Journal of Universal Computer Science, vol. 14, no. 8 (2008), 1285-1313
submitted: 15/11/07, accepted: 25/3/08, appeared: 28/4/08

Mismatch Avoidance in Web Services Software Architectures

Cristina Gacek
(University of Newcastle, United Kingdom
cristina.gacek@newcastle.ac.uk)

Carl Gamble
(University of Newcastle, United Kingdom
c.j.gamble@newcastle.ac.uk)

Abstract: Architectural mismatches are a recognized obstacle to successful software reuse. An architectural mismatch occurs when two or more software components are connected to form a system and those components make differing and incompatible assumptions about their interactions or the environment in which they exist.

Mismatch detection and avoidance has been previously discussed in existing literature. These typically take the form of generic rules and guidelines. Service Oriented Architectures (SOA) are becoming one of the main trends in the current engineering of software. Using web services, as defined by W3C Web Services Architecture Working Group, supports the engineering of SOA by providing rules and restrictions that apply to the definition of web services and how they can interact with other components to form a larger system. We see this as an opportunity to define a web services style with corresponding rules to avoid the introduction of architectural mismatches at design time.

In this paper we describe the development of an environment which supports SOA development by enabling their description, as well as facilitating the detection of potential mismatches between web services. Here we define a web services style in the architectural description language ACME & Armani, and present the environment that we developed in ACME Studio using our web services style definition. This is accompanied by a small case study illustrating the use of our environment.

Key Words: software architecture, architectural style, architectural mismatch, acme, web services

Category: D.2.11, D.2.12

1 Introduction

The practice of software construction in a component-based fashion heavily based on software components reuse has long been recognized as an important solution for the software crisis [McIlroy 1969]. It is a powerful means of not only reducing software development costs in the long run, but also reducing the risk of project failure, improving software quality, shortening development time, and greatly increasing the productivity of the individual software developer [Gacek 1998]. This vision is still to fully become a reality. Obstacles to date have ranged from various organisational to technical barriers. Technical barriers include the occurrence of architectural mismatches during systems' composition from various independent software parts.

An architectural mismatch occurs when two or more software components are connected to form a system, and those components make differing and incompatible assumptions about their interactions or the environment in which they exist. The presence of an architectural mismatch between composing elements within a system can hinder reuse in a variety of ways. Problems can range from preventing elements' composition altogether to experiencing undesired side effects at run-time. Hence, architectural mismatches must be handled appropriately, by either being avoided during development and/or system reconfiguration, or being tolerated at run time.

Mismatch detection and avoidance has been previously discussed in existing literature [Abd-Allah 1996] [Bhuta and Boehm 2007] [Davis et al. 2002] [DeLine 1999] [DeLine 2001] [Egyed et al. 2000] [Hepner et al. 2006] [Uchitel and Yankelevich 2000] [Yakimovich et al. 1999]. These works typically take the form of generic rules and guidelines. The approach discussed in this paper also adopts this line of work. We believe that architectural styles have much to offer in this respect: they provide a vocabulary of architectural elements; parameters for the architect to follow; and constraints to check the validity of the individually chosen attribute values, as well as the overall system configuration.

Service-Oriented Architectures (SOAs) are becoming one of the main trends in the current engineering of software. Web services are a recent approach towards supporting SOAs, building from standards agreed upon by various community stakeholders, while avoiding proprietary middleware solutions. Put simply, a Web service is any system that provides a network interface that is described by a published WSDL [W3C 2006c][W3C 2006d] [W3C 2006e] [W3C 2006f] document and uses SOAP [W3C 2006a] as its message format. In this respect it is fair to term Web services as being an integration middleware [Baker 2002] or standard for presenting the interface parts of SOA [Ferguson and Stockton 2005] [Behr 2003]. Hence, using web services, as defined by W3C Web Services Architecture Working Group [W3C 2006b], supports the engineering of SOAs by providing rules and restrictions that apply to the definition of web services and how they can interact with other components to form a larger system. We see this as an opportunity to define a web services style with corresponding rules to avoid the introduction of architectural mismatches during system design.

In this paper we describe the development of an environment which supports SOA development by enabling their description, as well as facilitating the detection of potential mismatches when using components that are web services. First we investigate the characteristics of web services informally, then summarise the findings before showing how the resulting elements and constraints were used to define a web services style in the architectural description language ACME &

Armani. We then present the environment that we developed in ACME Studio using our web services style definition. This is accompanied by a small case study illustrating the use of our environment.

2 Background

Software architectures represent the high-level design of a software system. They provide critical abstractions with which it is possible to reason about and describe the structure and behaviour of a system[1]. They are typically described in terms of their components, connectors, configurations, and their corresponding constraints. Components are the loci of computation and storage; connectors represent the conduits through which control and/or data flow; and configurations imply how the components and connectors are composed in order to form a software system. Other basic concepts relevant for the actual description of a software architecture are ports and roles. Ports represent the interfaces of components, whereas roles are the points at which connectors attach to ports such that control and/or data can flow from one component to another. As a set of theses elements is attached and given properties it forms a system configuration that can then be analysed.

Architectural mismatches prevent the successful integration of components to form a system. Architectural mismatches were first discussed by Garlan et al., when they introduced the term [Garlan et al. 1995]. In this seminal paper, the authors presented some specific problems experienced while building a specific system. Since then several works have focused on trying to avoid mismatches [Abd-Allah 1996] [Bhuta and Boehm 2007] [Davis et al. 2002] [DeLine 2001] [Egyed et al. 2000] [Uchitel and Yankelevich 2000]. The two main threads for doing this have been in terms of either characterizing components and connectors to be used during system composition and analyzing for mismatches using those characteristics; or using architectural styles as a factor guiding to possible areas where mismatches could occur. The latter build on the former by restricting some of the choices available to components and connectors, such that the types of mismatches that can occur during system composition are drastically reduced, facilitating analysis and their detection.

There are several related works in the area of classification of components and connectors. Abd-Allah [Abd-Allah 1996] and Gacek [Gacek 1998] made use of conceptual features for individual components whose choices may undermine system composition by introducing architectural mismatches. These were informed by styles, but not limited to be applied in that context. Works by DeLine [DeLine 1999] and later by Yakimovic et al. [Yakimovich et al. 1999] pro-

[1] We refrain from presenting a thorough introduction to software architectures here. Readers are referred elsewhere for such material [Bass et al. 1998, Perry and Wolf 1992, Shaw and Garlan 1996].

pose categories over which components can disagree in composition. Building on those efforts, Davis et al. have studied several classifications of architectural mismatches, producing an overall characterization of the system, control and data aspects that are relevant to define for detecting mismatches [Davis et al. 2002].

Other important pieces of related work focus on architectural styles. Shaw and Clements [Shaw and Clements 1997] present a classification of architectural styles based on a set of features focusing on control and data issues. The focus of their paper is to classify styles. They do not address compositional issues. Many other works concentrate on formally describing and analyzing specific architectural styles. The ones that are more closely related to the work we present here include papers on architectural styles described in both Z and ACME [Abowd et al. 1995, Stiger and Gamble 1997, Garlan 2003].

