion of Web services on the WSDL definition (W3C note, 2001), SOAP version 1.2, and UDDI version 2.0.

This work is based on two previous papers. In [10] we have presented basics of our formal framework. In [11] we have discussed requirements for a formally defined Web component framework. This work applies and extends results from both sources. The main novelty of our work is the consideration of Web component life cycles – important to describe business processes, interactions and workflow aspects. So far, this is a major limitation in component frameworks. Only a few papers have addressed this problem theoretically [9].

We outline a Web component architecture in Section 2. The description of services and aspects of a type system formalising them is dealt with in Section 3. Matching and interaction are key activities – their semantics in form of operational process descriptions is investigated in Section 4. Another key element in a Web component framework is a protocol capturing the various activities, see Section 5. We end with related work and some conclusions.

2 Web Component Architecture

An architecture for Web components should consist of a *description language* for semantic component descriptions, a *matching and interaction protocol* implementing 2-phase (or 2-layered) composition, and a *set of services* including discovery, matching, configuration, and interaction. Such an architecture would describe a Web-based component middleware platform. Description languages and protocols omit details about how components are discovered, how they are stored and made available. This can be supported by special services, such as a broker service. A number of services will depend on the semantic formalism made available through the description language.

The *composition architecture* shall be *layered*. We distinguish a matching layer and an interaction layer. Connections for interactions are established after successful matching. These connections are needed for service activation and service reply. This architecture is a reflection of the component life cycle. The *component life cycle* – matching before interaction – needs to be formalised by a *composition protocol*. This affects each component in isolation, but also the composition of components. Protocol constraints can be expressed by appropriate transition rules.

The *type system* and in particular *subtypes* can play a major role. Subtypes can determine what a suitable match for a service request might be. The classical definition of a subtype [16] – an instance of a subtype can always be used in any context in which an instance of a supertype was expected – can formulate the essence of consistent matching between component services.

Ports are abstract access points to component services. Port descriptions are part of component interfaces. *Port types* can reflect various properties, e.g. the port orientation (input or output), the role (is the port involved in matching components or in the interaction of components), or the transport capacity. Port types can be used to express structural and behavioural constraints. A protocol endpoint is actually a family of ports with different roles.

3 Description of Services

3.1 Description Languages

Web services without semantical information can be described by the *Web Services Description Language WSDL*. A Web service description consists of five sections in two parts.

- An *abstract* protocol-independent part consists of type, data and operation descriptions. The operation part, called 'portType', describes operations that implement the service functionality in terms of its typed input and output parameters. These parameters are described in a data part, called 'message'. Types for the messages can be defined in a 'types' section.
- The 'binding' to a specific protocol is one of the two sections of the *concrete* part of the service description. It describes how a service is activated using the protocol under consideration. The final section is called 'service'; it links the service to a particular location where the service can be found. The protocol determines the format to be used to activate a Web service.

Single services could be grouped into components. We suggest a *Web Components Specification Language (WCSL)*. We will motivate this language by a schematic example following the structure of the WSDL. The purpose of WCSL is similar to WSDL, except that we expect automation to play an important role in the processing of WCSL descriptions. Formal semantics will be given based on a typed π-calculus variant. Components are syntactically characterised by an interface with service signatures, separated into import and export elements. The type system will capture the semantical properties of Web services and components.

3.2 Data Elements and their Types

The entities in a Web composition system are data elements, ports and components. Data elements are characterised by the usual value domains as types. WSDL suggests the following notation for these elements, allowing basic and structured types to be defined:

```
<element name="dataType">
     <complexType>
          <all> <element name="aNumber" type="int"/> </all>
     </complexType>
</element>
```

Basic and complex data types shall be assumed, but not explicitly specified. We also assume a connector type representing connections between ports.

