Representing and Analysing Composed Web Services using CRESS

Kenneth J. Turner
Computing Science and Mathematics, University of Stirling
United Kingdom FK9 4LA
Email kjt@cs.stir.ac.uk

Abstract

Composite web services are defined using the industry-standard language BPEL (Business Process Execution Logic). There is a strong need for graphical and automated support for this task. It is explained how CRESS (Chisel Representation Employing Systematic Specification) has been extended to meet this challenge. CRESS supports straightforward graphical descriptions of composite web services. Sample descriptions are presented of these. It is outlined how they are automatically implemented and systematically analysed using the target languages BPEL and LOTOS (Language Of Temporal Ordering Specification).

Keywords: BPEL (Business Process Execution Logic), LOTOS (Language Of Temporal Ordering Specification), web service

1 Introduction

1.1 Background to Web Services

Web services have become a very popular way of providing access to distributed applications. These may be legacy applications given a web service wrapping, or they may be new applications designed for the web. Web services are widely documented online (e.g. see *www.webservices.org* for a commercial perspective, *www.webservicesarchitect.com* for a technical perspective, and *www.xmethods.net* for public web services).

The interface to a web service is defined in WSDL (Web Services Description Language [26]). However this is purely syntactic, essentially defining the types of inputs, outputs and faults for web service operations. WSDL also defines the binding of operations to a transfer mechanism such as SOAP (Simple Object Access

Protocol [23]). WSDL does not define the semantics of a web service; formalising the behaviour of a web service is therefore desirable. Being XML-based and therefore textual, WSDL can be created and edited manually. However, this is an intricate and error-prone task. For this reason, most solutions aim to create WSDL automatically.

WSDL describes an isolated web service. The excitement in web services is *composing* them into what is called a *business process*. Assume (realistically) the following web services: airlines take flight bookings, hotels take room bookings, hire firms take car bookings, and banks accept electronic payments. A travel agency can then build a business process that arranges all aspects of a trip. Flights, accommodation and car hire can be arranged and paid for through a single service.

Closely related terms are used to describe the coordination of web services. Business process (flow) modelling has an independent existence as a technique for describing business activities and their interrelationships. This lent itself naturally to describing flows among web services, e.g. with IBM's WSFL (Web Services Flow Language) or Microsoft's XLANG. Choreography languages such as WS-CDL (Web Services Choreography Description Language [25]) are used to characterise high-level enterprise collaborations based on web services. Orchestration languages such as BPEL deal with the combination of web services to create more complex composite ones.

Unfortunately, many competing standards emerged for composing web services [14]. Of these, WSFL and XLANG were leading examples. A significant step was made with agreement on the industry-wide specification called BPEL4WS (Business Process Execution Language for Web Services [2]). This is being standardised as WS-BPEL [3]. For brevity, both are referred to as BPEL in this paper.

WSFL stemmed from work on business process modelling. It therefore emphasises flows among activities. In contrast, XLANG had its origins in structured program design. It therefore emphasises constructs for sequencing, alternation and iteration. BPEL is a careful fusion of both approaches that gives considerable flexibility to the developer. Both IBM and Microsoft contributed to the development of BPEL. It is the author's perception that WSFL and XLANG are now effectively superseded by it for composing web services. [14] provides a detailed and useful comparison of various web service flow languages.

BPEL is a recent and evolving language, so tool support is still developing. BPEL is also XML-based, so it can be difficult to understand a complex flow from a purely textual description. A graphical view of composed web services is therefore highly desirable.

1.2 Overview of The Paper

Section 2 explains the context of CRESS for describing web services. This is expanded in section 3 that gives a brief overview of web service concepts and how CRESS represents these. Three interrelated examples are used to illustrate the approach. The translation of CRESS into BPEL and WSDL is outlined in section 4. A lender service is used as an example of what files are generated. Section 5 discusses the practical implications of what is achieved through formal analysis of web services. Scenario-based validation is introduced as a pragmatic way of validating both specifications and implementations of web services.

2 The Context of Web Services

This section gives the context of web service modelling and analysis. It introduces the philosophy of CRESS (Chisel Representation Employing Structured Specification). This was developed by the author for describing services, typically in communications. Although initially used with voice services, the value of CRESS has now been demonstrated for modelling web services as well.

2.1 Web Service Modelling

Web services are well established and widely supported by commercial tools. Languages like Java and C# have web service bindings. Web services are supported by frameworks like Microsoft's .NET and Sun's ONE (Open Net Environment).

The Semantic Web, e.g. as supported by RDF (Resource Description Framework), aims to give formal meaning to the content of web pages. The approach relies on creating ontologies containing properties and interrelationships of the terms in some domain. Semantic Web Services apply similar techniques. OWL (Web Ontology Language, e.g. [24]) has been developed as a common standard for describing ontologies in a web service context. DAML-S (DARPA Agent Markup Language – Services), now OWL-S, is an expressive elaboration of the ideas in RDF for use with agent-based systems and web services. WSMF (Web Services Modeling Framework) is a comparable development, but inspired by WSFL. Purely ontological approaches focus on the content of what web services provide. Of greater interest here are approaches that also describe the behaviour and composition of web services. OWL-S is therefore particularly relevant.

BPMN (Business Process Modeling Notation [4]) might be viewed as a competitor notation to CRESS for describing web services. However, BPMN is a very large notation (the standard is almost 300 pages). It also has a single purpose: describing business processes. BPMN is essentially a front-end for creating web services. Tools supporting BPMN often have complex interfaces, with simple flow diagrams being supplemented by form-based entry of activity parameters. Of course, BPMN is aimed at large-scale industrial development of web services. It also supports the entirety of the capabilities found in, for example, BPEL.

In contrast, CRESS is a compact and general-purpose notation that has now been proven on services from four different domains. Like BPMN, CRESS describes web services graphically. However as will be seen, CRESS descriptions are high-level and can be understood with limited training; this papers explains nearly everything required to create web services using CRESS. Unlike BPMN and its associated tools, CRESS offers automated translation to formal specifications (e.g. LOTOS, SDL) as well as to implementations (e.g. BPEL, WSDL). This means that CRESS descriptions of web services obtain the benefits of automated verification and validation as well as automated implementation.

CRESS introduces a feature concept that is lacking in other web service approaches. This allows capabilities to be shared among web services, invisible to the user and even the developer. Compared to BPMN, the main lack in CRESS is that it does not support every capability of BPEL – though it is believed that the most important aspects are covered. If necessary CRESS could be used as a precursor to BPMN, allowing major aspects of web service functionality to be analysed and demonstrated before committing substantial implementation efforts.

2.2 Web Service Support

The focus of this paper is on composing web services. Due to the relative newness of BPEL, support is only now maturing. Commercial products include IBM's WebSphere, Microsoft's BizTalk Server and Active Endpoint's WebFlow. Oracle's BPEL Process Manager is made available for prototyping purposes, including a wide range of examples and extensive tutorials. ActiveBPEL (*www.activebpel.org*) is an open-source community development that builds on AXIS (Apache Extensible Interaction System), an implementation of SOAP for the Tomcat web server.