2.1 Environments and ADLS

All modelling languages need a syntax and semantics for them to have meaning and to be understood. Software architectures are described with architecture description languages (ADLs). ADLs have been the focus of much research since the mid 90's. Initially many notations were termed or used as ADLs, but in 2000 [Medvidovic and Taylor 2000] and then again in 2007 [Medvidovic et al. 2007] Medvidovic et. al. set about describing what should be included in an ADL. The 2000 paper described "first generation" ADLs, which should represent the three main architectural concepts of components, connectors and configurations but should also include tool support for editing and analysis. In their later 2007 paper they comment on a possible reason why many of these first generation ADLs were not widely adopted. They argue it was due to ADL environments concentrating on technological issues and make the argument for a "second generation" ADL which should incorporate both domain and business oriented concerns. There are two main issues with an ADL covering such a wide range of concepts, the first is maintaining consistency between related artifacts in different views and the second is where a specific language positions itself on a scale from specialising for a specific domain to providing general support for all domains.

In our work we use the ADL ACME[Group 2006a] and Armani[Monroe 2001] with its associated tool support ACME Studio[Group 2006b].

2.2 Web Services and SOA

SOA is a term which can frequently be found in work relating to web services, but the literature seems lacking in precise descriptions. This may be due to them being a paradigm and not a hard protocol, however the OASIS consortium has produced a reference model [OASIS 2006] which outlines the key features of SOA along with their relationships. A direct quote from the model states :

> Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

The above statement along with the three key aspects of SOA cited by OASIS *(visibility, interaction and real world effects)* are used as the guidance in this work.

The use of web services is one of the possible ways to implement a SOA [Stal 2006]. Web services themselves have been the focus of many research papers, with attempts at characterising their behaviour [Momtahan et al. 2006] and formalising their descriptions [Coleman 2004, Yeung et al. 2006, Yeung 2006]. These works concentrate on providing detailed formal models of specific narrow focussed aspects of web services and not the more broad architectural style presented in this paper.

3 Understanding SOA with Web Services

To be able to analyse web services for the purpose of building an architectural style we first need to know what characteristics to assess them against, however there is no canonical set of properties available. The choice of ADL could have offered guidance, but as ACME was designed as an extensible architecture interchange language and is not domain specific it suggests very little in this respect. Therefore the characteristics considered are drawn from the three main works mentioned above [Shaw and Clements 1996, DeLine 1999, Gacek 1998], and then examined in the context of web services as defined by the W3C.

The characteristics fall into four groups as follows:

Topology: The configuration of the components and connectors making up the system and when these are identified

Characterisation: The type of components, connectors and the representation of the data they use

Internal Behaviour: behaviour of the component not visible to others

External Behaviour: behaviour of the component visible to other components

First we present the characteristics which are influenced by the minimal constraints associated with web services and are included in the style, and then we briefly discuss those which are not included.

3.1 Characteristics Relevant to the Web Services Based Architectural Style

Only two of the topological characteristics found have any bearing on the architectural style we produced, the first of which was **infrastructure and resource availability**. This characteristic captures the dependency assumptions a component makes about the system, such as the interfaces it expects to find in the supporting software and hardware infrastructure [Gacek 1998]. While we found no constraints on the geometry of web service system topologies, it is fair to assume that a web service consumer will only attempt to connect to a web service provider interface.

Also under the topology banner comes **connection establishment**, which covers two aspects, when is the identification of a component, with which a connection is made, known and how is it made available to the component. For both aspects there are differences between components which consume a provided service, the client and those which provide it, the service. There is an underlying principle in SOA that services should be discoverable, which in turn implies that prior to an interaction neither the service nor the client know the identity of each other. This strongly points towards components in a web service architecture not being pre-bound in any way. The second aspect also differs between clients and services. Clients are supposed to discover services and therefore there identification, a URI, by searching registries and then using binding information held in a WSDL document. Services on the other hand will likely only discover the identity of the client when the interaction starts through some mechanism in either the transport protocol or the message packaging as clients are not obliged to publish any interface description before using a service.

From the Characterisation category we found several more relevant properties. The first two items **components** and **connectors** [Shaw and Clements 1997] are a broad statement about what types of components and connectors we expect to find in a system. In software architectures that are based on the use of web services it is valuable to distinguish between three different types of components given the specific roles that they play. These are: *services* which are web service components available to be discovered and integrated in various applications; *clients* that require services available as web services; and *intermediaries* that act as mediators between the clients and various servers. Note that clients and intermediaries may be web services themselves, and there may be any number of intermediaries mediating interactions between clients and servers. Given that web services are an implementation of SOA [Stal 2006], we deduce that they must provide access to some logical resource via a networked interface. Also from the W3C[2] we find that to be considered a web service the component must have an interface described by a

[2] http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/

WSDL document and also utilise SOAP as its message format. The associated connectors are largely unconstrained except that clearly they must carry SOAP messages and be compatible with whichever transport protocol is used by the web services.

Data mode [DeLine 1999] refers to the abstract mechanism employed by a component to share data, such as a shared memory location, a broadcast message or an explicit transfer in a method call. Along with the choice of mechanism, it also includes the concepts of pass by value or by reference. SOAP messages are sent on a point to point basis between component ports and, as they are the only allowed means of communication in the style, it follows that the data they contain is passed on a by value basis.

Data representation [DeLine 1999] refers to the syntactic manifestation of the data being shared between components. At is simplest level this could mean the bitwise representation of an integer, for example how many bits long it is and if it is big endian or little endian. With larger data structures, such a spreadsheet document, the components also need to agree on details of the structure in which the data resides. For web services both of these issues are resolved by the use of SOAP, which gives both a commonly understood structure and set of primitive data types which may be used.

None of the characteristics that fell within internal behaviour were constrained by either web services or SOA descriptions, so we move on to the external behaviour characteristics.

Here we found that the characteristics of **data and control transfer** and **transfer protocol** [DeLine 1999] were both greatly influenced by the web service specifications. The two characteristics refer to components agreeing on what is transferred during an interaction, data and/or control and on the number and direction of transfers respectively. These, with the possible exception of control transfer which is still implicit, are very clearly encapsulated in the message exchange patterns defined for web services, which are described next. Though these patterns only describe individual client or service ports, they do not extend to the longer term choreography between them, for example the fact that a component may expect an interaction on port 1 before it will allow an interaction on port 2 is not included.

There are two distinct versions of the main description language used by web services, web services description language (WSDL), WSDL 1.1 [W3C 2006c] and WSDL 2.0 [W3C 2006f]. These languages allow designers to describe the interfaces provided and required by a web service. The two versions perform largely the same function, however they differ in one main respect, WSDL 2.0 offers an extended set of message exchange patterns compared to those in WSDL 1.1, these will now be briefly described.

The **out-only / in-only** message exchange pattern, called **notification /**

one-way in WSDL 1.1 terms, consists of just a single message sent from one port to another with no response. This is shown in Figure 1.

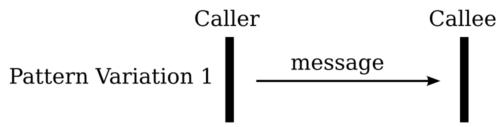

Figure 1: WSDL 1.1 Notify / One way and also WSDL 2.0 Out-only / In-only that exhibit the same message exchange pattern

The **robust-out-only / robust-in-only** pattern, which has no equivalent in WSDL 1.1, extends the previous pattern by allowing an optional message in response which would indicate a fault has occurred. This is shown in Figure 2.