Data elements and connectors can be assembled into messages. Two sample messages shall be defined – containing a data item and a connection:

```
<message name="InData">
```

T	$::= B$	Basic type
	$\mid L$	Link type
	$\mid \textsc{Sig}(T \times \ldots \times T \times L)$	Signature
	$\mid \textsc{Prd}(T)$	Predicate
L	$::= P\ C$	Port and channel type
P	$::= \pm\ (\textsc{Req} \mid \textsc{Pro} \mid \textsc{Inv} \mid \textsc{Exe} \mid \textsc{Rec} \mid \textsc{Rep})$	Port type
C	$::= \textsc{Ctr}(T \times T \times T)$	Contract
	$\mid \textsc{CAc}(T \times \ldots \times T \times L)$	Connector activation
	$\mid \textsc{CRe}(T)$	Connector reply

Fig. 1. Type Language Syntax.

```
        <part name="body" element="dataType"/>       </message>
<message name="serv_I">
        <part name="body" element="connectorType"/>  </message>
```

3.3 Type Language Syntax

The type system plays a key role in our composition and interaction model. A typing context Γ is a finite set of bindings – mappings from names to types. Three types of judgments shall be 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 1. The constructors Ctr, CAc, and CRe are the link-type constructors. Their purpose is to classify channels based on the data that is transferred along them. We leave the set of basic value types unspecified. We assume that there is at least one basic type B. The XML Schema framework [3] provides the setting to define basic and structured types for Web services and Web components. Sig and Prd are standard constructors for service signatures and predicates; the other type constructors are specific to the component context.

3.4 Ports and their Types

The most important entities are the ports, which represent services. Port types define the services based on input and output messages. We extend the

WSDL port type specification by contractual information:

```
<portType name="serv">
  <operationContract  name="serv_C" precon="pre" postcon="post">
      <input  message="serv_I" />           </operationContract>
  <operationConnector name="serv_I">
      <input  message="InData"  />
      <output message="OutData" />
      <reply  message="serv_R"  />         </operationConnector>
</portType>
```

Each port *serv* is essentially a family of ports $serv = (serv_C, serv_I, serv_R)$. The first port $serv_C$ is the *contract port*, representing an abstract interface described by a signature, a precondition and a postcondition. $serv_I$ and $serv_R$ are *connector ports* – $serv_I$ handles the service invocation and input and $serv_R$ handles the service output. $serv_I$ is the connector activation (or interaction) port. The port $serv_R$ carries the reply from the service invocation. We distinguish a port type and a channel type for each port:

- *Port types* describe the functionality of a port within the component (e.g. contract or connector port) and its orientation (in- or out-port). Port types are referred to by $\mathcal{T}_p(serv)$ or $serv :_p t$ for port *serv*, e.g. $\mathcal{T}_p(\overline{serv_C}) = \textsc{Req}$ and $\mathcal{T}_p(serv'_C) = \textsc{Pro}$ are requestor and provider ports. Each port has also an orientation, called the *polarity*. Contract and connector activation ports are output ports ('+' : the port can only send) and the reply port is an input port ('−' : the port can only receive) for the service client.
- *Channel types* for a port $serv = (serv_C, serv_I, serv_R)$ describe the expected capacity, i.e. what kind of entities can be transported: $serv_C :_C \textsc{Ctr}(\textsc{Sig}(T_1, \ldots, T_n, +\textsc{CRe}(T)), \textsc{Prd}(\textsc{pre}), \textsc{Prd}(\textsc{post}))$ for contract ports, $serv_I :_C \textsc{CAc}(T_1, \ldots, T_n, +\textsc{CRe}(T))$ for connector ports, and $serv_R :_C \textsc{CRe}(T)$ for reply ports. Channel types 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 t are denoted by $\mathcal{T}_c(serv)$ or $serv :_c t$ for port *serv*.

A contract consists of a service signature, a pre- and a postcondition. Connectors when transferred on channels have to satisfy a contract type. On connector activation ports, data values and a reply channel can be transferred; on connector reply ports only data can be transferred. The key criteria for matching, i.e. the successful connection of two components through a connector, are contracts (this will be explained in Section 4). 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 reflects the fact that parameters are passed, and possibly a result has to be transferred back on a channel with a different capacity T. Pre- and postconditions are formed using the predicate type constructor $\textsc{Prd}$.