Both ActiveBPEL and BPEL Process Manager provide BPEL interpreters; ActiveBPEL has been mainly used in the work of this paper. Of greater interest here are BPEL design tools. Given the complexity of BPEL, graphical design tools are highly desirable. Both Active WebFlow and BPEL Process Manager offer graphical design integrated with IBM's ECLIPSE. BPEL processes can be defined by drag-and-drop. Activity properties are manually entered in forms, though help is provided to locate information such as web service partners and process variables. A significant difference is that BPEL Process Manager currently shows only a simple representation of flows; arbitrary graphs are just shown as parallel activities. Active WebFlow fully displays flows, and also supports visual simulation and debugging of BPEL processes.

So far, research on formalising web services has been limited. Most approaches use either finite state methods (e.g. Petri nets or finite state automata) or process algebras. Since the semantics of a process algebra is often given as an LTS (Labelled Transition System), the two approaches are not so fundamentally different. For example, [9] gives a timed semantics for XLANG that allows web services to be checked for interoperability, and also to be implemented via transition systems.

As an example of finite state methods, LTSA-WS (Labelled Transition System Analyzer for Web Services [7, 8]) is also close in aim to CRESS. LTSA-WS allows composed web services to be described in a BPEL-like manner. Service compositions and workflow descriptions are automatically translated into FSP (Finite State Processes) to check safety and liveness properties. CRESS differs in being a multi-purpose approach that works with many different kinds of services and with many different target languages. A CRESS description is pitched at a more abstract level. CRESS may be used with any analytic technique using the formal languages it supports, although it offers its own approach based on scenario validation.

As an example of process algebraic methods, LOTOS has been used to specify web services. Plain process algebras often lack the ability to specify data, which is often an important characteristic of web services. LOTOS offers the advantage of specifying both data and behaviour within one language. The work described in [5, 6] supports automated conversion between BPEL and LOTOS. This has been used to specify, analyse and implement a stock management system and also negotiation through web services. CRESS differs from this work in using more abstract descriptions that are translated into BPEL and LOTOS; there is no interconversion among these representations. CRESS descriptions are language-independent, and can thus be used to create specifications in other formal languages (e.g. SDL). CRESS also offers a graphical notation that is more comprehensible to the non-specialist. This is important since web service development typically involves business and marketing staff as well as technical experts.

2.3 CRESS for Service Modelling

The author developed CRESS from early work by BellCore on Chisel [1] for describing voice services. CRESS has been used to specify and analyse voice services for the IN (Intelligent Network) [15], SIP (Session Initiation Protocol) Internet Telephony [16], and IVR (Interactive Voice Response) [17].

Service descriptions in CRESS are graphical and accessible to non-specialists. Yet, CRESS descriptions are automatically translated into formal languages for analysis, and into implementation languages for execution. CRESS may in principle be translated into any constructive specification language. However, the focus has been on standardised languages used in communications: LOTOS (Language Of Temporal Ordering Specification [10]) and SDL (Specification and Description Language [12]).

CRESS has plug-in code for the application domain and the target language. This provides direct support for each domain, but makes CRESS readily extensible for new purposes. CRESS has explicit support for features, allowing these to extend a base service in a variety of ways.

Essentially, CRESS describes the flow of actions in a service. It was therefore natural to investigate whether CRESS might be used for describing web service flows. This has proven to be an appropriate application of CRESS; in fact the notation has required surprisingly little extension. The main descriptive additions have been for simultaneous definition of multiple services, structured data types, and parallel flows. A back-end translator from CRESS to BPEL and WSDL has of course been a necessary and substantial addition.

Despite being a research tool, CRESS offers a number of distinctive benefits when used with web services:

- CRESS descriptions of composed web services are graphical and therefore attractive to industrial users. The same notation also applies to a wide variety of services, and not just to web services.
- CRESS service descriptions are automatically translated into implementations – BPEL and WSDL for web services. The translated code is neatly organised with extensive comments. This means that CRESS-generated code can be readily related to a service description, unlike commercial tools where the output does not seem to be intended for human use. Even if CRESS is used to implement web services, the generated code can also be used with commercial tools.
- Most importantly, the *same* CRESS service descriptions are automatically translated into formal specifications (currently LOTOS for web services, but SDL is generally supported by CRESS). Commercial tools are limited to manual simulation and debugging. By formalising web services, CRESS provides access to rigorous analysis. This may use any technique associated with the formal language, such as state space exploration or model-checking.
- Formal languages are common as a basis for investigating compatibility of services (called the feature interaction problem in telephony). Compatibility can be checked with any approach based on the formal languages supported by CRESS. However, CRESS offers its own technique based on validation of use-case scenarios [20]. Interference among web services has so far received little attention. [21, 22] are two of the few papers on this topic.

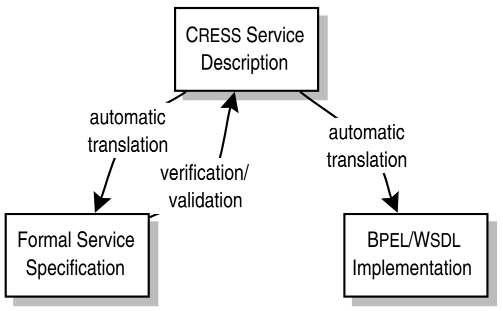

Figure 1: Web Service Development with CRESS

- CRESS introduces a feature concept that is lacking in other web service approaches. The idea, borrowed from telephony, is to extend services with features that provide modular and automatic additions of functionality.

CRESS is used to give higher-level descriptions of services that are automatically translated into other languages for other purposes. As an example, its use for web services is shown in figure 1. A CRESS description is initially given of a service. Since CRESS supports abstract and compact service descriptions, the effort at this stage is small compared to actual implementation. The CRESS is automatically translated into a formal specification that is prototyped, verified and validated. Typically this exposes flaws and weaknesses that require the CRESS description to be improved. Once the CRESS has been finalised, it is automatically turned into a real implementation that is deployed in a web server.

Translation into a formal specification gives precise meaning to web services (and their associated languages). More importantly, it allows automated verification of desirable properties such as interoperability and freedom from deadlock. This allows the description of a service to be checked prior to implementation. A number of specification languages (including LOTOS and SDL) allow formal specifications to be animated. This is very useful for demonstrating a service in prototype form to a customer.

The same CRESS description is also automatically translated into a working implementation: BPEL and WSDL for web services. The formalisation will have established confidence in the service description, so the need for testing the service implementation is significantly reduced. However some factors can really only be evaluated with the actual implementation, e.g. usability and performance.

[18] discusses the translation and formal analysis of web services using CRESS supported by LOTOS. The present paper gives the complementary view of practical web development using CRESS supported by BPEL and WSDL.

3 CRESS Description of Web Services

This section introduces web service concepts and their representation in CRESS. As concrete illustrations, three interrelated examples are given of web services.

3.1 Basic Web Service Concepts

A composite web service is termed a *business process*. It exchanges messages with *partner* web services, considered as service providers. A web service may be invoked *synchronously* (a request and associated response) or *asynchronously* (a request followed by a separate response). A business process is itself a web service with respect to its users. Web services have communication *ports* where *operations* are invoked. An unsuccessful operation gives rise to a *fault*. *Compensation* undoes work following a fault. *Correlation* links asynchronous messages to the correct business process instance.

A CRESS diagram is a directed graph. In BPEL terms, this defines an executable business process. Someone who knows BPEL will find the CRESS representation familiar, but more compact. Numbered nodes contain actions, termed activities for web services. Arcs between nodes may be labelled with expression guards or event guards. Expression guards are used to make alternative choices; they may be followed by assignments in the form / *variable* <– *expression*. Event guards introduce behaviour conditional on an event. In a CRESS web service description, the events are faults, requests for compensation and correlation requests.