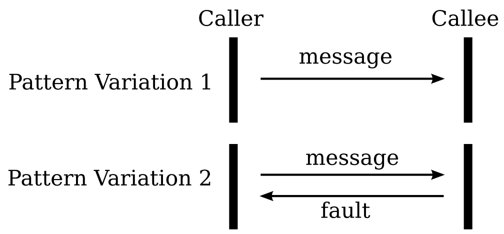

Figure 2: WSDL 2.0 Robust-out-only / Robust-in-only message exchange pattern

WSDL also allows for two way message patterns, **out-in / in-out** called **solicit-response / request-response** in WSDL 1.1, is the first of these. It consists of a single message sent from one port to the other which is then expected to reply with either the correct response or a message indicating a fault. This is shown in Figure 3.

The final message pattern included is **out-optional-in / in-optional-out** and has no equivalent in WSDL 1.1. This pattern starts with a single message sent from one port to the other that then has the options of replying with the correct response, sending a fault message or not responding at all. In the case that it sends the response message the port which sent the initial message can then send a fault message if necessary. This pattern is shown in Figure 4.

The above patterns are presented in their matching pairs, however there are a number of pattern pairs that could be described as partial matches. A partial match is where the message patterns expected by one port are a proper subset of the other's. In this situation it may be possible to constrain the behaviour of the

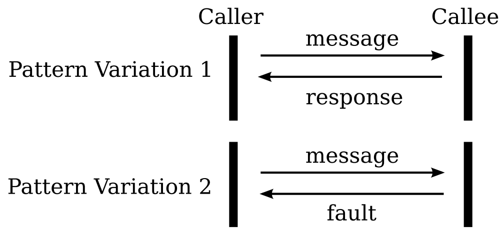

Figure 3: WSDL 1.1 Solicit response / Request response and WSDL 2.0 Out-in / In-out message exchange pattern

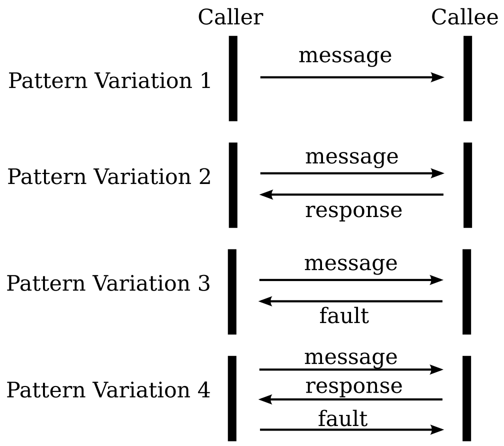

Figure 4: WSDL 2.0 Out-optional-in / In-optional-out message exchange pattern

port with the super set of message patterns such that it behaves in accordance with the expectations of the other. An example of this would be a robust-out-only port connected with an in-only port (Figure 5), so long as the component with the robust-out-only port is prepared to never receive a fault then the two ports may interoperate. This is also true of a number of other message pattern pairs such as out-optional-in with robust-in-only and out-optional-in with in-out.

3.2 Characteristics Irrelevant to the Style Description

Many more characteristics are untouched by the minimal web service specifications, these then are not included in the architectural style we present.

In the topological field we find that neither the **data topology** nor the **control topology**, which describe the overall geometric form of the data and control flows are prescribed. There is also no constraint on the **control / data shapes** or **control / data directions** which convey if there are implications

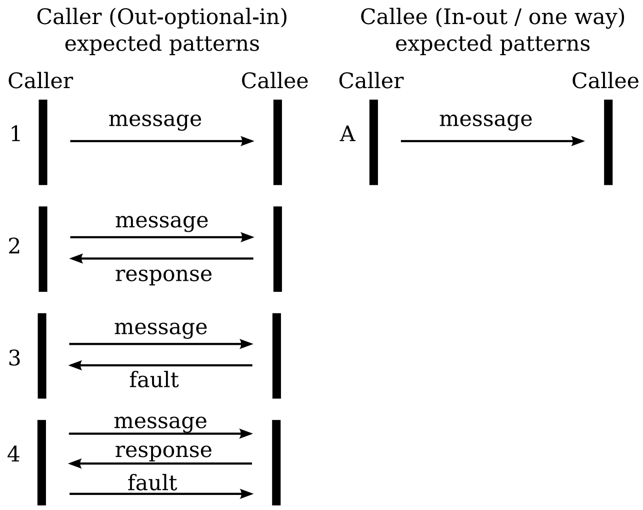

Figure 5: Partial match of Out-optional-in and In-only / One way. The callee message pattern "A" is only matched by the caller pattern "1".

between the shape and direction of the control flows and the data flows, and vice versa. Thus the shape of the architecture of a web services based system cannot be characterised in the same way as say a pipe and filter system.

Internal behaviour is highly unconstrained, with characteristics like **state persistence** and **state scope** not described. Meaning that components may or may not maintain state between invocations and they may or may not partition their internal state so the effects of one invocation are hidden from another concurrent invocation. Also, while there may be an intuition that a web service should have **concurrency support** in some way there is no constraint on if or how this is to be achieved.

There are several aspects of external behaviour which are not addressed by the standards either. **Control synchronicity** which looks at how dependent system components are upon each others' states is not touched upon. Dynamic properties such as the expected **data continuity** and **timing issues** are similarly untouched. Finally, while some message exchange patterns provide for fault messages to be sent as part of an exchange, **failure tolerance** and **error recovery** methods are neither constrained nor describable using the minimum set of specifications.

3.3 Summary

The findings of the above can be summarised into two lists. The first list includes those characteristics which are constrained or made explicit when complying with the minimum set of specifications applied to web services and the second includes those characteristics which are left free and at the choice of the designer of the component.

Constraints

- All components must be accessible to other via a network;
- each port on each component must be described by at least one WSDL document;
- each component must encode messages as SOAP;
- each connector must use transport protocols compatible with SOAP;
- service ports should allow clients to bind to them at invocation time;
- data should be passed on a " by value" basis;

Freedoms

- Control topology is unconstrained and undisclosed;
- control synchronicity is unconstrained and undisclosed;
- data topology is unconstrained;
- data and control topologies are not constrained to be isomorphic;
- data and control directionality are not constrained by each other;
- data continuity is unconstrained;
- components may or may not maintain state;
- components may or may not support concurrent invocations;
- components are not constrained to respond in any timescale;
- components may or may not support error recovery mechanisms.

4 The Environment

For this study we have used the ADL ACME & Armani [Group 2006a] and its associated tool support ACME Studio[3] as a platform for developing an environment informed about our web services style. This enables the description of the architecture of web services based software systems, while checking for the presence of some knowm possible architectural mismatches relating to that style.