4 Semantics of Matching and Interaction

The concrete part of WSDL concerns the protocol binding and association of the location for Web services, preparing for service activation. The infrastructure for Web service activation and reply can be provided by the SOAP protocol [3]. SOAP – the *Simple Object Access Protocol* – is an XML-based protocol for service invocations and replies designed to support remote activations of services specified in WSDL. The discovery of services is supported by a directory framework UDDI – *Universal Description, Discovery and Integration.* UDDI acts as a marketplace for services or components.

Matching of services and the interaction between services and components are the key activities. The introduction of semantic service descriptions requires to pay more attention to the problem of matching required and provided services before a connections is established and components interact. The binding part of our suggested WCSL needs to separate matching binding and interaction binding. The latter needs to address activation and reply.

4.1 Subtypes and Matching

Subtyping $S \leq T$ shall be used to define matching of services and components. A subtype concept goes beyond the basic and structured types provided by the WSDL types section. A subtype relation between ports determines whether two ports that represent services match. Channel types of contract ports are contracts consisting of a service signature, a precondition and a postcondition. For a service request $m_C :_c \textsc{Ctr}(Sig,Pre,Post)$ and a provided service $n_C :_c \textsc{Ctr}(Sig',Pre',Post')$, we say that n_C matches m_C, or $n_C \leq m_C$, if $Sig = Sig' \wedge Pre \rightarrow Pre' \wedge Post' \rightarrow Post$ [4]. This is the combination of two classical refinement relations (weaken the precondition and strengthen the postcondition) from the Refinement Calculus [1,8].

The semantics of the type system can be defined by typing rules for basic types, type constructors, subtypes and process expressions – see Figure 2. Typing rules for the type constructors (contract, connector, signature, predicate) are omitted, except for the one for contracts, I-Ctr. If 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 type $\textsc{Ctr}(\textsc{Sig}(T_1, \ldots, T_n, \textsc{CRe}(T)), \textsc{Prd}(F_1), \textsc{Prd}(F_2))$. Two structural rules contribute to the definition of the subtype relation $\leq$ as a preorder: the reflexivity rule S-Refl and the transitivity rule S-Trans:

$$[\text{S-Refl}]\ \frac{S =_\beta T}{\Gamma \vdash S \leq T} \qquad [\text{S-Trans}]\ \frac{\Gamma \vdash S \leq T \quad \Gamma \vdash T \leq U}{\Gamma \vdash S \leq U}$$

The subtyping rules for signatures and predicates are S-Sig and S-Prd. The names *Cond, Pre, Post, Sig* and their primed variants are type variables. A

[3] SOAP might influence the standardisation of the XML Protocol [3] currently in progress.
[4] Variants providing more flexibility, e.g. signature inclusion, can certainly be considered.

$$[\text{I-Ctr}]\ \frac{\Gamma \vdash s :_c \textsc{Sig}(T_1, \ldots, T_n, \textsc{CRe}(T))\ \ \Gamma \vdash p_1 :_c \textsc{Prd}(F_1)\ \ \Gamma \vdash p_2 :_c \textsc{Prd}(F_2)}{\Gamma \vdash \textsc{Ctr}(s, p_1, p_2) :_c \textsc{Ctr}(\textsc{Sig}(T_1, \ldots, T_n, \textsc{CRe}(T)), \textsc{Prd}(F_1), \textsc{Prd}(F_2))}$$

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

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

$$[\text{S-Ctr}]\ \frac{\Gamma \vdash \mathit{Pre} \leq \mathit{Pre'}\ \ \Gamma \vdash \mathit{Post'} \leq \mathit{Post}\ \ \Gamma \vdash \mathit{Sig'} \leq \mathit{Sig}}{\Gamma \vdash \textsc{Ctr}(\mathit{Sig'}, \mathit{Pre'}, \mathit{Post'}) \leq \textsc{Ctr}(\mathit{Sig}, \mathit{Pre}, \mathit{Post})}$$

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

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

Fig. 2. Typing rules.

condition is subtype of another if it implies it: *Cond* $\leq$ *Cond'* if *Cond* $\rightarrow$ *Cond'*. A contract forms a subtype of another if its precondition is weakened and its postcondition is strengthened, see S-Ctr. 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.