For business processes, CRESS is required to offer sophisticated control over parallelism. Branches in a CRESS diagram normally reflect alternatives. However **Fork**/**Join** can be used to bracket parallel paths. Although BPEL has different constructs for sequence, iteration and graph-like flow of control, CRESS models these all in a uniform way. Each CRESS node (BPEL activity) is joined by a CRESS arc (BPEL link). A CRESS expression guard (BPEL transition condition) determines whether an arc may be traversed. This requires the prior activity to terminate successfully and the guard to hold.

CRESS names are given in simple or hierarchic form. Operation names have the format *partner.port.operation*. Fault names have the format *fault.variable*, the fault variable being optional. Simple variables have the types defined by XSD (XML Schema Definition, e.g. **Float** *f*, **Integer** *i*, **String** *s*). Structured variables defined in braces {...} are accessed in the form *structure.field*.

3.2 CRESS for Business Processes

The subset of CRESS activities appearing in this paper is explained below; CRESS supports more than is described here. As usual, '?' means optional, '*' means zero or more times, and '|' denotes choice.

Invoke *operation output* (*input faults**)? An asynchronous (one-way) invocation sends only an output. A synchronous (two-way) invocation exchanges an output and an input with a partner web service. CRESS requires potential faults to be declared statically, though their occurrence is dynamic. The faults that may arise in a business process are implied by **Invoke**, **Reply** and **Throw**.

Receive *operation input* Typically this is used at the start of a business process to receive a request for service. An initial **Receive** creates a new instance of the process. Each such **Receive** is usually matched by a **Reply** for the same operation. **Receive** also accepts an asynchronous response to an earlier one-way **Invoke**.

Reply *operation* **(***output* | *fault***)** Typically this is used at the end of a business process to provide some output. Alternatively, a fault may be signalled.

Fork *strictness*? This is used to introduce parallel paths; further forks may be nested to any depth. Normally, failure to complete parallel paths as expected leads to a fault. This is strict parallelism, and may be indicated explicitly as *strict* (the default). If this is too stringent, *loose* may be used instead.

Join *condition*? Each **Fork** is matched by **Join**. By default, only one of the parallel paths leading to **Join** need terminate successfully. However, an explicit join condition may be defined over termination of parallel activities. In CRESS, the condition uses the node numbers of immediately prior activities. For example, 1 && (2 || 3) means that activity 1 and either activity 2 or 3 must terminate successfully. In turn, this means that the prior activities must also succeed.

Throw *fault* This reports a fault as an event to be caught elsewhere by an event handler.

Compensate *scope*? This is called after a fault to undo previous work. An explicit scope (CRESS node number) indicates which compensation to perform. In the absence of this, compensation handlers are called in reverse order of completion.

The **Throw** and **Compensate** actions cause event handlers to be invoked. A fault is considered by progressively larger scopes within the process until it is handled. Compensation also deals with progressively larger scopes. In BPEL a handler may be defined in any scope, but in CRESS it is either global or associated with an **Invoke**. (This is a small restriction that accords with common practice anyway.)

Catch *fault* This introduces a separate flow for dealing with the specified fault. If a fault has just a name and no value, it is handled by a **Catch** with a matching fault name. A fault with name and value is handled by a **Catch** with matching fault name and variable type, or by a **Catch** without a fault name but a matching type.

CatchAll This introduces a separate flow that deals with any fault.

Compensation This introduces a separate flow for undoing the work of a process.

3.3 A Lender Web Service

A loan service is a frequent example for business processes; the one here is based on that in the BPEL standard. LoanStar is a *lender* that offers a loan to an online customer, who makes a *proposal* containing name, address and loan amount. Deciding whether to make a loan may require time-consuming enquiries. If the amount is less than 10000, LoanStar asks its business partner RiskTaker to make a quick assessment via its web service. RiskTaker is an *assessor* that evaluates the risk of the loan. If the risk is low, LoanStar offers to lend at a basic rate of 3.5%. Otherwise, LoanStar asks its business partner FirstRate to determine a realistic loan rate. FirstRate is an *approver* that thoroughly evaluates a loan proposal; its loan rate is returned by LoanStar to the customer.

This example involves multiple web services: two partner web services (*assessor*, *approver*), and the business process itself (*lender*, see figure 2). The loan customer (the borrower) acts like a further web service, and may indeed be one.

Nearly everything needed to understand figure 2 has been explained earlier. A rounded rectangle contains a CRESS rule box, where **Uses** declares diagram variables like *proposal* and *risk*, and macros like *basicRate*.

All the web services happen to communicate via port *loan*, but the port names could vary. The borrower operation (nodes 1, 3, 5) is named *quote*, the approver operation (node 2) *approve*, and the assessor operation (node 6) *assess*.

In general, a CRESS configuration diagram defines the environment for services. For web services, it maps partners to namespace prefixes, namespace URIs (Uniform Resource Identifiers), and web service URIs. For example, for the lender service it might say:

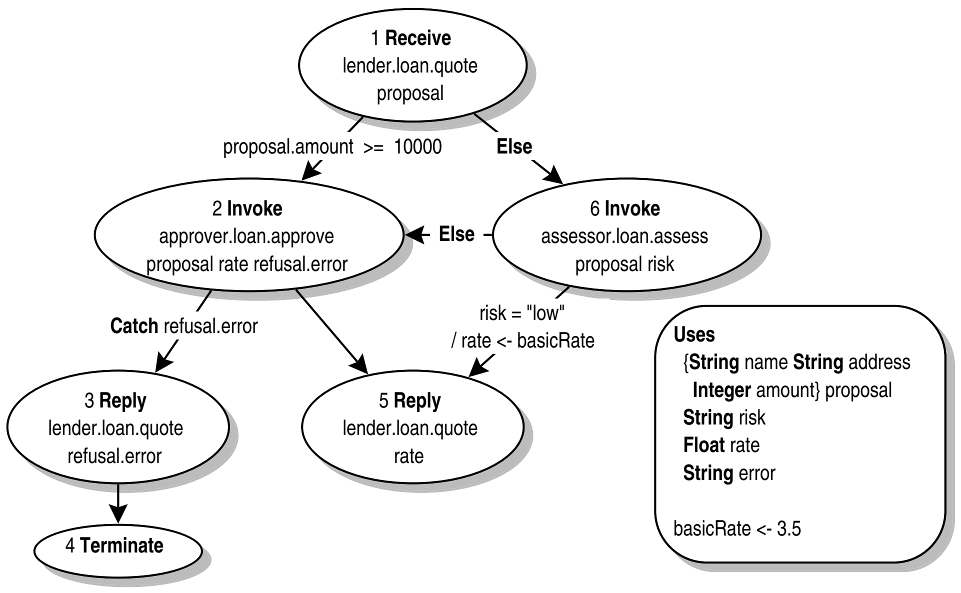

Figure 2: Lender Business Process

approver	app	urn:FirstRate	http://localhost:8080/services/Approver
assessor	ass	urn:RiskTaker	http://localhost:8080/services/Assessor
lender	lend	urn:LoanStar	http://localhost:8080/services/Lender

For the work reported here, web services were given simple URNs (Uniform Resource Names) and were deployed using ActiveBPEL on the local host. In practice, actual URIs would be used (e.g. *http://www.firstrate.com:8080/axis/services/Approver* for FirstRate). The definitions common to a service and its partners use namespace *defs*.