ACME was originally designed as an architecture interchange language to be used as a common representation for moving architectural descriptions from one ADL and environment to another. As such it supports the required concepts of components, connectors, ports, roles and configurations, while not imposing any behavioural semantics on these elements. This feature along with its support for the user to define arbitrary properties, which are unprocessed beyond simple syntax checking, made it an attractive candidate for an exploratory study such as this. It also supports the definition of constraints upon the system, in the form of predicates written in the language Armani [Monroe 2001]. Most importantly for this work ACME allows the definition of architectural families (styles) that allow the definition of types of components, ports, roles and connectors and can include Armani predicate rules that act upon them to check the compliance of a system to that style.

ACME is supported by ACME Studio, which provides a number of desirable features, a graphical editor to manipulate the elements, syntax checking of any system or family description and execution of any rules included in a style. Figure 6 shows the interface of ACME Studio, highlighting the key interface elements that are:

Element Palette: where architectural elements from a style are presented;

System View: to where elements can be dragged and dropped to form the topology of a system;

Element View: where the properties of an element may be viewed and edited and the rules relating to that element may be viewed.

The rules in this paper are presented in ACME, a BNF for which can be found in [Monroe 2001], with the example presented graphically using the views presented in Figure 6.

[3] ACME Studio 2.2.9b was used for this work, at the time of writing there are newer versions with enhanced functionality, version 3.1.3, but it has a few teething problems that still need to be addressed.

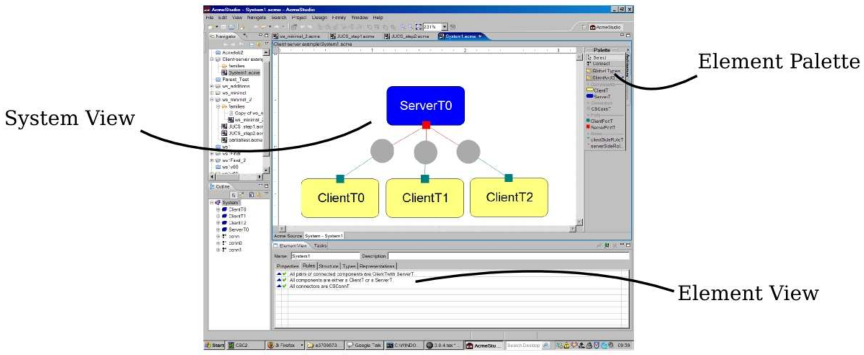

Figure 6: ACME version 2.2.9b interface, with the key elements highlighted.

4.1 Definition of the Web Services Based Software Architectural Style

We now present the description of the style in its ACME & Armani form. This is comprised of the definition of the relevant ports, components, connectors, roles, and valid configuration rules. We first present the port types and data structures they use, followed by the component types then the single connector type. Finally we present the configuration rules. Note that there are no specialised roles in this style, so the default ACME roles with no explicit properties or rules are used.

4.1.1 Ports and Data Structures

The ports in this style contain all the properties required by the style. ACME supports inheritance between types so most of the properties are found in a `PortTWS_Common` type, with `PortTWS_Service` and `PortTWS_Client` extending and specialising from it, shown in Figure 7[4]. The definitions of the data types used by the properties can be found in Figure 8.

`PortTWS_Common` starts with an `EndPointList` property. End points are defined in WSDL and define the URI and message packaging protocol used by a port. A port my have more than one end point. This property, as with all those that do not have predefined values by the style, has an Armani rule to check that it is populated, which is considered to be requirement for a system to be compliant with the style.

Next in the `PortTWS_Common` definition we have the three properties that embody the message exchange pattern characteristics of a port. First we have

[4] All ACME Armani descriptions presented in here have had their comments removed for brevity of the descriptions. A complete description of the style including all relevant comments can be found elsewhere [Gacek and Gamble 2008]

`InOurControlDomain`, which determines if "we" have administrative control over a port and therefore it may be possible to alter its definition. This is vital to the rules defined in the connector that check compatibility of the message exchange patterns of two connected ports. This property uses a `SafeBoolean` type we defined due to not being able to confirm if a property using the native boolean type has been populated by the architect.

The `MessageExchangePatterns` property represents the actual messages, their order and direction expected by this port. It is represented by a data type `messagePatterns`, which is a set of `validExchange`. A `validExchange` represents one complete path through the message exchange pattern as a sequence of `message`. Finally a `message` is a record consisting of a string token representing the message name or syntax and a direction token that shows if the message is outbound or inbound from the point of view of the port that sends the first message. Thus we can completely describe the messaging behaviour expected by a port in a way that allows for message definitions to be refined as development continues.

In `PortTWS_Common` we also have `SendsFirstMessage`, a `SafeBoolean` type where we define if a port sends the first message in the pattern or expects to receive it.

The definition of `PortTWS_Client` comes next. This port is identical to `PortTWS_Common` except that it declares itself in property `InInterface` to be part of the client interface. As previously discussed in Section 3 we saw that there is no requirement for client interfaces to be publicised so it needs no other properties.

Finally, we define `PortTWS_Service` that also extends `PortTWS_Common`. The service interface is required to be published so we have two additional properties here. `EndPointAddressList` stores a set of strings representing the address of that port. There are two rules associated with it, the first checks that the list is populated and the second checks there is one address for each end point offered by the port. The second property `WsdlDocRefs` is where the location of any WSDL documents that include this port are stored. This is not a functional property of the port, but, since it is required in SOAs that service ports be discoverable, this property has been included in the style.

4.1.2 Components

There are four types of component declared in the style, none of which have properties of their own but contain rules relating to the port types they can have, shown in Figure 9. The `CompTWSCommon` comes first and neither has any properties or rules, but it has been included as a place holder as future developments of this work may utilise it. The three types, `CompTWSClient`, `CompTWSService` and `CompTWSIntermediary` that extend `CompTWSCommon` all have a similar structure

```
Port Type PortTWSCommon = {
    Property EndPointList : EndPoints;
    invariant size(EndPointList) > 0;

    Property InOurControlDomain : SafeBoolean;
    invariant InOurControlDomain == Yes OR
        InOurControlDomain == No;

    Property MessageExchangePatterns : messagePatterns;
    invariant size(MessageExchangePatterns) > 0;

    Property SendsFirstMessage : SafeBoolean;
    invariant SendsFirstMessage == Yes OR
        SendsFirstMessage == No;
}


Port Type PortTWSClient extends PortTWSCommon with {
    Property InInterface : Interfaces = Client;
}

Port Type PortTWSService extends PortTWSCommon with {
    Property InInterface : Interfaces = Service;

    Property EndPointAddressList : EndPointAddresses;
    invariant size(EndPointAddressList) > 0;
    invariant size(EndPointAddressList) == size(
        EndPointList);

    Property WsdlDocRefs : WsdlDocs;
    invariant size(WsdlDocRefs) > 0;
}
```

Figure 7: The ACME descriptions of the three port types defined in the style.

so are explained together. `CompTWSClient` represents a client component that only consumes services and thus its rules only allow it to have `PortTWSClient` type ports. `CompTWSService` represents a service provider and so its rules only allow it to have `PortTWSService` type ports. The third type `CompTWSIntermediary` represents a brokerage type component that offers services to some components while consuming services of others.