4.2 Component Composition

The development of a notation describing the *process* of component composition based on matching and interaction is the next step. We use a typed π-calculus to define Web component matching and interaction behaviour.

The syntax of composition expressions P involving action prefixes π_i is:

$$P ::= \nu m\ P \mid P_1 | P_2 \mid\ !P \mid \Sigma_{i \in I}\, \pi_i.P_i \mid 0$$

Restriction $\nu m\ P$ means that m is only visible in P. Summation $\pi_i.P_i$ means that one action prefix π_i is chosen and the process transfers to state P_i. Iteration $!P$ means that the process is executed an arbitrary number of times. We also need abstractions, i.e. defining equations of the form $A(a) = P_A$ [5]. This

[5] 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.

follows the presentation of the π-calculus in [7].

The basic element describing activity in the π-calculus are actions [13]. Actions are combined to process expressions. Actions are expressed as prefixes π to the process expressions: $\pi ::= \textsc{Ptype}\ \overline{x}\langle y\rangle \mid \textsc{Ptype}\ x(y) \mid \tau$. Actions can be divided into output actions $\overline{x}\langle y\rangle$ (the name y is sent along channel/port x), input $x(y)$ (i.e. y is received along x), and a silent non-observable action τ. We have annotated the action prefixes π by port types Ptype, which explain the role of the port with respect to component life cycle activities such as service request or service invocation:

$$
\begin{array}{lcll}
\pi ::= & \textsc{Req}\ \overline{m_C}\langle m_I\rangle & + & \text{Request} \\
& \textsc{Pro}\ n_C(n_I) & - & \text{Provide} \\
& \textsc{Inv}\ \overline{m_I}\langle a_1, \ldots a_l, m_R\rangle & + & \text{Invoke} \\
& \textsc{Exe}\ n_I(x_1, \ldots, x_k, n_R) & - & \text{Execute} \\
& \textsc{Rep}\ \overline{n_R}\langle b\rangle & + & \text{Reply} \\
& \textsc{Res}\ m_R(y) & - & \text{Result}
\end{array}
$$

The operational semantics of the notation, in particular the two main forms of composition *matching* and *interaction*, shall now be discussed.

4.3 Matching and Connection

Matching and connector establishment are two different activities in the Web services framework. We can distinguish

(i) a *commitment phase* where both components try to form a contract; or, more technically, try to work out and agree on the necessary channel capacity for interaction. UDDI provides the basic infrastructure.

(ii) a *connector establishment phase*, or *connection phase*, where an interaction channel (a connector) is established for later interaction, i.e. activation of remote services. SOAP is the communication infrastructure.

We will formalise these activities in form of transition rules.

A key feature in a Web component framework is an agent or broker to match and to prepare the connection of services. UDDI is a service that allows providers to publish their services and requestors to enquire about suitable services. UDDI provides two APIs, the Inquiry API and the Publisher's API, in order to automate the process of matching required and provided services. Services can be grouped into a UDDI business-service structure, a container for services resembling a component. We suggest to extend this feature to components including contractual descriptions. Two services match if their contract types form a subtype relationship. A subtype relationship can result in a commitment, which is a prerequisite for the establishment of a connection.

For a composition expression $\overline{m_C}\langle m_I\rangle.C|n_C(n_I).P$ we can say that both processes commit themselves to a communication along the channel between ports m_C and n_C, if their contracts match. The **contract rule** [T-Ctr] formalising the process of matching and commitment is defined as follows:

$$\frac{\textsc{Req}\ \overline{m_C}\nu c\langle m_I\rangle.C \xrightarrow{\overline{m_C}\nu c\langle m_I\rangle} \nu c\langle m_I\rangle.C \quad \textsc{Pro}\ n_C(n_I).P \xrightarrow{n_C(n_I)} (n_I).P}{\textsc{Req}\ \overline{m_C}\nu c\langle m_I\rangle.C+M_1|\textsc{Pro}\ n_C(n_I).P+M_2 \xrightarrow{\tau} (n_I).P@\nu c\langle m_I\rangle.C}\langle\ t_{n_C} \leq t_{m_C}$$

The annotations Req and Pro denote port types, i.e. $\overline{m_C} :_p \textsc{Req}$ and $n_C :_p \textsc{Pro}$. 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. The matching is also guarded by the channel type constraint $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)$.