3.4 A Car Supplier Web Service

As a further example, DoubleQuote is a *supplier* that offers its online customers a good deal on car orders. A customer (the buyer) provides a *need* containing name, address and car model. This is passed to two dealers, each of which responds with an *offer* giving the dealer reference, name, car price, and days for delivery. DoubleQuote works with two business partners: BigDeal (acting as *dealer1*) and WheelerDealer (acting as *dealer2*). The better quote (i.e. lower price, or earlier delivery date if equal) is turned into a definite order. The offer is also returned to the buyer. A dealer indicates it cannot supply a car by replying with infinite price. (It would also have been possible to indicate this by throwing a fault.)

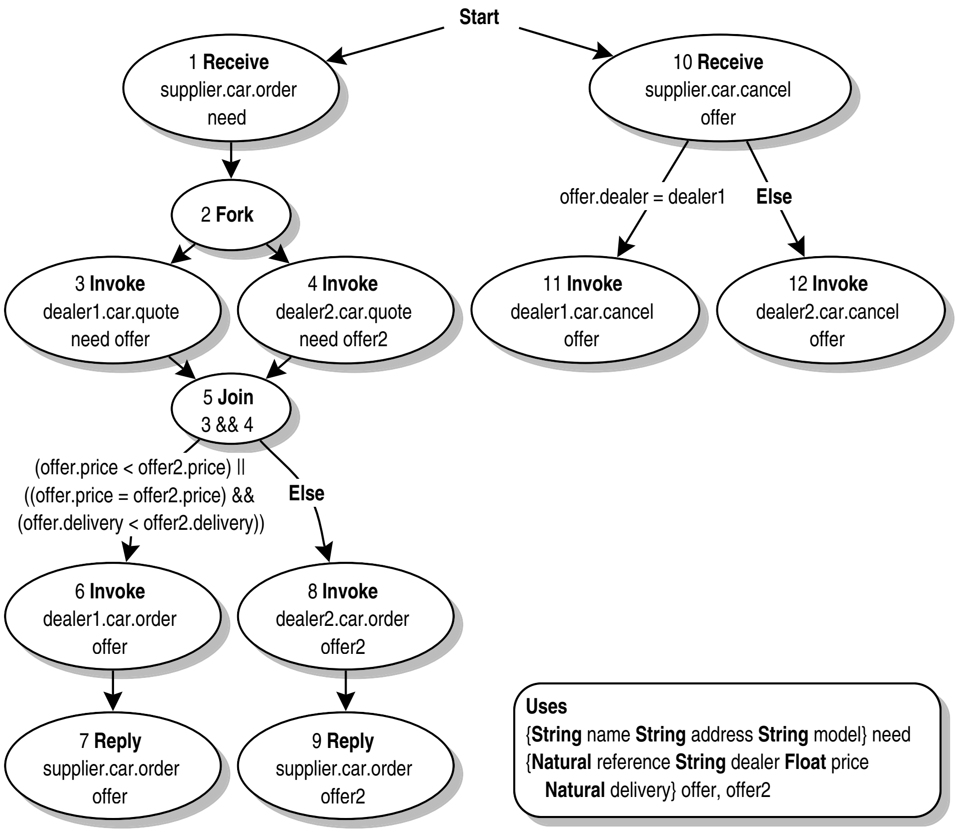

Figure 3: Car Supplier Business Process

Again, multiple web services are involved: the dealers (*dealer1*, *dealer2*) and the business process itself (*supplier*, see figure 3). All partners happen to use the same port *car*. The buyer operations are *order* (nodes 1, 7, 9) and *cancel* (node 10). The dealer operations are *quote* (nodes 3, 4), *order* (nodes 6, 8), and *cancel* (nodes 11, 12).

Figure 3 uses concepts explained earlier. The supplier obtains dealer quotations in parallel (nodes 2 to 5) in order to save time. Both quotes must be obtained (3 && 4 in node 5) for the quotation process to terminate successfully. Whichever dealer offer is selected leads to an order (nodes 6 to 9). Since the better offer is turned into a definite order, the order may have to be undone if the buyer renegues (or the calling web service faults). DoubleQuote therefore allows an order to be cancelled through the selected dealer (nodes 11, 12).

3.5 A Car Broker Web Service

As a final example, CarMen is a *broker* that provides an online service to negotiate car purchases and loans for these. Customers provide their *need* in the form of name, address and car model. CarMen first uses its business partner DoubleQuote (section 3.4) to order the car. If the car is unavailable (the price is infinite), CarMen informs its customer of refusal (by responding with a fault). Otherwise, CarMen now asks its business partner LoanStar (section 3.3) to arrange a loan for the car price. (Since a price is a float, it is rounded to a whole number before applying for a loan.) If a quote can be provided, the customer receives a *schedule* with the dealer reference, name, price, delivery period and loan rate.

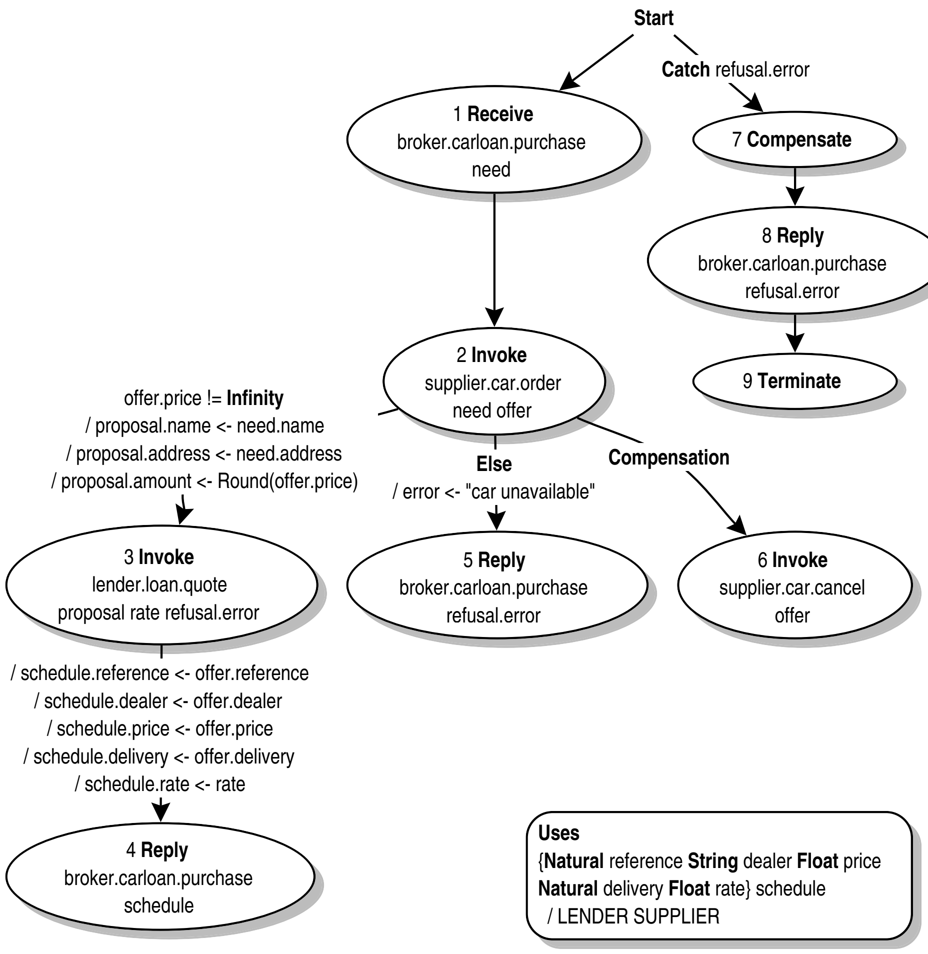

Figure 4: Car Broker Business Process

Now the situation with web services is very complex. The CarMen business process (*broker*, figure 4) indirectly invokes three web services for supplying the car (*supplier*, figure 3). The CarMen business process also indirectly invokes three web services to arrange a loan for it (*lender*, figure 2). The beauty of web services is that this is all invisible to CarMen's customer, who sees a single web service for ordering a car and receiving a schedule for its delivery and financing.

