Kenneth J. Turner. Formalising Web Services. In Fern Wang, editor,
Proc. Formal Techniques for Networked and Distributed Systems (FORTE XVIII),
Lecture Notes in Computer Science 3731, pages 473-488,
Copyright Springer, Berlin, October 2005.

Formalising Web Services

Kenneth J. Turner

Computing Science and Mathematics, University of Stirling, Scotland FK9 4LA
Email kjt@cs.stir.ac.uk

Abstract. Despite the popularity of web services, creating them manually is an intricate task. Composite web services are defined using the evolving standard for BPEL (Business Process Execution Logic). It is explained how CRESS (Chisel Representation Employing Systematic Specification) has been extended to meet the challenge of graphically and formally describing web services. Sample CRESS descriptions are presented of web services. These are automatically translated into LOTOS, permitting rigorous analysis and automated validation.

1 Introduction

1.1 Background

Web services have become a popular way of providing access to distributed applications. These may be legacy applications given a web service wrapping, or purpose-designed applications. This paper describes an unusual application of formal methods (LOTOS) to modern developments in communications systems (web services).

The interface to a web service is defined in WSDL (Web Services Description Language). However this is purely syntactic and does not define the semantics of a web service. Although WSDL can be manually created and edited, this is an intricate and error-prone task. For this reason, most commercial solutions aim to create WSDL automatically from the code of an application.

WSDL describes an *isolated* web service. The current thrust in web service research is on composing them into what are called *business process*. (Other terms used include business flow and web service choreography.) Assume that the following web services exist: airlines take flight bookings, hotels reserve rooms, car hire firms book vehicles, and banks accept electronic payments. A travel agency can then build a business process that arranges all these aspects of a trip through a single web service.

Unfortunately, many competing standards emerged for composing web services. Harmonisation was achieved with the multi-company specification for BPEL4WS (Business Process Execution Language for Web Services [1]). This is being standardised as WS-BPEL (Web Services Business Process Execution Language [2]). BPEL4WS is stable, and has been used for most of the work reported here. However it has shortcomings, so WS-BPEL has also been used for reference. For brevity, this paper refers to BPEL and web services with all the interpretations discussed above.

BPEL is a recent and evolving language, so tool support is still developing. It can be very difficult to understand a complex flow from the XML in BPEL. A graphical view of composed web services is thus very desirable. BPMN (Business Process Modeling Notation [3]) has been developed to give a high-level graphical view of such services.

This paper emphasises the *composition* of web services, not the description of *isolated* web services. This is partly because web service creation is now well automated, and partly because many web services already exist. Composing web services, i.e. defining web-based business processes, has attracted considerable industrial interest.

The author has previously developed CRESS (Chisel Representation Employing Structured Specification) as a general-purpose graphical notation for services. CRESS has been used to specify and analyse voice services from the IN (Intelligent Network) [6], Internet Telephony [7], and IVR (Interactive Voice Response) [8]. Service descriptions in CRESS are graphical and accessible to non-specialists. A major advantage of CRESS descriptions is that they are automatically translated into formal languages for analysis, as well as into implementation languages for deployment. CRESS offers benefits of comprehensibility, portability, rigorous analysis and automated implementation.

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. CRESS is designed to be extensible, with plug-in modules for each application domain and each target language. Substantial work has been required because web services are quite distinctive. However, adding web services as a new CRESS domain has benefited from much of the existing CRESS framework. For example, CRESS has explicit support for features that allow a base service to be extended in a modular manner. The existing CRESS lexical analyser, parser and code generators have also been reused for web services.

The work described in this paper discusses how composed web services are represented using CRESS and translated into LOTOS. This automatically creates formal models of web services, and allows them to be rigorously analysed. Since web developers are unlikely to be familiar with formal methods, the use of LOTOS is hidden as much as possible in the approach. CRESS descriptions can be formally validated without seeing or understanding the underlying LOTOS. In additional work not reported here, the *same* CRESS descriptions of web services are automatically translated into BPEL and WSDL for implementation and deployment of web services.

1.2 Relationship to Other Work

Web services are well established and are widely supported by commercial tools; it would not be sensible to try competing with these. However the focus of this paper is on web service composition. Due to the relative newness of BPEL, support is only now maturing. Major products include IBM's WebSphere, Microsoft's BizTalk Server, Active EndPoint's ActiveBPEL, and Oracle's BPEL Process Manager. None of these provides a formal basis or rigorous analysis.

BPMN can be viewed as a competitor notation to CRESS for describing web services. However, BPMN is a very large notation (the standard runs to almost 300 pages). It also has a single purpose: describing business processes. BPMN is only a front-end for creating web services; tool support for creating (say) BPEL is only now emerging. In contrast, CRESS is a compact and general-purpose notation that has now been proven on services from four different domains. CRESS offers automated translation to formal languages (e.g. LOTOS, SDL) as well as to implementations (e.g. BPEL, VoiceXML). CRESS also introduces a feature concept that is lacking in other web service approaches.

There has been only limited research on formalising web services. [4] is closest to the present paper. This work supports automated conversion between BPEL and LOTOS.

CRESS differs in using a more abstract, graphical description that is translated into BPEL and LOTOS; there is no interconversion among these representations.