4.1.3 Connector

The style defines a single connector type `CompTWSCommon` that is shown split over two Figures 10 and 11. The connector has no explicit properties of its own but it contains rules that make it the loci of mismatch detection. The first of these rules, shown in Figure 10 line 2, asserts that the connector may only have two

```
Property Type WsdlDocs = Set{string};

Property Type SafeBoolean = Enum { Yes, No };

Property Type legalSoapVersions = Enum { SOAP1_1, SOAP1_2
     };
Property Type legalTransportProtocols = Enum { HTTP1_0,
    HTTP1_1 };
Property Type EndPoint = Record [
     Transport : legalTransportProtocols;
     Encoding : legalSoapVersions;
];
Property Type EndPoints = Set{EndPoint};

Property Type EndPointAddresses = Set{string};

Property Type message = Record [
     ST : string;
     DT : string;
];
Property Type validExchange = Sequence<message>;
Property Type messagePatterns = Set{validExchange};

Property Type Interfaces = Enum { Client, Service };
```

Figure 8: The data structures created to represent the properties used in the style.

roles, this is to embody web service connections being point to point in nature. The second rule, Figure 10 lines 4 - 8 is a check that two connected ports have end points that have at least one matching pair of end point protocols. The final two rules in Figure 10 on lines 10 - 12 and 14 - 15 check that one of the connected ports expects to send the first message and the other expects to receive the first message.

The final two rules, shown in Figure 11, are both concerned with checking the compatibility of the message exchange patterns of the two connected ports. The first rule is defined as a heuristic and the second is defined as an invariant, as all the other rules in the style are. This does not affect how they are evaluated but instead determines how a failure of a rule is displayed. When an invariant rule evaluates to false a red warning triangle is displayed over the component or connector in question. However when a heuristic rule is failed then a yellow warning is given, indicating that a potentially less significant rule has been broken.

The message exchange pattern rules are based upon there being three possible outcomes of comparing the patterns of two connected ports. Remembering that a message exchange pattern is described using a set of valid exchanges, we define the first outcome, a complete match, as existing when the set of valid exchanges

```
Component Type CompTWSCommon = {
}

Component Type CompTWSClient extends CompTWSCommon with {
     invariant Forall p : port in self.Ports |
         satisfiesType(p, PortTWSClient) ;

     invariant size(self.ports) > 0;
}

Component Type CompTWSService extends CompTWSCommon with
    {
     invariant Forall p : port in self.Ports |
         satisfiesType(p, PortTWSService);

     invariant size(self.ports) > 0;
}

Component Type CompTWSIntermediary extends CompTWSCommon
    with {
     invariant Forall p : port in self.Ports |
         satisfiesType(p, PortTWSClient) OR satisfiesType(
         p, PortTWSService) ;

     invariant Exists p : port in self.Ports |
         satisfiesType(p, PortTWSClient) ;

     invariant Exists p : port in self.Ports |
         satisfiesType(p, PortTWSService) ;
}
```

Figure 9: The ACME description of the component types used in the style.

of one port is identical to the other's. We can then also say that when the sets of valid exchanges are disjoint then we have a mismatch. However as we saw in Sections 3 there are situations where one message exchange pattern may be a partial match for another. We can now define two conditions for a partial match to exist, they are:

– one set of valid exchanges must be a proper subset of the other; and

– the port with the superset must be within "our" domain of control so "we" may reduce its set of valid exchanges to match that of the other port[5].

The two rules are constructed such that only one of them can fail on any one connector. So if the message exchange patterns completely match then neither

[5] The decision about whether this is possible or not must lie with the architect as understanding the implications of implementing the reduction in behaviour is outside the scope of this style.

```
Connector Type ConnTWS = {
  invariant size(self.roles) == 2;

  invariant Forall r1 : role in self.roles |
    Forall r2 : role in self.roles |
    Forall p1 : PortTWSCommon in r1.attachedPorts |
    Forall p2 : PortTWSCommon in r2.attachedPorts |
    (r1 != r2 AND attached(r1, p1) AND attached(r2, p2))
   -> size(intersection(p1.EndPointList, p2.EndPointList))
    > 0;

   invariant Exists r : role in self.roles |
   Forall p : PortTWSCommon in r.attachedPorts | 1
   attached(r, p) -> p.SendsFirstMessage == Yes ;

  invariant Exists r : role in self.roles |
   Forall p : PortTWSCommon in r.attachedPorts |
 attached(r, p) -> p.SendsFirstMessage == No  ;
 }
```

Figure 10: Part 1 of the ACME description of the single connector type defined, with the message exchange pattern rules removed.

rule will fail, if the conditions for a partial match are found then the heuristic rule will fail. Finally if neither a complete match nor a partial match is found then the invariant will fail. In this way we are able to flag either a partial match or a mismatch being found and provide a visual clue to the architect regarding the degree of problem to be solved.

4.2 Configuration Rules

Finally we come to the rules that govern the configuration of the system. As we saw in Section 3, there are no constraints on the topology of a system of web services at all, however the web service style components will expect to connect to other web service style components. Also this style is aimed only at detecting mismatches between web services and may give false positives or negatives if other types of components are introduced. So two rules are defined, shown in Figure 12. The first states that all components found in a system of this style must satisfy the requirement to be of one of the three component types `CompTWSClient`, `CompTWSService` or `CompTWSIntermediary`. The second rule checks that all connectors in the system must satisfy the single connector type in the style `CompTWSCommon`, without which no mismatch detection will take place.

```
heuristic Forall r1 : role in self.roles |
  Forall r2 : role in self.roles |
  Forall p1 : PortTWSCommon in r1.attachedPorts |
  Forall p2 : PortTWSCommon in r2.attachedPorts |
  (r1 != r2 AND attached(r1, p1) AND attached(r2, p2)) ->
  (!((p1.InOurControlDomain == Yes
  AND
  (!(isSubset(p1.MessageExchangePatterns , p2.
      MessageExchangePatterns)))
  AND
  isSubset(p2.MessageExchangePatterns , p1.
      MessageExchangePatterns))
  OR
  (p2.InOurControlDomain == Yes
  AND
  (!(isSubset(p2.MessageExchangePatterns , p1.
      MessageExchangePatterns)))
  AND
  isSubset(p1.MessageExchangePatterns , p2.
      MessageExchangePatterns))));

invariant Forall r1 : role in self.roles |
  Forall r2 : role in self.roles |
  Forall p1 : PortTWSCommon in r1.attachedPorts |
  Forall p2 : PortTWSCommon in r2.attachedPorts |
  (r1 != r2 AND attached(r1, p1) AND attached(r2, p2)) ->
  (p2.MessageExchangePatterns == p1.MessageExchangePatterns
      )
  OR
  (p1.InOurControlDomain == Yes
  AND
  (!(isSubset(p1.MessageExchangePatterns , p2.
      MessageExchangePatterns)))
  AND
  (isSubset(p2.MessageExchangePatterns , p1.
      MessageExchangePatterns)))
  OR
  (p2.InOurControlDomain == Yes
  AND
  (!(isSubset(p2.MessageExchangePatterns , p1.
      MessageExchangePatterns)))
  AND
  (isSubset(p1.MessageExchangePatterns , p2.
      MessageExchangePatterns)));
```

Figure 11: Part 2 of the ACME description of the single connector type defined, showing the rules relating to checking message exchange pattern compatibility.

```
Family ws_minimal_3 = {

  invariant Forall comp : component in self.Components |
      satisfiesType(comp, CompTWSClient) OR satisfiesType(
      comp, CompTWSService) OR satisfiesType(comp,
      CompTWSIntermediary);

  invariant Forall conn : connector in self.connectors |
      satisfiesType(conn, ConnTWS);
}
```

Figure 12: The ACME description of the configuration rules that check that all components and connectors in a system satisfy the requirements of this style.