Figure 4 illustrates some additional constructs mentioned earlier. The **Uses** clause defines dependence on the partner business processes *lender* and *supplier* (named following '/'). If the *lender* throws a *refusal* fault (node 3), this is caught and compensation is invoked (node 7). The compensation associated with a car order (node 3) requires the *supplier* to cancel this (node 6). The same fault is then returned to the customer (node 8).

4 Translating CRESS Web Services to BPEL and WSDL

This section outlines the general principles behind translating CRESS web service descriptions into BPEL and WSDL. As an example, the generation of files for the lender service is explained.

4.1 Service Deployment

Web services require a considerable amount of XML that is generated automatically by CRESS. Translation and deployment of a CRESS business process is entirely automated, except for the one-off implementation of partner web services. Partner services are automatically deployed using AXIS, while the business process is automatically deployed using ActiveBPEL.

The most important generated code is the BPEL describing the business process. A WSDL definition is created for the business process since it is a web service in its own right. A WSDL file is also created for message and type definitions that are common to the business process and its partners.

The translation from CRESS to BPEL is fairly complex, partly because BPEL needs to be defined in a particular order, and partly because a lot of information has to be inferred by CRESS.

4.2 Handlers

CRESS has a uniform representation for handlers. A node may be governed by an event guard that relies on dynamic occurrence of some event. For example, **Catch** and **CatchAll** introduce a fault handler, while **Compensation** introduces a compensation handler.

In CRESS, handlers are either global or are associated with an **Invoke** (which is where faults or compensation, for example, are most likely to occur). Although BPEL4WS allows handlers to be defined globally for an entire business process, global compensation is quite problematic to support. For this reason, WS-BPEL does not support this capability. To avoid this restriction, a global handler in CRESS is in fact defined as part of the top-level flow rather than as part of the top-level process.

4.3 Service Flow

A web service may use a variety of constructs to describe the flow: conditions (*if*, *switch*), sequences (*sequence*), loops (*while*), arbitrary parallel flows (*flow*), and several kinds of handlers (event, fault, compensation, correlation). CRESS simplifies this to conditions (expression guards), arbitrary flows, and one kind of handler (event guard). A number of constructs used by BPEL are intentionally hidden by CRESS. For example scopes are implicit, and specialised constructs such as *onMessage* as opposed to *receive* are used implicitly by CRESS as required.

CRESS automatically determines and declares the links among activities, which are then chained using BPEL *source* and *target* elements. The BPEL function *getLinkStatus* is used to check whether a linked activity has terminated successfully. An awkward case to handle is assignment on a CRESS arc leaving a node. This has to be translated along with the prior node, dependent on whether the transition condition holds or not.

The flows of handlers, and therefore the links among their activities, are independent of the main flow. However by treating the top-level flow as the global level, CRESS is able to declare a single set of links that applies to both the main flow and the handlers.

4.4 Data Types

Data types in CRESS are either the usual ones defined by XML Schemas (e.g. float, integer) or are structures consisting of a number of fields.

The use of variables in BPEL is somewhat ugly. They are automatically characterised by CRESS as message variables (input, output) or data variables (assignment, expression). Unfortunately the syntax and usage of these is different in BPEL. CRESS generates XSD complex types for structured variables, and uses XSD built-in types for simple variables. Variables are used in expressions by the BPEL function *getVariableData*. Fields in structured variables are accessed by XPATH expressions.

4.5 Partner Services

The WSDL for partners is automatically generated from the CRESS diagrams, along with Web Service Deployment Descriptors. If a partner service already exists, its WSDL can be used directly. The CRESS view is likely to be a subset of the partner WSDL, since a business process is likely to use only certain ports and operations of an already defined partner. If a partner web service does not already exist, its WSDL is translated into Java using the AXIS tool *wsdl2java*. The skeleton partner service must then be implemented manually. If this is an existing legacy application, it is usually straightforward to give it a web service wrapping. If the application is new, then of course substantial implementation effort is required anyway. The partner web services in sections 3.3 to 3.5 were given plausible implementations in Java. If the CRESS translator finds an existing implementation (e.g. *approver.java*), it uses this instead of the default service skeleton.

4.6 The CRESS Toolset

CRESS is supported by a large toolset that accepts diagrams from three graphical editors: *Diagram!* from Lighthouse Design, *yEd* from yWorks, CRESS's own CHIVE editor. CRESS handles services in four domains: Intelligent Networks, Internet telephony, Interactive Voice Response, Web Services. CRESS translates diagrams into four target languages: BPEL/WSDL, LOTOS, SDL, VoiceXML.

The toolset is written in Perl for portability, and has been used on four different platforms (Linux, OpenStep, Solaris, Windows). The toolset comprises five main programs plus supporting modules – about 27,000 lines of code in total. As will be seen in section 5.2, CRESS is supported by the separate but related MUSTARD tool for validating services. The entire toolset is available for non-commercial, research purposes (*www.cs.stir.ac.uk/~kjt/research/cress.html*).

The structure of the CRESS toolset appears in figure 5; overlapping shapes indicate where variations may exist. The boxed area represents the CRESS toolset proper. The primary inputs are service (and feature) diagrams, drawn with one of several graphical editors. The CRESS tools are designed to be driven from a development environment for the target language. For example, a simple command (TOPO toolset for LOTOS) or a button click (Telelogic Tau for SDL) is used to invoke the translation and analysis.

Initially, the front-end tools for the target language (e.g. a graphical editor for LOTOS) are used to create a framework for the target language and application domain. For example, this might be LOTOS as used to described web services. The framework establishes everything that is common, such as the structure of the resulting specification, common data types, and common processes. The framework

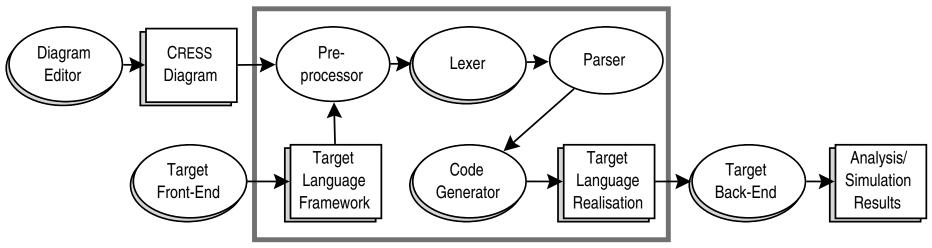

Figure 5: Structure of The CRESS Toolset

is specific to a language-domain combination, but is independent of the particular services. The framework has hooks for where service information must be added. The CRESS preprocessor uses the service diagrams to establish what must be added. It uses the CRESS lexer to convert the diagrams into neutral format, independent of any diagram editor. The CRESS parser then converts this information into an abstract syntax graph. (This is not a tree because diagrams may contain cyclic references.) The end result describes the entire set of services in language-independent form.