LTSA-WS (Labelled Transition System Analyzer for Web Services [5]) 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. CRESS may be used with any analytic technique using on the formal languages it supports, although it offers its own approach based on scenario validation.

The CRESS notation is described and illustrated elsewhere (e.g. [6–8]). Only a brief overview is therefore given here; the notation is explained through examples. Section 2 illustrates how CRESS is used to describe business processes. Section 3 outlines the translation of CRESS service descriptions into LOTOS. Section 4 shows how the resulting specifications can be formally analysed in a variety of useful ways.

2 CRESS Description of Business Processes

A brief introduction is given to the concepts of business processes. The CRESS representation of these is then explained, mainly with reference to some realistic examples.

2.1 CRESS for Business Processes

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 immediate response) or *asynchronously* (a request followed by a later 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* applies where work has to be undone due to a fault (e.g. a partial travel booking has to be cancelled). *Correlation* is used to link asynchronous messages to the correct business process instance.

A CRESS diagram is a directed graph that shows the flow of activities. In BPEL terms, a CRESS diagram defines an executable business process. Numbered nodes in a CRESS diagram correspond to BPEL activities. These are inputs and outputs (communications with other web services) or actions (internal to the web service). A BPEL activity is considered to terminate successfully or to fail (due to a fault).

In a CRESS diagram, arcs (BPEL links) join the nodes. CRESS nodes and arcs may have assignments in the form / *variable* <– *expression*. Arcs may be labelled by expression guards or event guards. Expression guards control alternative choices (switches in BPEL). Event guards introduce behaviour that is conditional on some event occurring (handlers in BPEL). The CRESS concept of event encompasses BPEL events, faults, requests for compensation and correlation requests.

For business processes, CRESS is required to offer sophisticated flow of control. Branches in a CRESS diagram normally reflect alternatives. However business processes need fine-grained control over parallelism. Although BPEL has separate constructs for sequence, iteration and graph-like flows, CRESS models them all in a uniform way.

2.2 CRESS for Business Activities

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*, **Natural** *n*, **String** *s*). CRESS can also define structured types, e.g. the following that defines two *offer* variables:

{**Natural** reference **String** dealer **Float** price **Natural** delivery} offer, offer2

Such a structured type is named implicitly after the first variable: *Offer*. Structured variables accesses have the form *offer.price*.

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; a correlation handler is used to match incoming messages to the correct instance. Each such **Receive** is 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 a response. 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** must terminate successfully. However, an explicit join condition may be defined over the termination status of parallel activities. In CRESS, the expression 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 activities prior to 1, 2 and 3 must also succeed.

Throw *fault* This reports a fault as an event to be caught elsewhere by a fault 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 a CRESS event handler to be invoked. In BPEL these may be defined inside any scope of a process. In CRESS, scopes are implicit. As a consequence, event handlers may only be global or associated with an **Invoke**. (This is a small restriction that accords with common BPEL practice anyway.) The handlers appearing in this paper are as follows:

Catch *fault* This defines how to handle the specified fault. If a fault has just a name and no value, it is handled by a **Catch** with a matching fault name only. A fault with name and value is handled by a **Catch** with matching fault name and variable type, otherwise by a **Catch** without a fault name but a matching type of fault value. (Although not illustrated in this paper, **CatchAll** handles any fault.) A fault handler applies where it is defined, and to subsidiary activities. If a fault occurs, it is considered by the current scope; if unmatched, it is considered by higher-level scopes until a matching handler is found. No match for a fault terminates the application.

Compensation This defines how to undo work due to a fault. A compensation handler applies only where it is defined, and is enabled only once the corresponding activity completes successfully. If a compensation handler is executed, it expects to see the process state at the time it was enabled. It also cannot alter the current process state. In effect, the process must maintain a stack of compensation states.

2.3 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 submits a *proposal* containing name, address and loan amount. If the amount is 10000 or more, LoanStar asks its business partner FirstRate to perform a full assessment. FirstRate is an *approver* that thoroughly evaluates a loan proposal. The loan rate it determines is returned by LoanStar to its customer. FirstRate may cause a *refusal* fault (e.g. error message ‘unacceptable’) because a loan cannot be offered.

A full assessment is costly, so a loan for less than 10000 is evaluated more simply. LoanStar asks its business partner RiskTaker to make a simple assessment. RiskTaker is an *assessor* that evaluates the risk of a loan. If the risk is low, LoanStar offers to lend at a basic rate of 3.5%. If the risk is not low, LoanStar asks FirstRate for a full assessment.

This example involves multiple web services: two partner web services (*assessor*, *approver*), and the business process itself (*lender*). The loan customer acts like a web service, and may be one. The CRESS description of the business process is in figure 1. The concepts needed to understand this have been explained earlier. Nodes (inputs, outputs, actions) in ellipses are linked by arcs (plain or guarded). If the *approver* invocation causes a *refusal* fault (node 2), this is caught by the associated handler (node 3).

The rounded rectangle at the bottom right of figure 1 is a CRESS rule box. **Uses** declares diagram variables, here *proposal*, *risk*, *rate* and *error*. Rule boxes have other purposes such as defining macros, event-triggered assignments and subsidiary diagrams.

An input or output names the partner, port and operation (e.g. *lender.loan.quote*). In this example, all the web services happen to communicate via port *loan*, but the port names could vary among services. The lender operation is *quote*, the approver operation is *approve*, and the assessor operation is *assess*.