The contract rule differs from the original π-calculus reaction rule which requires channel names to be the same [7,13]. We only require a subtype relationship between ports. Type systems for the π-calculus usually constrain data that is sent; here we constrain reaction, i.e. the interaction between agents. The receiver can *accept* an input based on the type, not the name. The contract rule cannot be translated into the match-rule found in some π-calculus variants. The contract rule is, however, 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 [13]. We have chosen to introduce a fresh variable c instead.

Service descriptions that have been matched using UDDI features can result in connected and interacting components. Each service description describes the interface of the service and how to connect to it. A binding template contains the information to actually invoke the service. In order to support connector establishment after commitment, UDDI specifications include an XML schema for SOAP messages.

The commitment of two matching services m_C and n_C leaves two residues: $\langle m_I\rangle.C$ is called *concretion* and $(n_I).P$ is called *abstraction*, see [13] 4.3. A restricted concretion $\nu c\langle m\rangle.C$ can be introduced. Concretion and abstraction together result in a reaction, expressed by a construct that we call **connector establishment** [6]: $(n_I).P@\nu c\langle m_I\rangle.C \stackrel{\text{def}}{=} \nu c(\{c/m_I\}C|\{c/n_I\}P)$ [7] which shall be abbreviated by a binding $C \frown P$. The connection yields a proper process describing the establishment of a connector c. The binding $C \frown P$ introduces the connector c, a fresh variable free in C and P. The connector c is a private (restricted) channel. The concrete part of a WSDL specification describes bindings – information necessary for connector establishments:

```
<binding name="portSOAPbinding" type="port">
  <soap:binding style="document" transport="..."/>
  <operation name="port">
```

[6] Usually called *application* in the literature, see [7] Chapter 12.1.
[7] The substitution $\{b/a\}P$ means that b replaces a in P.

```
    <soap:operation soapAction="http://www. .. .com/../serv"/>
    <input>    <soap:body use="literal" />    </input>
    <output>   <soap:body use="literal" />   </output>
   </operation>
</binding>
```

Our connector establishment implements the UDDI invocation model where a binding template is cached by the service user and used at a later stage to invoke the remote service.

4.4 Interaction

UDDI- and WSDL-bindings provide basic connector descriptions. The actual implementation of binding and interaction (connector activation and reply) is realised using e.g. SOAP. Here is the SOAP connector activiation – part of a SOAP envelope – for service `serv` with input data and reply channel:

```
<soap:operation soapAction="http://www. .. .com/../serv"/>
<soap-env:body>
    <port service="http://www. .. .com/../serv">
         <InData>  a    </InData>
         <Reply>   m_R  </Reply>
    </port>
</soap-env:body>
```

We assume that a private channel – the connector representing the SOAP connection `serv` – has been established between client and provider. Such a channel is used if a client requesting m_I is 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 sent to the provider in form of messages.

$$\frac{\textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle.C \xrightarrow{\overline{m_I}\langle a,m_R\rangle} C \quad \textsc{Exe}\ n_I(x, n_R).P \xrightarrow{n_I(x,n_R)} P}{\textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle.C + M_1 | \textsc{Exe}\ n_I(x, n_R).P + M_2 \xrightarrow{\tau} C \frown \{a/x\}P} \langle\ t_{n_I} \leq t_{m_I}$$

is the **connector activation rule** [T-CAC]. Types t_{m_I} and t_{n_I} represent 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 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 been successfully executed. A protocol, specified in form of a component life cycle, can guarantee this.

Finally, the **connector reply rule** [T-CRE] gives semantics to a SOAP reply:

$$\frac{\textsc{Res}\ m_R(y).C \xrightarrow{m_R(y)} C \quad \textsc{Rep}\ \overline{n_R}\langle b\rangle.P \xrightarrow{\overline{n_R}\langle b\rangle} P}{\textsc{Res}\ m_R(y).C + M_1 | \textsc{Rep}\ \overline{n_R}\langle b\rangle.P + M_2 \xrightarrow{\tau} \{b/y\}C \frown P} \langle\ t_{n_R} \leq t_{m_R}$$

We assume $t_b \leq t_y$. Here, b is the result of the internal computation triggered by the activation of P. We have decided to formulate the reply in a separate rule, and not to address the creation of a private reply channel replacing m_R and n_R within the connector activiation rule. The typing constraint that RES- and REP-ports have to match is more explicit in this form.

