Kenneth J. Turner and Koon Leai Larry Tan.
Graphical Composition of Grid Services. In Didier Buchs and Nicolas Guelfi, editors,
Proc. International Conference on Rapid Introduction of Software Engineering Techniques,
Lecture Notes in Computer Science 4401, pages 1-17, Springer, Berlin, May 2007.

Graphical Composition of Grid Services

Kenneth J. Turner and Koon Leai Larry Tan

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

Abstract. Grid services and web services have similarities but also significant differences. Although conceived for web services, it is seen how BPEL (Business Process Execution Logic) can be used to orchestrate a collection of grid services. It is explained how CRESS (Chisel Representation Employing Systematic Specification) has been extended to describe grid service composition. The CRESS descriptions are automatically converted into BPEL/WSDL code for practical realisation of the composed services. This achieves orchestration of grid services deployed using the widely used Globus Toolkit and ActiveBPEL interpreter. The same CRESS descriptions are automatically translated into LOTOS, allowing systematic checks for interoperability and logical errors prior to implementation.

1 Introduction

1.1 Motivation

This paper presents a unique blend of ideas from different technical areas: distributed computing, software engineering, service-oriented architecture, and formal methods. Grid computing has emerged as a leading form of distributed computing. However, grid computing has largely focused on the development of isolated applications. Service-oriented architecture provides a framework for combining grid services into new ones.

The emphasis of this paper is on integrating software engineering techniques (visual programming, formal methods) into an evolving application area of considerable importance (grid computing). The aim has been to achieve immediate and practical benefits from advanced software techniques. Grid computing is a comparatively new field that has so far focused mainly on pragmatic, programmatic aspects. The work presented here offers a number of advantages:

- As with component-based approaches, grid services are combined into new composite services using BPEL as an emerging standard for web services.
- Grid service composition is described graphically, making it comprehensible to less technical users. Compared to the automatically generated code, the approach is compact and much more attractive than writing the raw XML that underlies it.
- A sound technique has been defined, benefiting from formal methods behind the scenes yet supporting automated implementation.

The approach is therefore application-driven (orchestrating grid services), novel (combining practice and theory), practical (automated implementation and validation), and integrated (complementing existing grid practice).

1.2 Background to Grid Computing

Grid computing is named by analogy with the electrical power grid. Just as power stations are linked into a universal electrical supply, so computational resources can be linked into a computing grid. Distributed computing is hardly a new area. But the architecture and software technologies behind the grid have captured the attention of those who perform large-scale computing, e.g. in the sciences. Grid computing offers a number of distinctive advantages that include:

- support for virtual organisations that transcend conventional boundaries, and may come together only for a particular task
- portals that provide ready access to grid-enabled resources
- single sign-on, whereby an authenticated user can make use of distributed resources such as data repositories or computational servers
- security, including flexible mechanisms for delegating credentials to third parties to act on behalf of the user
- distributed and parallel computing.

Grid computing is governed by OGSA (Open Grid Services Architecture [8]). Open standards for the grid are being created by the GGF (Global Grid Forum). Grid applications often make themselves available via services that are comparable to web services – another area of vigorous development. For a time, grid services and web services did not share compatible standards. The major issue was the need for stateful services that have persistent state. A grid-specific solution to this was developed. However, this was clearly something that web services could also benefit from.

A harmonised solution was defined in the form of WSRF (Web Services Resource Framework [10]). This is a collection of interrelated standards such as WS-Resource and WS-ResourceProperties. WSRF is implemented by various toolsets for grid computing such as GT4 (Globus Toolkit version 4, *www.globus.org*).

1.3 Background to Service Orchestration

This paper emphasises the *composition* of grid services, not the description of *isolated* grid services. Composing services has attracted considerable industrial interest. This is achieved by defining a *business process* that captures the logic of how the individual services are combined. The term *orchestration* is also used for this. A nice feature of the approach is that a composed service acts as a service in its own right.

Competing solutions were originally developed for orchestrating web services. A major advance was the multi-company specification for BPEL4WS (Business Process Execution Language for Web Services [1]), which is being standardised as WS-BPEL (Web Services Business Process Execution Language [2]). BPEL is now relatively well established as the way of composing web services. However, its use for composing grid services has received only limited attention. The work reported in this paper has used ActiveBPEL (an open-source BPEL interpreter, *www.activebpel.org*).

1.4 Background to CRESS

CRESS (Communication Representation Employing Structured Specification) was developed as a general-purpose graphical notation for services. Essentially, CRESS describes the flow of actions in a service. It thus lends itself to describing flows that combine grid services.