2.4 A Car Supplier Web Service

As a further example, DoubleQuote is a *supplier* that offers online customers a good deal on car orders. A customer provides a *need* containing name, address and car model.

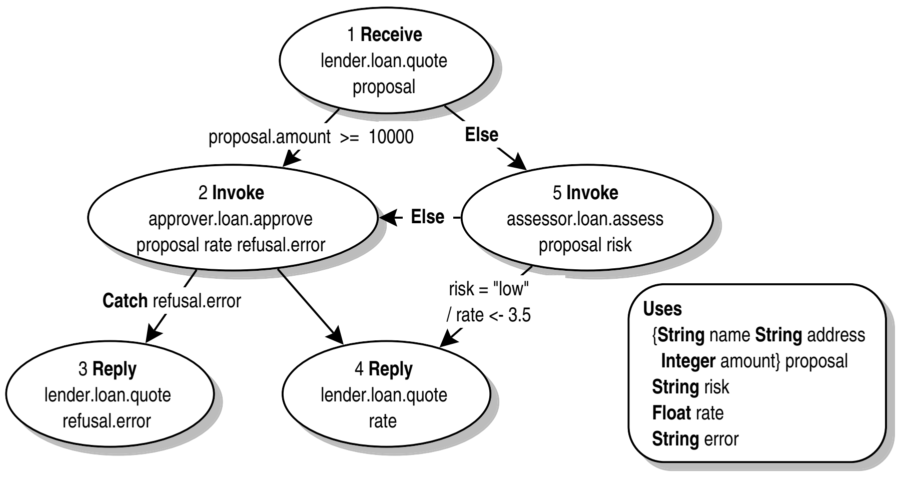

Fig. 1. Lender Business Process

The request for a quotation is passed to two dealers, each of which responds with an *offer* giving the dealer reference, name, price and delivery period.

DoubleQuote works with two business partners: BigDeal (acting as *dealer1*) and WheelerDealer (acting as *dealer2*). A dealer indicates that it cannot supply the model by replying with infinite price. (It would alternatively be possible to signal this by a fault.) The better offer is selected: the lower price, or the earlier delivery date if equal. This offer is sent to the appropriate dealer as a definite order. If necessary, the customer may later cancel the order corresponding to the selected offer.

Again, there are multiple web services: the dealers (*dealer1*, *dealer2*), the business process itself (*supplier*), and possibly the customer. The CRESS description of *supplier* is in figure 2. All partners happen to have the same port name *car*. The supplier operations are *order* and *cancel*, while the dealer operations are *quote*, *order* and *cancel*.

In figure 2, 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 a reply (node 7 or 9). Since a definite order is placed, it may be necessary to undo this if the DoubleQuote buyer renegues (or the calling web service faults). DoubleQuote therefore allows a previous order to be cancelled by the relevant dealer (nodes 10 to 12).

2.5 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. A customer provides a *need* with name, address and car model. CarMen first uses its business partner DoubleQuote (section 2.4) to order the car on the best terms. If the car is unavailable (the price is infinite), CarMen informs its customer of refusal by causing a fault with error message ‘car unavailable’. Otherwise, CarMen asks its business partner LoanStar (section 2.3) to arrange a loan for the car price. If a loan can be provided, the customer receives a *schedule* containing the dealer

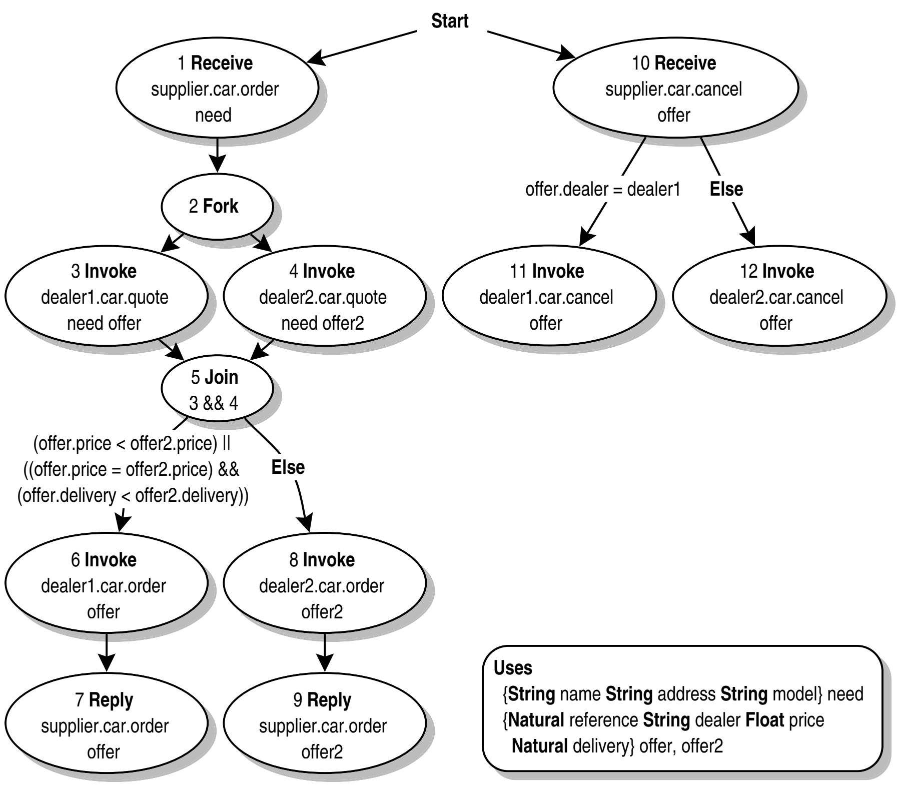
Start
1 Receive supplier.car.order need
2 Fork
3 Invoke dealer1.car.quote need offer
4 Invoke dealer2.car.quote need offer2
5 Join 3 && 4
(offer.price < offer2.price) || ((offer.price = offer2.price) && (offer.delivery < offer2.delivery))
Else
6 Invoke dealer1.car.order offer
8 Invoke dealer2.car.order offer2
7 Reply supplier.car.order offer
9 Reply supplier.car.order offer2
10 Receive supplier.car.cancel offer
offer.dealer = dealer1
Else
11 Invoke dealer1.car.cancel offer
12 Invoke dealer2.car.cancel offer
Uses
{String name String address String model} need
{Natural reference String dealer Float price Natural delivery} offer, offer2