4.5 Type Safety

Type safety concerns the relation between the type system and the operational semantics. The operational semantics is defined in a transitional form, 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.

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. Connector type $T_I = \text{CAC}(T_1, \ldots, T_n, \text{CRE}(T))$ **satisfies** contract type $T_C = \text{CTR}(\mathit{Sig}, \mathit{Pre}, \mathit{Post})$, or $T_I \models T_C$, if for a service port p the connector port p_I satisfies the following constraints: $\text{SIG}(T_1, \ldots, T_n, \text{CRE}(T)) = \mathit{Sig}$ 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 types.

We can now define **well-typedness** of **simple actions** [W-ACT]:

- $\Gamma \vdash \text{REQ}\ \overline{m_C}\langle m_I \rangle$ if $\mathcal{T}_c(m_I) \models \mathcal{T}_c(m_C)$, otherwise $\text{REQ}\ \overline{m_C}\langle m_I \rangle$ fails.
- $\Gamma \vdash \text{PRO}\ n_C(n_I)$ if $\mathcal{T}_c(n_I) \models \mathcal{T}_c(n_C)$, otherwise $\text{PRO}\ \ n_C(n_I)$ fails.
- $\Gamma \vdash \text{INV}\ \overline{m_I}\langle a, m_R \rangle$ if $\mathit{type}(a), \mathcal{T}_c(m_R) \models \mathcal{T}_c(m_I)$, otherwise $\text{INV}\ \overline{m_I}\langle a, m_R \rangle$ fails.
- $\Gamma \vdash \text{EXE}\ n_I(y, n_R)$ if $\mathit{type}(y), \mathcal{T}_c(n_R) \models \mathcal{T}_c(n_I)$, otherwise $\text{EXE}\ n_I(y, n_R)$ fails.

The execution of an action fails, if 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. If $\text{REQ}\ \overline{m_C}\langle m_I \rangle$ and $\text{PRO}\ n_C(n_I)$ are well-typed, but do not satisfy the subtype constraint $\mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)$, then $\text{REQ}\ \overline{m_C}\langle m_I \rangle | \text{PRO}\ n_C(n_I)$ fails. The **well-typedness** of **parallel compositions** is defined by rule [W-PARCOMP]:

$$\frac{\Gamma \vdash \text{REQ}\ \overline{m_C}\langle m_I \rangle \quad \Gamma \vdash \text{PRO}\ n_C(n_I) \quad \Gamma \vdash \mathcal{T}_c(n_C) \leq \mathcal{T}_c(m_C)}{\Gamma \vdash \text{REQ}\ \overline{m_C}\langle m_I \rangle | \text{PRO}\ n_C(n_I)}$$

Well-typedness guarantees correct composition and interaction behaviour according to the specifications given through the type system.

Based on these constructions, we can obtain the following safety properties, presented here without proof:

(i) Substitution lemma: if $\Gamma \vdash C$ and $\Gamma \vdash x : T, v : T$, then $\Gamma \vdash \{v/x\}C$.

(ii) Evaluation cannot fail in well-typed programs: if $\Gamma \vdash C$ then the execution of C does not fail.

(iii) Transition preserves typing: if $\Gamma \vdash C_1$ and $C_1 \rightarrow C_2$ then $\Gamma \vdash C_2$.

5 A Component Composition and Interaction Protocol

In the previous sections, we have seen several stages in the life cycle of a component such as service matching, connector establishment, or service invocation. The full life cycle of clients, providers, and systems consisting of both clients and providers can be specified in a standard form. This standard form formalises a *component composition and interaction protocol*. The behaviour of components is a key element in the description of Web services. However, a corresponding construct does not exist for the Web services platform.