A CRESS code generator (of which there are several) translates the graph of services into the same target language as the original framework. This is then passed to the back-end tools for the target language (e.g. a LOTOS semantic analyser or model-checker). At a minimum, these tools check the syntactic and static semantic correctness of the generated specification. For formal languages, they support automated analysis through verification, validation and simulation. For implementation languages, they support automated execution.

4.7 Translation of the Lender Service

Figure 6 illustrates the procedure for translating CRESS diagrams into BPEL and WSDL. The example here is for the lender service in figure 2; the three columns show the files generated for the approver, the lender and the assessor.

BPEL and WSDL are created for the main lender service. A separate WSDL file is created for type definitions that are shared by the lender service and its partners. For use with ActiveBPEL, a catalogue of WSDL files and a process deployment descriptor are also generated.

WSDL is created for the two partner services (approver and assessor). If their implementations do not already exist, this WSDL is converted into skeleton Java implementations by *wsdl2java*.

Translation of just the lender service and its partner requires generating 25 source files totalling nearly 2,000 lines of code. It is therefore not practicable to

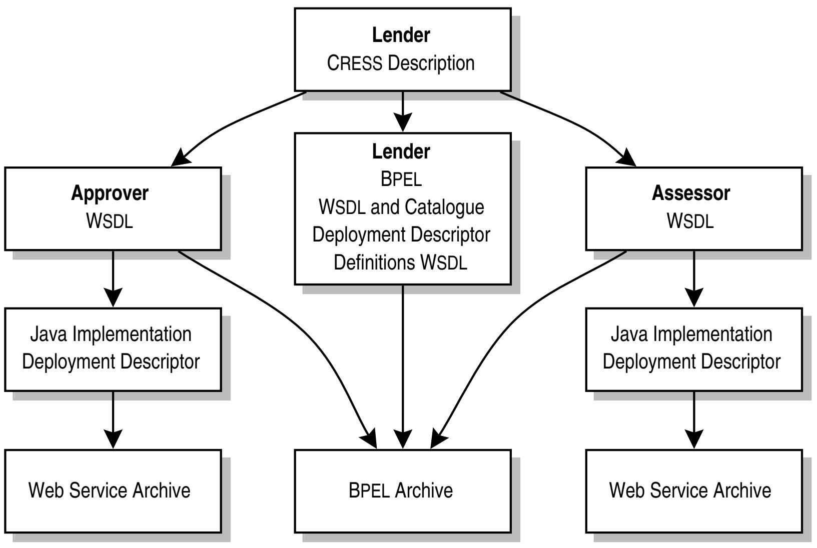

Figure 6: Translation of Lender

give extracts here. Instead, the code generated by the translation has been placed online at *www.cs.stir.ac.uk/~kjt/software/download/cress-lender.tar.gz*. The interested reader can download these files to see what the translation strategy described in this paper produces. The files can also be executed in a BPEL environment.

5 Analysing CRESS Web Service Descriptions

This section discusses the practical implications of formally analysing web services, and looks at scenario-based validation as a pragmatic approach.

5.1 Service Verification

After a CRESS description has been translated into a formal specification, a variety of formal analyses can be performed:

- State exploration allows the discovery of undesirable conditions such as deadlock (stalled progress) or livelock (unproductive loops). Because business processes combine separately developed web services, deadlock can readily arise if they are not fully interoperable. Systematic analysis discovers such deadlocks.
- Web services should also have desirable safety properties (nothing bad happens) and liveness properties (something good eventually happens). Con-

sider the *broker* service in figure 4. An invocation of it should not fault (safety), and every invocation of the service should eventually receive a response (liveness). Model-checking allows such properties to be verified against a service specification.

- WSDL can be written in many ways that reflect the same functionality. However, WSDL is purely syntactic and without a formal model. As a result, it can be problematic to check whether two WSDL descriptions are equivalent. The formal model of a service precisely describes its behaviour. From a formal methods point of view, it can be checked whether two specifications are equivalent in some sense. This is automated using a chosen notion of equivalence, e.g. whether the two have the same external behaviour (observational equivalence).
- The design of a web service is proprietary and may be confidential. For example, the car supplier in figure 3 may not wish to publicise which dealers are used. The owner of a web service, however, needs to publish a high-level description of the service. There is then a question of whether the high-level description is consistent with its detailed design. Formal checking of equivalence can determine this.
- A similar issue arises because services evolve over time. For example, the car supplier may later decide to get quotes from three dealers, or may decide to use different dealers. The question is then whether the new service is functionally equivalent to the old one. Again, automated equivalence checking can settle this.

In formal methods, such techniques are termed verification (proof). Unfortunately, state space explosion often limits what can be checked of realistic services.

5.2 Service Validation

Because of practical limits to verification, CRESS also makes use of validation (testing). In fact, both specifications and implementations are validated by the same means. The idea is to characterise the behaviour of services through use-case scenarios. If these scenarios deal with all critical aspects of a service, they can be used to check whether a service behaves as expected. Of course, such an approach is incomplete – drastically so, since typically only a tiny fraction of possible behaviour is checked. However, systematic testing methodologies can be used from fields such as hardware design, software engineering, and protocol conformance testing.

The author has developed MUSTARD (Multiple-Use Scenario Test and Refusal Description [20]) as a language-independent and tool-independent approach for

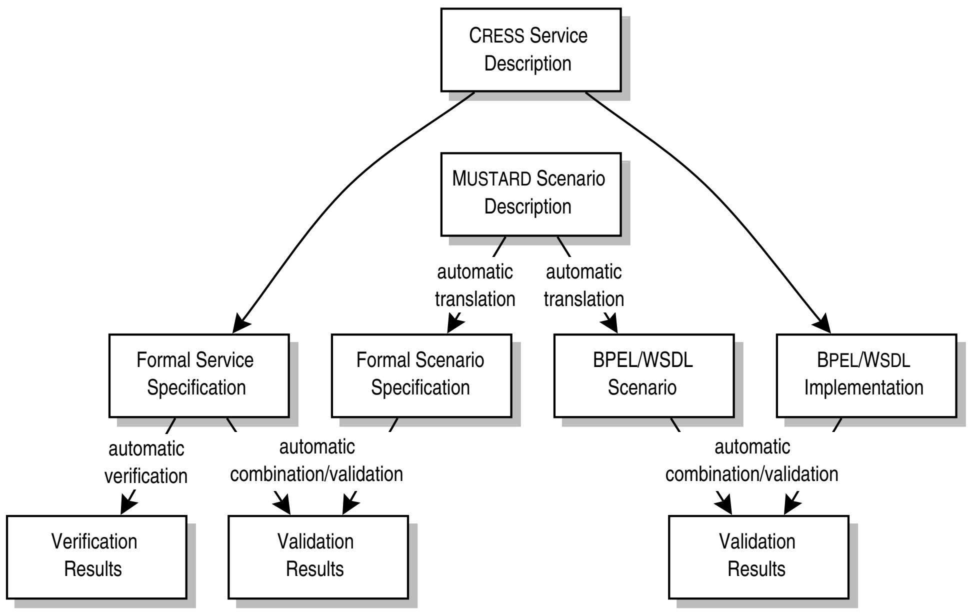
CRESS Service Description
MUSTARD Scenario Description
automatic translation
automatic translation
Formal Service Specification
Formal Scenario Specification
BPEL/WSDL Scenario
BPEL/WSDL Implementation
automatic verification
automatic combination/validation
automatic combination/validation
Verification Results
Validation Results
Validation Results

Figure 7: Scenario-Based Validation

expressing use-case scenarios. (Although MUSTARD is the culinary counterpart of CRESS and is typically used along with it, MUSTARD is also used independently to validate system descriptions in various domains.) The approach of MUSTARD is illustrated in figure 7. The left-hand side of this figure deals with a formal specification, while the right-hand side deals with an actual implementation.