Fig. 2. Car Supplier Business Process

reference, name, price, delivery period and loan rate. If a loan is refused (e.g. because the customer financial record is bad), a loan refusal fault will occur. Since the car has already been ordered, compensation requires the order to be cancelled. The refusal is then returned to the customer.

The CRESS description of this business process is in figure 3. This time, the **Uses** clause also references the subsidiary services *lender* and *supplier*. If the *lender* invocation in node 3 causes a *refusal* fault, it is intercepted by the global fault handler (nodes 7, 8). This calls the compensation handler in node 6 and returns the fault to the customer.

The situation with web services is now very complex. The *broker* (figure 3) invokes the *supplier* to order the car (figure 2) and the *lender* to arrange a loan (figure 1). In turn, each of these invokes two further web services. A total of seven web services is therefore involved. The beauty of web services is that this is all invisible to CarMen's customer, who sees a single web service for ordering and financing the purchase of a car. In fact, the internal details of a business process are intentionally hidden since this is confidential. This also allows businesses to change their internal procedures, e.g. the supplier may change dealers or may use more than two dealers.

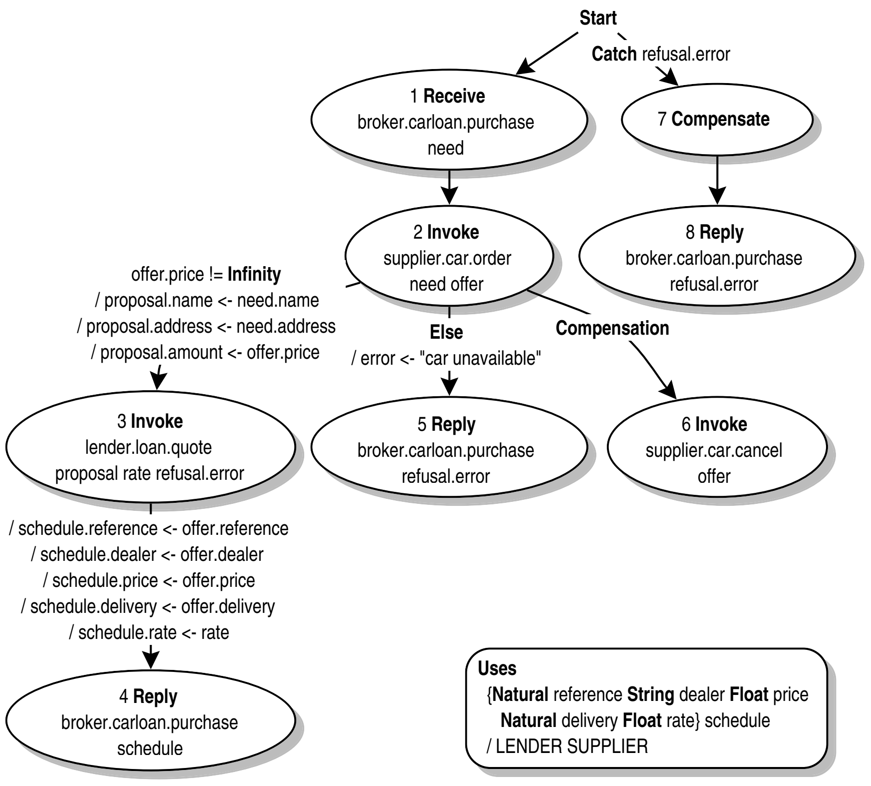

Fig. 3. Car Broker Business Process

3 Translating Web Services to LOTOS

The general principles of translating CRESS diagrams into LOTOS are explained in [6, 8]. The generated code is neatly laid out and well commented. The CRESS framework is largely reusable for web services. However, web services have distinct characteristics that require extension to this approach. The translation strategy is illustrated in this section with extracts from the LOTOS generated by the examples in figures 1, 2 and 3.