Clients are parameterised by a list of required services. Requests have to be satisfied before any interaction can happen. Once a connection is established, a service can be used several times. All service requests need to be satisfied – expressed by the parallel composition of the individual ports:

$$C_i(m_1,\ldots,m_l) \stackrel{\text{def}}{=} \textsc{Req}\ \overline{m_C^1}\langle m_I^1\rangle.!(\textsc{Inv}\ \overline{m_I^1}\langle a^1, m_R^1\rangle.\textsc{Res}\ m_R^1(y^1).0) \mid \ldots \mid \textsc{Req}\ \overline{m_C^l}\langle m_I^l\rangle.!(\textsc{Inv}\ \overline{m_I^l}\langle a^l, m_R^l\rangle.\textsc{Res}\ m_R^l(y^l).0)$$

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

$$P(n_1,\ldots,n_k) \stackrel{\text{def}}{=} !(\ \textsc{Pro}\ n_C^1(n_I^1).!(\textsc{Exe}\ n_I^1(y^1, n_R^1).\textsc{Rep}\ \overline{n_R^1}\langle b\rangle.0) + \ldots + \textsc{Pro}\ n_C^k(n_I^k).!(\textsc{Exe}\ n_I^k(y^k, n_R^k).\textsc{Rep}\ \overline{n_R^k}\langle b\rangle.0)\ )$$

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 to form a composed system:

$$CS \stackrel{\text{def}}{=} C_1(m_{1_1},\ldots,m_{1_{m_1}}) \mid \ldots \mid C_j(m_{j_1},\ldots m_{j_{m_j}}) \mid P(n_1,\ldots,n_k)$$

A component can be both client and provider, i.e. can import and export services:

$$CS \stackrel{\text{def}}{=} (\textsc{Req}\ m_C^1\langle m_I^1\rangle.0 | \ldots | \textsc{Req}\ m_C^l\langle m_I^l\rangle.0).\ !(\ !(\ \textsc{Inv}\ \overline{m_I^1}\langle..\rangle.\textsc{Rec}\ m_R^1(..).0\ + \ldots + \textsc{Inv}\ \overline{m_I^l}\langle..\rangle.\textsc{Rec}\ m_R^l(..).0\ ) + P(n_1,\ldots,n_l)\ )$$

The requirements have to be satisfied, i.e. connectors have to be established,

before any service can be provided. A service that is provided and actually invoked can then trigger the invocation of imported services.

The usage of the operations could be expressed in our WCSL in form of a component life cycle – here a client requesting a service and subsequently interacting with the service repeatedly:

```
<sequence>
    <request name="serv_C" precon="pre" postcon="post" />
    <repeat>
         <sequence>
               <invoke  name="serv_I"> ... </invoke>
               <receive name="serv_R"> ... </receive>
         </sequence>
   </repeat>
</sequence>
```

The semantics of this protocol client expression is

$$C(serv) \stackrel{\text{def}}{=} \textsc{Req}\ \overline{serv_C}\langle serv_I\rangle.\ !(\ \textsc{Inv}\ \overline{serv_I}\langle a, serv_R\rangle.\textsc{Res}\ serv_R(y).0\ )$$

which satisfies the client standard form C_i that has been presented above.

6 Related Work

A formally defined computing model for Web components is essential if analysis and reasoning services based on semantic descriptions shall be provided. Suitable frameworks for the formulation of this model are process calculi with typing, mobility, security, etc., e.g. the π-calculus [13] or the Ambient calculus [2]. In [10], we have presented a formal framework for component composition based on a typed π-calculus, which satisfies the requirements outlined above. Typed process models to formalise interaction between components, or objects, have also been used elsewhere. Nierstrasz [9] develops a formal type-theoretic framework for objects. Objects are characerised as regular processes that interact with each other. A two-layered type system distinguishes services types (contracts) and regular types (protocols). Two subtype notions – based on services types and regular types – define a notion of satisfiability between client and provider. Nierstrasz emphasises the orthogonality of the two different forms of types.