CRESS has been used to specify and analyse voice services from the Intelligent Network, Internet Telephony, and Interactive Voice Response. It has also been used to orchestrate web services [19]. In the new work reported here, CRESS has been extended to the composition of grid services. The present paper discusses how the same approach can be used for practical but formally-assisted development of grid services. Formally-based investigation of composite grid services will be reported in a future paper.

The work reported in this paper has been undertaken in the context of the GEODE project (Grid Enabled Occupational Data Environment, *www.geode.stir.ac.uk*). This project is researching the use of grid computing in social science, specifically grid services for occupational data analysis. The authors have investigated how services from this domain can be composed, formalised and rigorously analysed.

Service descriptions in CRESS are graphical and accessible to non-specialists. A major gain is that descriptions are automatically translated into implementation languages for deployment, and also into formal languages for analysis. CRESS offers benefits of comprehensibility, portability, automated implementation and rigorous analysis.

CRESS is extensible, with plug-in modules for application domains and target languages. Although web service support had already been developed for CRESS, it has been necessary to extend this significantly for use with grid services. In addition, grid services have specialised characteristics that require corresponding support in CRESS.

CRESS is intended as part of a formally-based method for developing services. In the context of grid computing, the steps are as follows:

- The desired composition of grid services is first described using CRESS. This gives a high-level overview of the service interrelationships. Because the description is graphical, it is relatively accessible even to non-specialists.
- The CRESS descriptions are then automatically translated into a formal language. CRESS supports standardised formal languages such as LOTOS (Language Of Temporal Ordering Specification [11]) and SDL (Specification and Description Language [12]), though this paper uses only LOTOS. Obtaining a formal specification of a composite service is useful in its own right: it gives precise meaning to the services and their combination.
- Although CRESS creates an outline formal specification for each of the partner services being combined, it defines just their basic functionality. This is sufficient to check basic properties such as interoperability. However for a fuller check of composite functionality, a more realistic specification is required of each partner. This allows a rigorous analysis to be performed prior to implementation.
- A competent designer can be expected to produce a satisfactory service implementation. However, combining services often leads to unexpected problems. The services may not have been designed to work together, and may not interoperate properly. The issues may range from the coarse (e.g. a disagreement over the interface) to the subtle (e.g. interference due to resource competition). This is akin to the

feature interaction problem in telephony, whereby independently designed features may conflict with each other. CRESS supports the rigorous evaluation of composite services. Problems may need to be corrected in either the CRESS descriptions or in the partner specifications. Several iterations may be required before the designer is satisfied that the composite grid service meets its requirements.
- The CRESS descriptions are then automatically translated into an implementation language. The interface to each service is defined by the generated WSDL (Web Services Description Language [22]). The orchestration of the services is defined by the generated BPEL. The partner implementations must be created independently, hopefully using the formal specifications already written. However, CRESS can generate outline code that is then completed by the implementer. This avoids simple causes of errors such as failing to respect the service interface.

1.5 Relationship to Other Work

As noted already, orchestration of web services has been well received in industry. Scientific workflow modelling has been studied by a number of projects. The MyGrid project has given an overview of these (*http://phoebus.cs.man.ac.uk/twiki/bin/view/Mygrid*). Only some of the better known workflow languages are mentioned below.

JOpera [16] was conceived mainly for orchestrating web services, though its applicability for grid services has also been investigated. JOpera claims greater flexibility and convenience than BPEL. Taverna [15] was also developed for web services, particularly for coordinating workflows in bioinformatics research. The underlying language SCUFL (Simple Conceptual Unified Flow Language) is intended to be multi-purpose, including applications in grid computing.

CRESS is designed for modelling composite services, but was not conceived as a workflow language. CRESS serves this role only when orchestrating grid or web services; its use in other domains is rather different. An important point is that CRESS focuses on generating code in standard languages. For service orchestration, this means BPEL/WSDL. This allows CRESS to exploit industrially relevant developments.

Several researchers have used BPEL to compose grid services. [5] describes a graphical plug-in for Eclipse that allows BPEL service compositions to be generated automatically. This work is notable for dealing with large-scale scientific applications. [3] discusses programmatic ways in which BPEL can support grid computing. [18] examines how extensibility mechanisms in BPEL can be used to orchestrate grid services. However, the focus of such work is pragmatic. For example, grid services may be given a web service wrapping for compatibility. (Semi-)automated methods of composing grid services have been investigated, e.g. work on adapting ideas from the semantic web [14].