3.1 Data Handling

BPEL simple types are translated into a limited range of LOTOS types. BPEL *boolean* corresponds to LOTOS *Bool*, BPEL *natural* to LOTOS *Nat*, and variations on BPEL *string* to LOTOS *Text*. Other numeric types in BPEL are mapped to LOTOS type *Number*. Numbers are problematic to handle in LOTOS since floating point numbers are required. BPEL 1.1 allows floating point variables, but fortunately requires only simple integer arithmetic. Text strings are also awkward in LOTOS since there is no character type. LOTOS has no lexical shorthands for numbers or strings, so an ugly syntax is required; their conventional form is shown in the code extracts that follow.

Expressions are translated into their obvious LOTOS equivalents. BPEL uses XPATH as its expression language, and so has access to a wide range of functions. The LOTOS framework has support for those required by BPEL 1.1, i.e. a subset of the arithmetic, logical and string functions in XPATH 1.0. Expression guards become LOTOS guarded choices. Assignments are turned into LOTOS **Let** statements.

BPEL requires use of structured variables. Each structured type is automatically translated into a LOTOS type with fields as operations. For example, *proposal* in figure 1 generates the type *Proposal*, with field operations such as *getName* and *setName*.

3.2 Basic Behaviour

Outputs (**Reply**, **Invoke**) and inputs (**Receive**, **Invoke**) correspond to LOTOS events. An activity sequence in a CRESS diagram becomes a sequence in LOTOS. However, parts of a CRESS diagram often have to be translated as separate LOTOS processes. This happens, for example, when part of a diagram is reached by different paths or is invoked as an event handler. A BPEL activity results in successful termination or failure. LOTOS behaviours therefore exit with state *True* or *False*. For simple behaviours, this is the *States* result of a process. It will be seen later that states are generalised when dealing with compensation handling or with concurrency.

All the aspects considered so far are illustrated in the following code for nodes 1, 2 and 5 in figure 1:

```
Process LENDER_1 [lender,approver,assessor]                        (* LENDER from 1 *)
 (error:Text,proposal:Proposal,rate:Number,risk:Text) : Exit(States) :=
  lender !loan !quote ?proposal:Proposal;                          (* LENDER receive 1 *)
  (
    [getAmount(proposal) Ge 10000]->                (* check proposal.amount >= 10000 *)
      LENDER_2 [lender,approver,assessor]                  (* LENDER invoke 2 (again) *)
        (error,proposal,rate,risk)
  []
    [Not(getAmount(proposal) Ge 10000)]->       (* Else after proposal.amount >= 10000 *)
      assessor !loan !assess !proposal;                     (* LENDER invoke 5 request *)
      assessor !loan !assess ?risk:Text;                   (* LENDER invoke 5 response *)
      (
        [risk Eq "low"]->                                       (* check risk = "low" *)
          (
            Let rate:Number = 3.5 In                                (* update local *)
              LENDER_4 [lender,approver,assessor]           (* LENDER reply 4 (again) *)
                (error,proposal,rate,risk)
          )
      []
        [Not(risk Eq "low")]->                             (* Else after risk = "low" *)
          LENDER_2 [lender,approver,assessor]              (* LENDER output 2 (again) *)
            (error,proposal,rate,risk)
      )
  )
EndProc                                                            (* end LENDER_1 *)
```

3.3 Event Handling

For each web service, the CRESS translator statically discovers where event handlers are defined and the scopes where these apply (global, or associated with an **Invoke**). An event dispatcher process is then generated with reference to these handlers according to their scopes. If a fault handler does not exist for the current scope, the global handler (if any) is tried. Faults have to be matched against handlers in a particular order: **Catch** with a matching fault name, **Catch** with a matching fault name and type, **Catch** with a matching fault type, **CatchAll**. A fault means unsuccessful termination, so event handlers always exit with a *False* status.

A **Compensate** action, a **Throw** action or a fault invokes the event dispatcher with information about the scope, fault name and fault value type. The fault handling rules of BPEL require fault values to be coerced into a single LOTOS type *Value*. This is needed so that the kind of value can be matched against **Catch**. For example, a fault handler expecting a string must check if the value is indeed a string; another handler for the same fault name might deal with floating point fault values.

As an example, **Invoke** in node 2 of figure 1 may generate a *refusal* fault. This calls the *LENDER_EVENT* dispatcher for scope 0 associated with node 2; there is just one event scope in this example. The *Match* operation compares the given fault name and value type with those in the event (*refusal* and *Text* in this case). When node 3 is called, the fault value (*error*) is set to a string by operation *Text*.

```
Process LENDER_2 [lender,approver,assessor]                          (* LENDER from 2 *)
 (error:Text,proposal:Proposal,rate:Number,risk:Text) : Exit(States) :=
  approver !loan !approve !proposal;                         (* LENDER invoke 2 request *)
  (
    approver !loan !approve !refusal ?error:Text;              (* LENDER invoke 2 fault *)
    LENDER_EVENT [lender,approver,assessor]                  (* call event dispatcher *)
      (error,proposal,rate,risk,0 Of Nat,refusal,Value(error))
  []
    approver !loan !approve ?rate:Number;                   (* LENDER invoke 2 response *)
    LENDER_4 [lender,approver,assessor]                      (* LENDER reply 4 (again) *)
      (error,proposal,rate,risk)
  )
EndProc                                                               (* end LENDER_2 *)
Process LENDER_EVENT [lender,approver,assessor]                     (* event dispatcher *)
 (error:Text,proposal:Proposal,rate:Number,risk:Text, scope:Nat,event:Event,value:Value) :
   Exit(States) :=
  [scope Eq 0]->                                                          (* scope 0 ? *)
    (
      [Match(event,kind,refusal,TextKind)]->               (* match for 'refusal.error'? *)
        LENDER_3 [lender,approver,assessor]                     (* call event handler *)
         (Text(value),proposal,rate,risk)
    )
EndProc                                                           (* end LENDER_EVENT *)
```

Compensation handling is much more complex to translate than fault handling. A compensation handler becomes available only when its associated scope has terminated successfully. The state of the process must also be stored for use by the compensation handler in case it is called later. When compensation is in use, LOTOS processes must therefore carry a *states* parameter as the history of compensation states.