4.3 Creating an Instance of the Web Service Based Architectural Style

Once our web service based architectural style has been described using ACME & Armani, it is straight forward to use ACME Studio to create instance architectures of that style. When a new system file is created in ACME Studio the architect is asked via a wizard, which of the architectural styles currently know to ACME Studio should be applied to that system. Subsequently, all elements depicted in the *Element Palette* correspond to the elements defined in the style description. All an architect needs to do is drag and drop elements from the palette into the *System View*. Upon selecting an individual element in the system view, it is possible to then populate its properties in the *Element View*.

Any analysis included in the chosen architectural style is run automatically by ACME studio whenever a change is made to the elements or attachments in system. The architect is then free to manipulate the system to meet its requirements and only needs to consider the analysis when one or more of the rules in the style are broken. When this happens ACME Studio displays a red warning triangle over each and every component and connector in which a problem has been found. The architect can then select each highlighted element in turn and find which rules have been broken by examining the rules section of the element view.

5 Case Study

We now present a scenario to demonstrate how this style can be used within ACME Studio to detect mismatches. The scenario is focused on the development of a new satellite navigation system based upon existing services but with some additional and, we believe, desirable functionality.

The service being developed consists of two separate software components the satellite navigation provider (SNP), which is centralised at some data centre and an in car navigation unit (NU). The NU unit has the usual functionality of selecting a route from the current location to a specified address, but it can also delegate route calculation back to the systems at SNP via web service connections over a General Packet Radio Service (GPRS [6]) connection. The routes calculated can then take into account the latest traffic reports and road works, leading to a potentially much better route choice. The central SNP systems can also update the route provided to individual NUs if there is a relevant traffic situation change. This is done by querying the current location of the vehicle and sending a new route plan if appropriate.

A second addition to the normal satellite navigation functionality is that SNP will contact and direct recovery services to the vehicle if a breakdown is signalled. To enhance the service provided, the NU unit can obtain some diagnostic information from the vehicles engine management unit (if available) so the recovery service can respond to the situation in the most appropriate way. The information is obtained from the engine management unit using web service protocols and is assumed to consist of raw sensors' information. Thus, we also include a service provided by the car manufacturers where by they will decode and collate the sensors' data and return a plain text diagnostic. The diagnostic, vehicle location and passenger status is passed to a number of recovery services, which return their assistance offers, consisting of estimated time of arrival (ETA), cost and details such as if they intend to attempt to repair on site or just to tow away. The user can then select which of the service offers to accept. Additionally, the recovery services may need to alter their ETA as a result of other breakdowns that have a higher priority, such as a lone female driver at night, in which case the new details of the recovery can be sent to the NU unit.

In Figure 13 we see the initial proposed architecture of this system consisting of components to represent SNP and NU that are being developed, as well as existing external services: two recovery services (RS1 & RS2), two car manufacturers (CM1 & CM2) and a selection of their engines with their corresponding management units (CM1E1, CM1E2 & CM2E1). These have been described within our ACME Studio environment based on our Web Services style description.

ACME Studio has placed three warning triangles on three of the connectors in the architecture. These warning triangles are overlaid on components or connectors to indicate that one or more constraints on them are not met. In this case that means that an architectural mismatch has been detected and is localised around that connector. A triangle does not indicate what the nature

[6] http://www.gsmworld.com/technology/gprs/index.shtml or for the specification detail see http://www.3gpp.org/ftp/Specs/html-info/0260.htm.

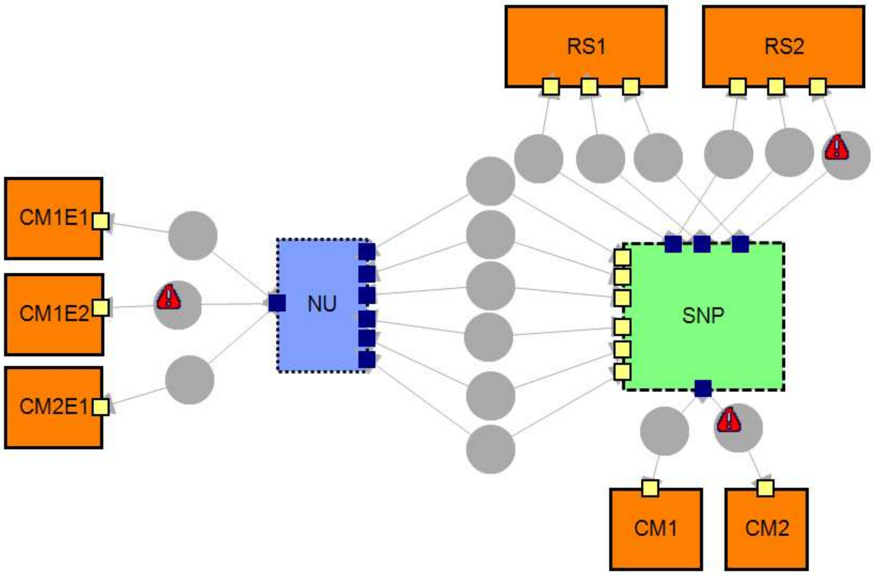

Figure 13: The initial system architecture with warning triangles showing where mismatches have been detected.

of the mismatch is, for that one must select the connector in question and note which of the rules are reported as failed. Figure 15 shows this view for the connector between NU and CM1E2. The rule indicates that there is no matching pair of enpoint protocols shared between the two ports as shown in the following two fragments from the architecture description, the first being from the port on NU and the second being from the port on CM1E2.

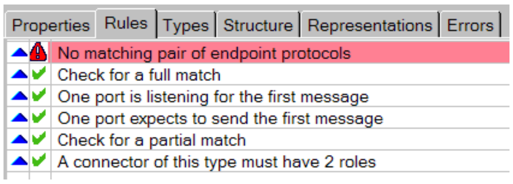

Figure 14: The rule summary for the connector between NU and CM1E2

```
// extract from the original NU port description
Property EndPointList : EndPoints = {[
    Transport = HTTP1_0;
    Encoding = SOAP1_1 ]};
```

```
// extract from the CM1E2 port description
Property EndPointList : EndPoints = {[
    Transport = HTTP1_0;
    Encoding = SOAP1_2 ]};
```

This is corrected by changing the SOAP processor used by NU to one which supports both SOAP1.1 and SOAP 1.2, which is described by altering the port description to be as follows.

```
// extract from the updated NU port description
Property EndPointList : EndPoints = {[
    Transport = HTTP1_0;
    Encoding = SOAP1_1 ], [
    Transport = HTTP1_0;
    Encoding = SOAP1_2 ]};
```

The second warning is found on a connector between SNP and RS2, examining the rules reveals that the mismatch relates to the messages exchanged between the ports, Figure 15. From the descriptions we can learn that while the port on RS2 expects request response message exchange pattern, the port on SNP is using a one way (notification) pattern, shown in Figure 16. This is so RS2 can get a confirmation that their services are still required after they have to change details of a previously accepted offer.