Some frameworks for advanced services architectures on the Web are already proposed. In [4], a component model underlying the Web services [15] platform is identified. It is admitted that strenghtening the component aspects will greatly improve the platform. Fensel and Bussler [5] present a platform for Web-based service, called *Web Services Modelling Framework* (WSMF). The development of the framework focussing on the integration of semantic Web technology is in progress – a formal semantics does currently not exist. The issue of composed Web services is addressed in [6]. Business processes

and interactions are the two types of processes that result in the composition of services. Service provider and requester are considered as in our approach. However, these approaches have not included proper components.

Some groups have addressed Web component broker systems. Among those are the Cell-project [12] and the ComponentXchange [14]. The former implements a two-layered system for component composition. The latter focusses on matching activities – there called trading. In [11] we have briefly described our own attempts to implement a component broker.

7 Conclusions

Web Services, which provide a remote procedure call (RPC) environment, should be seen as a first step towards a component middleware platform for the Web. Component technology for the Web, however, requires a rigorous underlying model. Our typed π-calculus-based operational semantics provides the foundation for various necessary features of Web component middleware – we have, for instance, discussed replacement issues in [11].

We have identified and formalised matching, commitment, connection and interaction as core services of component middleware. Their embedding into a component life cycle framework is essential. Component technology emphasises reuse and maintenance in the context of change and evolution. The π-calculus is an ideal formal framework to develop a life cycle-based approach to describe the process a component might be involved in. We have used the standard π-calculus. However, aspects such as internal mobility – the use of private names in a communication – suggests to consider other calculus forms. The private and the localised π-calculus [13] shall be investigated in search for a more suitable foundation in the future.

This presentation motivates a component middleware platform for the Web. Questions relating to particular services such as those offered by the CORBA platform for object-based middleware still need to be answered. We have addressed aspects relating to trading and life cycle services, however, others such as security or transactions still need to be looked at.

The ultimate goal of this research is a framework for the development and management of Web components. This would require modifications to the current Web services model. Work on the DAML-S services descriptions indicates the direction. In contrast to recent work on DAML-S, our work could provide a formal foundation. An integration of contracts is an essential element of these modifications. The notion of contracts, however, needs to be extended from request-response type interaction to more complex interaction patterns.

References

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

[2] L. Cardelli and A.D. Gordon. Mobile Ambients. In *Proceedings FoSSaCS'98*, pages 140–155. Springer Verlag, 1998.

[3] W3C World Wide Web Consortium. Extensible Markup Language (XML), 2001. http://www.w3.org/XML.

[4] F. Curbera, N. Mukhi, and S. Weerawarana. On the Emergence of a Web Services Component Model. In *Proceedings 6th Int. Workshop on Component-Oriented Programming WCOP2001*. http://research.microsoft.com/users/cszypers/events/, 2001.

[5] D. Fensel and C. Bussler. The Web Services Modeling Framework. Technical report, Vrije Universiteit Amsterdam, 2002.

[6] F. Leymann. Web Services Flow Language (WSFL 1.0), 2001. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

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

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

[9] Oscar Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 1–15, October 1993.

[10] C. Pahl. A Pi-Calculus based Framework for the Composition and Replacement of Components. In *Proc. OOPSLA Workshop on Specification and Verification of Component-Based Systems*, 2001.

[11] C. Pahl and D. Ward. Towards a Component Composition and Interaction Architecture for the Web. In *Proc. ETAPS Workshop on Software Composition SC2002*. Elsevier, ENTCS Series, 2002.

[12] R. Rinat and S.F. Smith. The Cell Project: Component Technology for the Internet. In *Proceedings 6th Int. Workshop on Component-Oriented Programming WCOP2001*. http://research.microsoft.com/users/cszypers/events/, 2001.

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

[14] V. Sriram, A. Kumar, D. Gupta, and P. Jalote. ComponentXchange: A Software Component Marketplace on the Internet. In *Proceedings 10th Int. Conference on the World-Wide Web WWW10*. International World-Wide Web Conference Consortium IW3C2, 2001.

[15] V. Vasudevan. A Web Services Primer, 2001. http://www.xml.com/pub/a/2001/04/04/webservices.

[16] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *ACM OOPS Messenger*, pages 8–87, 1990.