As each activity with compensation completes, it prefixes the current state (i.e. the process parameters) to the previous state list. In this way, a stack of compensation states is maintained. The following extract is from nodes 1 and 2 of figure 3. The first parameter of operation *State* is a *True* status (all that is used in simple processes), while the second parameter is the compensation scope (1 in this case, 0 being the global scope).

```
Process BROKER_1 [broker,supplier,lender]                    (* BROKER from start *)
 (error:Text,need:Need,offer:Offer,proposal:Proposal,rate:Number,
  schedule:Schedule,states:States) : Exit(States) :=
    broker !carloan !purchase ?need:Need;                    (* BROKER receive 1 *)
    supplier !car !order !need;                       (* BROKER invoke 2 request *)
    supplier !car !order ?offer:Offer;               (* BROKER invoke 2 response *)
    (
      Let states:States =                                        (* store state *)
        State(True,1,error,need,offer,proposal,rate,schedule) + states In ...
    )
EndProc                                                       (* end BROKER_1 *)
```

A **Compensate** action for a given scope invokes the event dispatcher. This searches the stored states for a matching compensation state. If found, the handler for this state is called. If not found (or no scope was specified by **Compensate**), the default action is to call all compensation handlers in reverse order of activity completion. The net effect is that compensation undoes previous work. In figure 3, for example, failure to obtain a loan causes the car order to be cancelled.

3.4 Concurrency

Parallel execution in BPEL (**Fork**, **Join**) is very tricky to render in LOTOS, despite the fact that LOTOS can readily specify concurrency. This is largely because BPEL has global variables that are shared among parallel execution paths, whereas LOTOS has only local state. It is also necessary to deal with the effects of event handlers during parallel execution, e.g. a fault may prematurely terminate one path and trigger compensation. By default, BPEL allows execution to continue if only one of the preceding parallel paths terminates successfully. However, an arbitrary combination of path termination statuses may be used to determine this.

The CRESS translation to LOTOS handles concurrency by collecting an exit state from each path. The status of each is then evaluated. If the **Join** condition is satisfied, execution can continue. If the condition is not satisfied, a *JoinFailure* fault is caused. However if the **Fork** specifies *loose* concurrency, the activity following **Join** is simply considered to have failed. This may allow other parts of the web service to continue.

Concurrency is a second reason for processes to carry their state as a parameter. Each parallel path exits with the current process state. The states from each path are reconciled, and the current process parameters are computed. In fact, BPEL acknowledges but does not solve the problem that the same variables may be altered in parallel path. The CRESS toolset performs a data flow analysis of diagrams as they are translated. This is essential anyway, for example to decide whether variables should be read

('?') or written ('!') in LOTOS events. The same data flow analysis detects variables that are altered on parallel paths, causing a warning to be issued during translation.

The following shows the translation of node 5 in figure 2 where the parallel paths from nodes 3 and 4 converge. As will be seen, the translation has to be very complex.

```
  (
    (
      SUPPLIER_3 [supplier,dealer1,dealer2]                    (* SUPPLIER output 3 *)
        (need,offer,offer2,states)
    ≫ Accept states:States In                                   (* accept fork states *)
      Exit(states,Any States)                                          (* fork exit *)
    )
  |||
    (
      SUPPLIER_4 [supplier,dealer1,dealer2]                    (* SUPPLIER output 4 *)
        (need,offer,offer2,states)
    ≫ Accept states:States In                                   (* accept fork states *)
      Exit(Any States,states)                                          (* fork exit *)
    )
  )
≫ Accept states0,states1:States In                              (* accept join states *)
  (
    Let state:State = State(AnyBool,need,offer,offer2) In        (* get state updates *)
    Let state0:State = Head(states0) In                        (* get SUPPLIER 3 state *)
    Let state1:State = Head(states1) In                        (* get SUPPLIER 4 state *)
    Let status0:Bool = getStatus(state0) In                   (* get SUPPLIER 3 status *)
    Let status1:Bool = getStatus(state1) In                   (* get SUPPLIER 4 status *)
    Let state:State = getState(state,state0,state1) In            (* reconcile states *)
    Let need:Need = getNeed(state) In                  (* set need from combined state *)
    Let offer:Offer = getOffer(state) In              (* set offer from combined state *)
    Let offer2:Offer = getOffer2(state) In           (* set offer2 from combined state *)
    Let states:States = getStates(Tail(states0),Tail(states1)) In  (* combine states *)
      [Not(status0 And status1)]->                                  (* join failed? *)
        SUPPLIER_EVENT [supplier,dealer1,dealer2]              (* call event dispatcher *)
         (need,offer,offer2,states,AnyNat,JoinFailure,AnyValue)
    []
      [status0 And status1]->                                  (* check join condition *)
        SUPPLIER_5 [supplier,dealer1,dealer2]                (* SUPPLIER from join 5 *)
         (need,offer,offer2,states)
  )
```