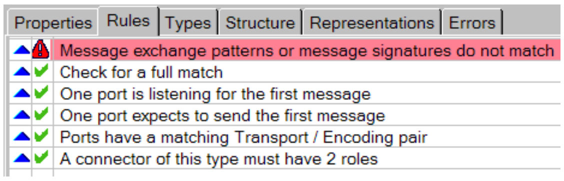

Figure 15: The rule summary for the connector between SNP and RS2

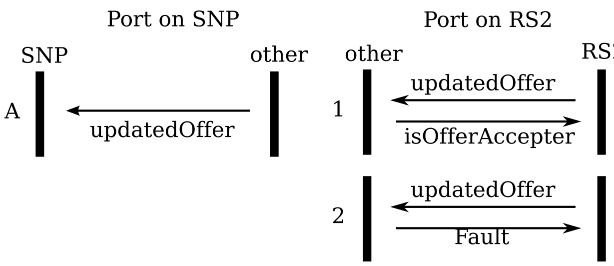

Figure 16: The mismatching message exchange patterns between SNP and RS2

```
// extract from the original SNP port description
Property MessageExchangePatterns : messagePatterns = {< [
  ST = "updateOffer";
  DT = "out" ] >};

// extract from the original RS2 port description
Property MessageExchangePatterns : messagePatterns = {< [
  ST = "updateOffer";
  DT = "out" ], [
  ST = "isUpdateAccepted";
  DT = "in" ] >, < [
  ST = "updateOffer";
  DT = "out" ], [
  ST = "fault";
  DT = "in" ] >};
```

To correctly interoperate with RS2 then it is necessary to add a new port to SNP which follows the expected interaction[7]. Then for completeness the interface between NU and SNP is altered such that the user can make the decision whether to accept the new offer or not.

The final warning exists on the connector between SNP and CM2. Looking at the rules summary for this connector we find that the same rule failed as for the previous connector, however this time examining the message exchange patterns shows that they are both of the request response type. So this time we also look at the tokens representing the data included in each message and find that this is where the problem lies. CM2 requires an additional data item to be sent before they respond with a diagnostic report, this is the vehicle chassis number that is not included in the raw sensor data. To avoid this mismatch another client port is added to SNP which has the same message exchange pattern as the original but also includes this extra information.

```
// extract from the original SNP port description
Property MessageExchangePatterns : messagePatterns = {< [
  ST = "rawVehicleData";
  DT = "out" ], [
  ST = "diagnosticInformation";
  DT = "in" ] >, < [
  ST = "rawVehicleData";
  DT = "out" ], [
  ST = "fault";
  DT = "in" ] >};

// extract from the original CM1 port description
Property MessageExchangePatterns : messagePatterns = {< [
  ST = "rawVehicleDataAndChassisNumber";
  DT = "out" ], [
  ST = "diagnosticInformation";
  DT = "in" ] >, < [
```

[7] I.e. the description of the new SNP port message exchange patterns property becomes identical to that of the RS2 port.

```
ST = "rawVehicleDataAndChassisNumber";
DT = "out" ], [
ST = "fault";
DT = "in" ] >};
```

With these corrections made we can see that the final architecture, shown in Figure 17 has no mismatches detected according to this architectural style. So actual development of the software components NU and SNP could now begin with greater confidence of success.

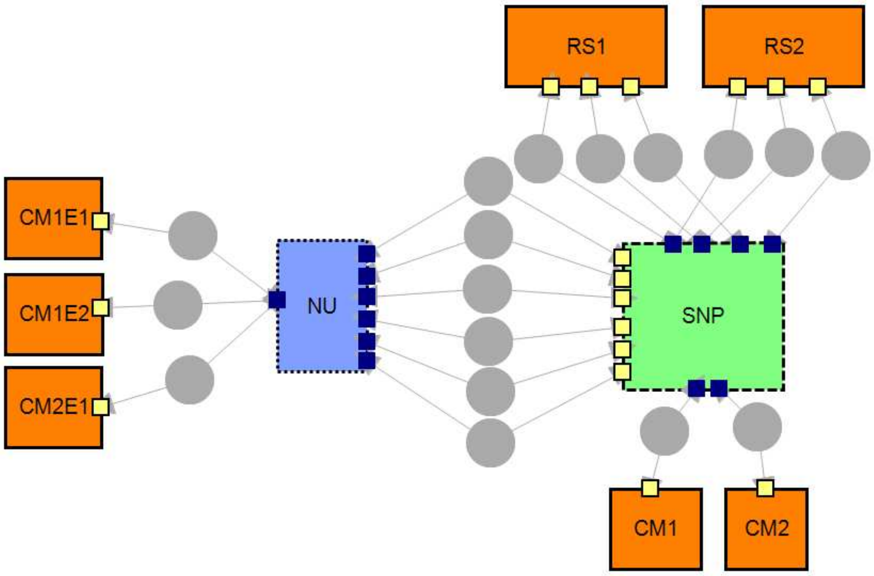

Figure 17: The final architecture of the envisaged system.

6 Conclusions

In this paper we have introduced a web services based architectural style, with its corresponding description in ACME & Armani. We have described the architecting environment that we have created such that individual application software architectures can be created using the style defined and subsequently analyzed for the presence of some known architectural mismatches. We have illustrated the use of our environment with a realistic case study. By doing so we have shown that, by using the proposed architectural style with its inherent properties and rules, it is possible to detect, at design time, architectural mismatches in web services based software systems.

To date we have been able to use our environment with several different case studies with varying degrees of complexity. In all cases we have been able to easily model all relevant properties. This includes the enforcement of the appropriate population of all relevant property values by the architect. This ensures that the meta-data required to detect mismatches is explicitly defined in the system configuration.

As illustrated by our case study, at times the same error message is given by our environment to report on differing yet related mismatches. An architect would need to know where to check in order to retrieve information guiding to the actual offending properties. We believe this could represent a barrier to the adoption of such an environment in an industrial setting. However, the benefits provided by the analysis might be considered to outweigh this problem.

As with other styles, the web services based architectural style provides some constraints, but also leaves some characteristics to be freely defined by the individual application being developed in that style. An approach like ours can clearly identify mismatches related to the characteristics that are constrained, but does not necessarily cover all possible mismatches, especially those related to unconstrained aspects.

We found that ACME & Armani provided both a flexible and at the same time challenging environment to build styles and system descriptions within. We were able to include all the properties and rules we wanted to, but not without some effort. For example, given two sets A and B there is no support for explicitly determining if $A \subset B$, instead it was necessary to check that $A \subseteq B$ and $\neg\ B \subseteq A$. This made some rules a little more complicated than necessary.

We did benefit however from the flexible data structures ACME allowed us to build, especially with respect to the message exchange patterns. This structure uses a string token to represent the messages exchanged during an interaction. The tokens may at first simply consist of message names, as in the scenario shown, but could equally hold a string representation of the complete message syntax as the system is developed towards an implementable design. The analysis rules which operate on the message exchange patterns can work equally well in both situations and will detect mismatches in them regardless of the detail level chosen by the designer. This adds weight to our belief that ACME, ACME Studio and Armani along with this type of style based analysis have potential to offer real benefits when building web service based systems.