Validation scenarios are described textually with an applicative (function-like) syntax or with an XML Schema-based syntax. The scenarios are translated into a language appropriate to what is being evaluated. For use with LOTOS, the scenarios are rendered in LOTOS. For use with SDL, the scenarios are rendered using MSCs (Message Sequence Charts [11]). For use with actual web services, the scenarios are rendered using BPEL and WSDL (i.e. as web services themselves).

When a formal specification is the target, the CRESS service description is automatically translated into its formalisation. This can be analysed by purely formal means, leading to verification results. MUSTARD scenarios are automatically translated into, say, LOTOS test processes. These are automatically combined with the main service specification, and are automatically checked.

The result of validation is a verdict on each scenario: pass, fail or inconclusive. A pass verdict means that the specification allows at least the behaviour of the scenario, while a fail verdict means this is not respected. Acceptance scenarios are designed to show that the expected happens, but this does not exclude extraneous behaviour. Refusal scenarios are designed to show that undesirable behaviour does not occur. It is possible for the same scenario to pass on some paths through the behaviour but to fail on others; such an outcome is regarded as inconclusive. Clearly a fail or inconclusive verdict requires re-examination of the CRESS service description, which may be incorrect or incomplete. Sometimes scenarios have to be corrected because they do not fully capture the core behaviour of the service.

After successfully validating scenarios, the developer should have confidence in the CRESS description. In theory, correct descriptions should lead to correct implementations. However, pragmatic problems may arise when implementations are deployed. For example there may be issues of usability, performance and independence.

There is insufficient space here to explain the MUSTARD notation, so reference to [20] and to the following example must suffice. Briefly, MUSTARD allows scenarios with sequences, alternatives, inter-service dependencies, non-determinism and concurrency. Since many web services act like RPCs (Remote Procedure Calls), their scenarios mainly check that the expected outputs are produced by the selected inputs. However, scenarios may involve several branched communications between the web service user and the service itself. Scenarios with concurrency are useful for finding race conditions, as well as for checking the service under load conditions. Both acceptance scenarios and refusal scenarios are valuable in determining the behaviour of a service at the edges of its ‘envelope’.

The following MUSTARD scenario (in applicative form) checks simultaneous requests to the *broker* process. (Strings are preceded by a single quote.) Two behaviours are checked in parallel (by interleaving) in case there are any unexpected interdependencies in the implementation:

- The first parallel branch offers a non-deterministic choice (test-decided) of two orders. A request for an Audi A5 expects to receive a schedule with dealer reference 8, name WheelerDealer, price 33000, delivery 30 days, loan rate 3.5%. Alternatively, a Renault Mégane may be ordered.
- The second parallel branch requests a Ford Mondeo. A deterministic choice (system-decided) then allows a specified schedule or an unavailable response in return.

```
test(Simultaneous_Purchases,                                    % simultaneous purchases scenario
  succeed(                                                      % behaviour must succeed
    interleave(                                                 % behaviours are interleaved
      decides(                                                  % non-deterministic choice
        sequence(                                               % Ken Turner buys an A5
          send(broker.carloan.purchase,Need('Ken Turner,'Stirling Scotland,'Audi A5)),
          read(broker.carloan.purchase,Schedule(8,'WheelerDealer,33000.,30,3.5))),
        sequence(                                                        % Sally Dean buys a Mégane
          send(broker.carloan.purchase,Need(′Sally Dean,′Cardiff Wales,′Renault Mégane)),
          read(broker.carloan.purchase,Schedule(1,′BigDeal,11000.,5,4.4)))),
      sequence(                                                      % Kurt Jenner requests a Mondeo
        send(broker.carloan.purchase,Need(′Kurt Jenner,′London England,′Ford Mondeo)),
        offer(                                                         % choice of schedule or fault
          read(broker.carloan.purchase,Schedule(6,′BigDeal,20000.,10,4.1)),
          read(broker.carloan.purchase,refusal,′car unavailable))))))
```

Of course, there is then the issue of where such scenarios come from. The author has separately developed PCL (Parameter Constraint Language [19]) for this kind of purpose. Fully automatic generation of useful tests from a complex specification is generally infeasible. Often input values may be chosen from an infinite set, of which only certain values are crucial to testing (cf. boundary testing). In a system with many inputs, the combinatorial combination of these inputs may yield an unworkable number of scenarios.

Instead, it is necessary to identify critical input values and their combinations using domain-specific knowledge. PCL is then used to annotate a specification with constraints on interesting input values and on useful orderings over inputs. This makes test generation practicable for specifications with complex data types, infinite data sorts or concurrency – all characteristic of web service specifications. Using algorithms described in [13, 19], the annotated specifications can be used to create useful validation scenarios.

5.3 Service Independence

[21] argues that interaction (i.e. interference) among web services is an integration issue. Integration is less of an issue if web service instances are truly independent and self-contained. However there are ways in which this assumption may be broken:

- Web service instances running on the same physical system are in competition and may suffer from resource interference or starvation.
- Partner web services may indirectly share resources and may therefore suffer from resource conflicts. As an example, the *approver* and *assessor* partners in figure 2 seem to be completely independent. In practice, they may rely on a common partner whose visibility is not apparent. For example, they may use a shared web service to validate the customer address, thereby leading to potential interference.
- A web service may appear to offer stateless (RPC) operations. For many reasons, including efficiency, web services may store the history of past operations. For example, the lender service may cache details of previous loan applications; a new loan application can then be evaluated more quickly.

However this negates the assumption of independence and may lead to interactions among different instances of the same service. For example if a customer is denied a loan because of a temporary overdraft, this may prevent getting a loan on a future occasion.

- Independence may also be violated if information about past operations is passed to other web service providers. For example, the lender service may make its loan assessments available to other web services. A previous failure to obtain a loan might thus cause a mail-order web service to refuse an order.

If the full descriptions of services are available, such issues can be discovered through analysis. However, this is often not possible because service descriptions are usually proprietary and unavailable. For this reason, violations of service independence may have to be checked empirically. The validation approach supported by CRESS allows this to be assessed in practice.

5.4 Feature Interaction

CRESS allows web services to be extended by means of features. Consider the sample web services discussed earlier. They all make use of a customer name and address. In fact, this is likely to be the case with many customer-oriented services. It would be useful to validate the name and address provided.

As a sample feature, figure 8 matches any **Receive** for a *quote*, *order* or *purchase* operation. (The asterisks denote any partner or port.) The formal parameter *details* is matched to the actual parameter of the **Receive** (*need* or *proposal* for the examples in this paper). The '+' after the number in node 1 means that the feature behaviour is appended to the **Receive**. This invokes the *normalise* operation of the external *checker* partner with the given name, receiving the name back in normalised form. For example, the author's name is normalised to 'KJ Turner' and stored in the name field of *details*. The feature then finishes, continuing with whatever followed the triggering **Receive**.