3.5 Partner Processes

Partner web services are translated as separate LOTOS processes, synchronised in parallel with the main LOTOS process. If the partner is an external web service (e.g. *approver* or *assessor* in figure 1), a skeleton specification is generated to match its port/operation signature. For example, the default specification of *approver* is:

```
Process APPROVER [approver] : Exit(States) :=                    (* APPROVER partner *)
  approver !loan !approve ?proposal:Proposal;            (* APPROVER 'approve' input *)
  (
    approver !loan !approve !AnyNumber;                (* APPROVER 'approve' output *)
    APPROVER [approver]                                        (* repeat APPROVER *)
  []
    approver !loan !approve !refusal !AnyText;           (* APPROVER 'refusal' fault *)
    APPROVER [approver]                                        (* repeat APPROVER *)
  )
EndProc                                                             (* end APPROVER *)
```

This is sufficient for basic validation of the *lender* web service, but does not permit useful analysis. It is therefore possible to give a more realistic specification of external partners. If the CRESS translator finds the file <*partner*>*.lot*, it uses this specification of the partner instead of the default one. In fact these specifications can be arbitrarily complex. The four external partners in figures 1 and 2 were given realistic specifications. For example, the *dealer* partners maintain 'databases' (lists) of car information, customer quotations and customer orders.

3.6 Overall Specification Structure

When the broker service in figure 3 is translated, the services in figures 1 and 2 are also incorporated. The result is 330 lines of automatically generated LOTOS data types and 310 lines defining LOTOS processes. To this must be added the 400 lines of manually specified partner processes. The generated code is embedded in a specification framework that provides generic support for any web service. This consists of 590 lines of LOTOS (mostly complex data types). In total, this amounts to just over 1600 lines of LOTOS – a manageable specification.

The translation of exactly the same services to BPEL makes an interesting comparison. For this, CRESS generates 60 source files and 3300 lines of code (mostly BPEL, WSDL and Java). So whether the translation to LOTOS or BPEL is considered, it is evident that the CRESS notation is very compact.

4 Rigorous Analysis of Web Services

4.1 The Value of Formalising Web Services

Developing a formal interpretation of BPEL has been valuable in its own right. For example, a number of errors, omissions and ambiguities have been found in the standard (mainly in complex areas such as event handling and data handling). A number of these errors in BPEL4WS have already been corrected in WS-BPEL. The formalisation of BPEL also provides a precise interpretation of the standard.

More importantly, the formalisation supports a wide variety of analyses. Some of the investigations have used the TOPO (and LOLA) tools for LOTOS, while others have used CADP. Both offer distinct capabilities. LOLA has the advantage of using LOTOS data types as specified; this is beneficial since web services are supported by some rather complex types. LOLA is particularly useful for performing formally-based validation. CADP complements this through capabilities such as state space minimisation, equivalence checking and model checking. The penalty in using CADP is that it places certain requirements on the LOTOS, mainly on the data types. Some of these issues are addressed by annotations, but actualised data types have to be expanded manually, and some data types need manual realisations.

Rigorous analysis aims to find problems with a web service viewed as a black box. Formal verification indicates where the LOTOS is incorrect; the automatically generated comments show where the CRESS description needs to be improved. Formal validation, however, is performed at a higher level, so the CRESS changes are more obvious.

4.2 Formal Checking

When web services are composed, there is a danger that they do not synchronise properly due to a misunderstanding over the interface. In LOTOS terms, this manifests itself as a deadlock. (A LOTOS web service either performs **Exit** or recurses.) This is easily checked by LOLA using its expansion capabilities. When using BPEL (or more exactly WSDL), it is difficult to manually check services for compatibility since WSDL interface descriptions can be written in different ways and yet be consistent.

The internal design of a web service is proprietary. The owner may, however, wish to publish an abstraction for public use. There is then a question of whether the private and public specifications are consistent with each other. Essentially the public specification must be equivalent (e.g. observationally) to the private specification. Web services also evolve, e.g. the external partners used by a business process may change. Again, there is an issue of whether an updated web service is equivalent to the former one. CADP supports these kind of analyses with the specifications generated from web services.

CADP also allows model checking of web service properties. Safety and liveness properties can be formulated in ACTL (Action-based Computational Temporal Logic). For example, the *lender* service must not fault (safety), and every invocation of the *broker* service must eventually receive a response (liveness).

4.3 Rigorous Validation

In practice, web services have to be manually debugged like any other program, though tools like ActiveBPEL provide visual simulation. The LOTOS generated for web services can, of course, be manually simulated – but again this is just debugging.

The author has developed MUSTARD (Multiple-Use Scenario Test and Refusal Description [10]) as a language-independent and tool-independent approach for expressing use case scenarios. These are translated into the chosen language (LOTOS here) and automatically validated against the specification (using LOLA). This is useful for initial validation of a specification, and also for later 'regression testing' following a change in the service description.