The aim of this paper is to define a style that covers *all* web services based software architectures. By doing so, aspects such as the long term choreography of message exchanges are outside its scope. This is caused by the fact that such information is not necessarily available for all web services components. We are working on an extended version of the style that does cover long term choreography, although this is at the price of no longer being able to be used in conjunction with *all* existing web services.

Acknowledgements

We take this opportunity to thank the reviewers for their thorough and insightful comments. These have helped us to dramatically improve the paper. We would also like to acknowledge the support for this work by the UK EPSRC projects DIRC and TrAmS.

References

[Abd-Allah 1996] Abd-Allah, A. (1996). *Composing Heterogeneous Software Architectures*. PhD thesis, Univeristy of Southern California, Los Angeles, CA.

[Abowd et al. 1995] Abowd, G., Allen, R., and Garlan, D. (1995). Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4:319–364.

[Baker 2002] Baker, S. (2002). Web services and corba. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 618–632, London, UK. Springer-Verlag.

[Bass et al. 1998] Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley.

[Behr 2003] Behr, A. (2003). Defining the soa. *Software Development Times*, pages 29–31.

[Bhuta and Boehm 2007] Bhuta, J. and Boehm, B. (2007). A framework for identification and resolution of interoperability mismatches in cots-based systems. In *IWICSS '07: Proceedings of the Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques*, page 2, Washington, DC, USA. IEEE Computer Society.

[Coleman 2004] Coleman, J. W. (2004). Features of bpel modelled via structural operational semantics. Master's thesis, Newcastle University.

[Davis et al. 2002] Davis, L., Gamble, R. F., and Payton, J. (2002). The impact of component architectures on interoperability. *J. Syst. Softw.*, 61(1):31–45.

[DeLine 1999] DeLine, R. (1999). A catalog of techniques for resolving packaging mismatch. In *SSR '99: Proceedings of the 1999 symposium on Software reusability*, pages 44–53, New York, NY, USA. ACM Press.

[DeLine 2001] DeLine, R. (2001). Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering*, 27(2):124–143.

[Egyed et al. 2000] Egyed, A., Medvidovic, N., and Gacek, C. (2000). Component-based perspective on software mismatch detection and resolution. *IEE Proceedings - Software*, 147(6):225–236.

[Ferguson and Stockton 2005] Ferguson, D. F. and Stockton, M. L. (2005). Service-oriented architecture: programming model and product architecture. *IBM Syst. J.*, 44(4):753–780.

[Gacek 1998] Gacek, C. (1998). *Detecting Architectural Mismatches During Systems Composition*. PhD thesis, University of Southern California.

[Gacek and Gamble 2008] Gacek, C. and Gamble, C. (2008). Minimal web services style — architectural style description and example instantiation. Technical Report CS-TR-1078, Newcastle University, Newcastle upon Tyne, United Kingdom.

[Garlan 2003] Garlan, D. (2003). Formal modeling and analysis of software architecture: Components, connectors, and events. In *Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures (SFM 2003)*, pages 1–24.

[Garlan et al. 1995] Garlan, D., Allen, R., and Ockerbloom, J. (1995). Architectural mismatch: why reuse is so hard. *Software, IEEE*, 12(6):17–26.

[Group 2006a] Group, A. (2006a). http://www.cs.cmu.edu/˜acme/.

[Group 2006b] Group, A. (2006b). http://www.cs.cmu.edu/˜acme/AcmeStudio/

[Hepner et al. 2006] Hepner, M., Gamble, R., Kelkar, M., Davis, L., and Flagg, D. (2006). Patterns of conflict among software components. *J. Syst. Softw.*, 79(4):537–551.

[McIlroy 1969] McIlroy, M. D. (1969). Mass produced software components. *Software Engineering,NATO Science Committee*, pages 138–155.

[Medvidovic et al. 2007] Medvidovic, N., Dashofy, E. M., and Taylor, R. N. (2007). Moving architectural description from under the technology lamppost. *Inf. Softw. Technol.*, 49(1):12–31.

[Medvidovic and Taylor 2000] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93.

[Momtahan et al. 2006] Momtahan, L., Martin, A., and Roscoe, A. W. (2006). A taxonomy of web services using csp. *Electr. Notes Theor. Comput. Sci.*, 151(2):71–87.

[Monroe 2001] Monroe, R. T. (2001). Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science.

[OASIS 2006] OASIS (2006). Reference model for service oriented architecture v 1.0. http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf.

[Perry and Wolf 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52.

[Shaw and Clements 1996] Shaw, M. and Clements, P. (1996). A field guide to boxology: Preliminary classification of architectural styles for software systems.

[Shaw and Clements 1997] Shaw, M. and Clements, P. C. (1997). A field guide to boxology: Preliminary classification of architectural styles for software systems. In *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, pages 6–13, Washington, DC, USA. IEEE Computer Society.

[Shaw and Garlan 1996] Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall.

[Stal 2006] Stal, M. (2006). Using architectural patterns and blueprints for service-oriented architecture. *IEEE Softw.*, 23(2):54–61.

[Stiger and Gamble 1997] Stiger, P. R. and Gamble, R. F. (1997). Blackboard systems formalized within a software architectural style. In *Systems, Man, and Cybernetics, 1997. 'Computational Cybernetics and Simulation'., 1997 IEEE International Conference on*, pages 1204–1209. IEEE.

[Uchitel and Yankelevich 2000] Uchitel, S. and Yankelevich, D. (2000). Enhancing architectural mismatch detection with assumptions. *ecbs*, 00:138–147.

[W3C 2006a] W3C (2006a). Soap version 1.2 part 1. http://www.w3.org/TR/soap12-part1/.

[W3C 2006b] W3C (2006b). Web services architecture. http://www.w3.org/TR/ws-arch/#introduction.

[W3C 2006c] W3C (2006c). Web services description language 1.1. http://www.w3.org/TR/wsdl.

[W3C 2006d] W3C (2006d). Web services description language 2.0 part 0 : Core jan 6 2006. http://www.w3.org/TR/2006/CR-wsdl20-primer-20060106/.

[W3C 2006e] W3C (2006e). Web services description language 2.0 part 1 : Core jan 6 2006. http://www.w3.org/TR/2006/CR-wsdl20-20060106/.

[W3C 2006f] W3C (2006f). Web services description language 2.0 part 2 : Adjuncts jan 6 2006. http://www.w3.org/TR/2006/CR-wsdl20-adjuncts-20060106/.

[Yakimovich et al. 1999] Yakimovich, D., Bieman, J. M., and Basili, V. R. (1999). Software architecture classification for estimating the cost of cots integration. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 296–302, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Yeung 2006] Yeung, W. L. (2006). Mapping ws-cdl and bpel into csp for behavioural specification and verification of web services. In *ECOWS*, pages 297–305.

[Yeung et al. 2006] Yeung, W. L., Wang, J., and Dong, W. (2006). Verifying choreographic descriptions of web services based on csp. In *SCW '06: Proceedings of the IEEE Services Computing Workshops (SCW'06)*, pages 97–104, Washington, DC, USA. IEEE Computer Society.