As a further sample feature, figure 9 similarly matches a **Receive** operation. This invokes the *check* operation of the *checker* partner to take the given name and address, receiving in return a check on whether the name-address combination is valid. This service might, for example, check a credit card database. The feature finishes if the combination is valid, allowing normal behaviour to continue. Otherwise it throws a *wrongContact* fault.

These features are automatically deployed by CRESS into the web services described earlier. There is an immediate question of whether a name should be normalised before being checked against an address. This is desirable, and is achieved in CRESS by associating features with priorities.

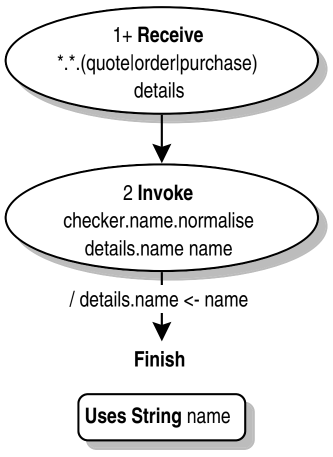

Figure 8: Name Normalising Feature

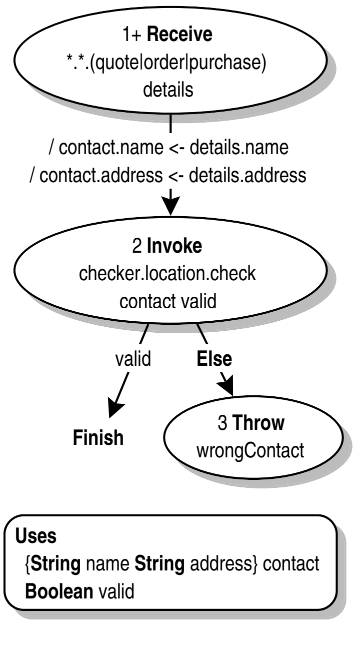

Figure 9: Address Checking Feature

However, the introduction of features may lead to interference among them. In fact, the name normalisation and address checking features may interact with each other. For example, the normal form of the author's name does not match the name known to his credit card company. As a result, normalising the name might hinder address checking.

This kind of problem is detected automatically during validation with CRESS. A common interpretation of feature interaction is that a feature behaves differently in the presence of other features. CRESS allows the behaviour of features to be validated in isolation (thus confirming their functional correctness) and also in combination (thus confirming absence of interactions).

6 Conclusions

It has been argued that isolated web services can benefit from formal models of their behaviour. A graphical description of business processes helps to make them more understandable. A high degree of automation is strongly desirable in the creation of web services. CRESS meets all of these requirements.

Compared to commercial tools, CRESS does not support all of BPEL (though it handles most things that are used in practice). However CRESS confers distinctive benefits: applicability to many types of services, human-usable automated implementation, service features, and translation to formal languages for rigorous analysis.

CRESS is complementary to current development practices for web services. The introduction of a compact graphical notation and rigorous analysis are believed to be beneficial additions. In future work, CRESS will be extended to cover more of BPEL. It is also planned to apply CRESS to the composition of grid services, which closely resemble web services.

References

[1] A. V. Aho, S. Gallagher, N. D. Griffeth, C. R. Schell, and D. F. Swayne. SCF3/Sculptor with Chisel: Requirements engineering for communications services. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 45–63. IOS Press, Amsterdam, Netherlands, Sept. 1998.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, editors. *Business Process Execution Language for Web Services*. Version 1.1. BEA, IBM, Microsoft, SAP, Siebel, May 2003.

[3] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Lie, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language*. Version 2.0 (Draft). Organization for The Advancement of Structured Information Standards, Billerica, Massachusetts, USA, Feb. 2005.

[4] BPMI. *Business Process Modeling Notation*. Version 1.0. Business Process Management Initiative, May 2004.

[5] A. Chirichiello and G. Salaün. Encoding abstract descriptions into executable web services: Towards a formal development. In *Proc. Web Intelligence 2005*. Institution of Electrical and Electronic Engineers Press, New York, USA, Dec. 2005.

[6] A. Ferrara. Web services: A process algebra approach. In *Proc. 2nd. International Conference on Service-Oriented Computing*, pages 242–251. ACM Press, New York, USA, Nov. 2004.

[7] H. Foster, S. Uchitel, J. Kramer, and J. Magee. Model-based verification of web service compositions. In *Automated Software Engineering 2003*, Montreal, Canada, Oct. 2003.

[8] H. Foster, S. Uchitel, J. Kramer, and J. Magee. Compatibility verification for web service choreography. In *2nd. International Conference on Web Services*, San Diego, California, USA, July 2004.

[9] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling web services interoperability. In *Proc. 6th International Conference on Enterprise Information Systems*, Porto, Portugal, Apr. 2004.

[10] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.

[11] ITU. *Message Sequence Chart (MSC)*. ITU-T Z.120. International Telecommunications Union, Geneva, Switzerland, 2000.

[12] ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 2000.

[13] C. Jard and T. Jéron. TGV: Theory, principles and algorithms. *Software Tools for Technology Transfer*, pages 297–315, Aug. 2005.

[14] S. Staab, W. van der Aalst, V. R. Benjamins, A. Sheth, J. A. Miller, C. Bussler, A. Maedche, D. Fensel, and D. Gannon. Web services: Been there, done that. *IEEE Intelligent Systems*, 18(1):72–85, Jan. 2003.

[15] K. J. Turner. Formalising the CHISEL feature notation. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 241–256. IOS Press, Amsterdam, Netherlands, May 2000.

[16] K. J. Turner. Modelling SIP services using CRESS. In D. A. Peled and M. Y. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XV)*, number 2529 in Lecture Notes in Computer Science, pages 162–177. Springer, Berlin, Germany, Nov. 2002.

[17] K. J. Turner. Analysing interactive voice services. *Computer Networks*, 45(5):665–685, Aug. 2004.

[18] K. J. Turner. Formalising web services. In F. Wang, editor, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XVIII)*, number 3731 in Lecture Notes in Computer Science, pages 473–488. Springer, Berlin, Germany, Oct. 2005.

[19] K. J. Turner. Test generation for radiotherapy accelerators. *Software Tools for Technology Transfer*, 7(4):361–375, Aug. 2005.

[20] K. J. Turner. Validating feature-based specifications. *Software Practice and Experience*, May 2005. In press.

[21] M. Weiss. Feature interactions in web services. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 149–156. IOS Press, Amsterdam, Netherlands, June 2003.

[22] M. Weiss and B. Esfandari. On feature interactions in web services. In *Proc. IEEE International Conference on Web Services*, pages 88–95, San Diego, California, July 2004.

[23] World-Wide Web Consortium. *Simple Object Access Protocol (SOAP)*. Version 1.2. World-Wide Web Consortium, June 2003.

[24] World-Wide Web Consortium. *Web Ontology Language (OWL) – Overview*. Version 1.0. World-Wide Web Consortium, Geneva, Switzerland, Feb. 2004.

[25] World-Wide Web Consortium. *Web Services Choreography Description Language*. Candidate Version 1.0. World-Wide Web Consortium, Geneva, Switzerland, Nov. 2005.

[26] World-Wide Web Consortium. *Web Services Description Language (WSDL)*. Version 2.0 (Draft). World-Wide Web Consortium, Aug. 2005.