There is insufficient space here to explain the MUSTARD notation, so reference to [10] and to the following example must suffice. Briefly, MUSTARD allows scenarios with sequences, alternatives, non-determinism and concurrency. The following MUSTARD scenario checks simultaneous requests to the *supplier* process. The first sequence requests an Audi A5, and expects to receive a schedule with dealer reference 8, name WheelerDealer, price 33000, delivery 30 days, loan rate 3.5%. The second requests a Ford Mondeo, and allows a specified schedule or an unavailable message in return.

```
test(Simultaneous_Purchases,                                      % simultaneous purchases scenario
  succeeds(                                                         % behaviour must succeed
    interleaves(                                                    % behaviours are interleaved
      sequences(                                                    % need request, schedule response
        send(broker.carloan.purchase,Need(′Ken Turner,′Stirling Scotland,′Audi A5)),
        read(broker.carloan.purchase,Schedule(8,′WheelerDealer,33000,30,3.5))),
      sequences(                                                    % need request, choice response
        send(broker.carloan.purchase,Need(′Kurt Jenner,′London England,′Ford Mondeo)),
        offers(                                                     % 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 [9]) for this kind of purpose. Trying to generate useful tests from a complex specification is generally infeasible. PCL is therefore 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.

4.4 Interaction among Services

Scenario-based validation is also a useful way of checking for interference among supposedly independent services. In telecommunications, this is called the feature interaction problem. Interactions may arise for technical reasons (e.g. conflicting services are activated by the same trigger) or for resource reasons (e.g. the services have a shared resource or external partner). One way of interpreting service interaction is that a service behaves differently in the presence of some other service.

Web services are formally validated by a range of MUSTARD scenarios that address all the critical characteristics of their behaviour. It then becomes possible to check services in isolation as well as in combination. This can effectively and efficiently detect interactions among services, though failure to detect interactions is not a guarantee that the services are interaction-free.

Web services are usually viewed as atomic and therefore do not incorporate add-on features (unlike telecommunications services). However it is useful to have a feature concept for web services. CRESS readily supports this in the same way as features can be added to voice services. A range of generic features has therefore been defined for web services; space does not allow them to be presented in detail here.

Consider the sample web services discussed earlier. They all make use of a customer name and address. The services could also perform other operations such as setting up an account or checking the status of a request. In all cases, it would be useful to validate the name and address provided. In fact this is a fraught problem, as all maintainers of mailing lists are aware.

A *name* feature has therefore been defined for normalising names. This is automatically invoked when a web service receives a given request with a name. It sets the name into a normal form (e.g. ‘KJ Turner’). A *contact* feature has also been defined for checking whether a name and address are known to be associated. This is automatically invoked when a given request with name and address is received by a web service.

When services are validated with MUSTARD using *contact* alone or with *name* as well, it is found that they behave differently (i.e. feature interaction occurs). The problem is obvious: if the *name* feature normalises a name, this may be inconsistent with the name recorded for an address. Of course, most feature interactions are obvious with hindsight. The value of automated analysis is that such problems are detected without detailed manual investigation when a new feature is added.

5 Conclusions

Business processes can benefit from formal models of their behaviour. A graphical description is much more understandable than the raw BPEL and WSDL. A high degree of automation is strongly desirable in the creation of web-based business processes. CRESS meets all of these requirements. Compared to commercial tools, CRESS does not support the entirety of web services. It handles nearly everything used in practice, a lack of timers being the main omission. However CRESS confers distinctive benefits: applicability to many domains, human-readable code for translated services, features as service add-ons, and translation to formal languages for rigorous analysis.

CRESS has now shown its worth in four rather different application domains: IN, Internet Telephony, IVR and web services. The toolset is portable, having been used on four different platforms. CRESS accepts diagrams drawn with three existing graphical editors, and generates code in five different languages. It is therefore an approach of wide practical and theoretical benefit.

References

1. T. Andrews *et al.*, editors. *Business Process Execution Language for Web Services*. Version 1.1. BEA, IBM, Microsoft, SAP, Siebel, May 2003.
2. A. Arkin *et al.*, editors. *Web Services Business Process Execution Language*. Version 2.0. OASIS, Billerica, Massachusetts, Feb. 2005.
3. BPMI. *Business Process Modeling Notation*. Version 1.0. Business Process Management Initiative, May 2004.
4. A. Ferrara. Web services: A process algebra approach. In *Proc. 2nd. Intl. Conf. on Service-Oriented Computing*, 242–251. ACM Press, New York, Nov. 2004.
5. H. Foster, S. Uchitel, J. Kramer, and J. Magee. Compatibility verification for web service choreography. In *2nd. Intl. Conf. on Web Services*, San Diego, California, July 2004.
6. K. J. Turner. Formalising the CHISEL feature notation. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions*, 241–256. IOS Press, Amsterdam, May 2000.
7. K. J. Turner. Modelling SIP services using CRESS. In D. A. Peled and M. Y. Vardi, editors, *Proc. FORTE XV*, LNCS 2529, 162–177. Springer, Berlin, Nov. 2002.
8. K. J. Turner. Analysing interactive voice services. *Computer Networks*, 45(5):665–685, Aug. 2004.
9. K. J. Turner. Test generation for radiotherapy accelerators. *Software Tools for Technology Transfer*, Oct. 2004, in press.
10. K. J. Turner. Validating feature-based specifications. *Software Practice and Experience*, May 2005, in press.