An important advantage of CRESS is that practical development is combined with a formal underpinning. Specifically, the same CRESS descriptions are used to derive implementations as well as formal specifications. The formalisation permits rigorous analysis through verification and validation. A number of approaches have been developed by others for formalising *web* services. However, the authors are unaware of any published work on formal methods for composing *grid* services.

As an example of finite state methods for web services, LTSA-WS (Labelled Transition System Analyzer for Web Services [7]) allows composed web services to be described in a BPEL-like manner. Service compositions and workflow descriptions are automatically checked for safety and liveness properties. WSAT (Web Service Analysis Tool [9]) models the interactions of composite web services in terms of the global sequences of message they exchange. For verification, these models are translated into Promela and verified with SPIN. The ORC (Orchestration) language has also been used to model the orchestration of web services. [17] discusses its translation into coloured Petri nets. Both this and the alternative translation into Promela support formal analysis of composed web services. CRESS, however, is a multi-purpose approach that works with many kinds of services and with many target languages.

As an example of process algebraic methods for web services, automated translation between BPEL and LOTOS has been developed [4, 6]. This has been used to specify, analyse and implement a stock management system and a negotiation service. 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 service development often involves non-computer scientists as well as technical experts.

The CRESS notation has been previously been described in other papers. More recently, [19] has shown how web services can be modelled by CRESS. Since grid services are similar, but certainly not the same, this paper focuses on the advances that have been necessary to model and analyse the composition of grid services.

2 Describing Composite Grid Services with CRESS

CRESS is a general-purpose notation for describing services. Figure 1 shows the subset of constructs needed in this paper for grid services; CRESS supports more than this.

2.1 CRESS Notation for Grid Services

External services are considered to be *partners*. They offer their services at *ports* where *operations* may be performed. Invoking a service may give rise to a *fault*.

A CRESS diagram shows the flow among activities, drawn as ellipses. Look ahead to figures 2 and 3 for examples of CRESS diagrams. Each activity has a number, an action and some parameters. Arcs between ellipses shown the flow of behaviour. Note that CRESS defines flows and not a state machine; state is implicit.

Normally a branch means an alternative, but following a **Fork** activity it means a parallel path. An arc may be labelled with a value guard or an event guard to control whether it is traversed. If a value guard holds, behaviour may follow that path. An event guard defines a possible path that is enabled only once the corresponding event occurs.

In CRESS, operation names have the form *partner.port.operation*. Fault names have the form *fault.variable*, the fault name or variable being optional.

CRESS	Meaning
/ *variable* <– *value*	assignment associated with a node or an arc
Catch *fault*	A handler for the specified fault. A fault with name and value requires a matching **Catch** name and variable type. A fault with only a value requires a matching **Catch** variable type. A fault is considered by the current scope and progressively higher-level scopes until a matching handler is found.
Compensate *scope*?	Called after a fault to undo work. Giving no scope means compensation handlers execute in reverse order of being enabled.
Compensation	A handler that defines how to undo work after a fault. A compensation handler is enabled only once the corresponding activity completes successfully. When executed, it expects to see the same process state as when it was enabled.
Fork *strictness*?	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 (**strict**, the default). Matched by **Join**.
Join *condition*?	Ends parallel paths. An explicit join condition may be defined over the termination status of parallel activities. This gives the node numbers of immediately prior activities, e.g. '1 && 2' means these (and the prior ones) must succeed.
Invoke *operation output* (*input faults**)?	An asynchronous (one-way) invocation for output only, or a synchronous (two-way) invocation for output-input with a partner service. Potential faults are declared statically, though their occurrence is dynamic.
Receive *operation input*	Typically used at the start to receive a request for service. An initial **Receive** creates a new process instance. Usually matched by a **Reply** for the same operation.
Reply *operation output* | *fault*	Typically used at the end to provide an output response. Alternatively, a fault may be thrown.
Terminate	Ends a process abruptly.

Fig. 1. CRESS Notation (using BNF)

A CRESS rule-box, drawn as a rounded rectangle, defines variables and subsidiary diagrams (among other things). Simple variables have types like **Float** *f* or **String** *s*. CRESS also supports grid computing types such as **Certificate** (a digital security certificate), **Name** (a qualified name) and **Reference** (an endpoint reference that characterises a service instance and its associated resources).

Structured types are defined using '[...]' for arrays and '{...}' for records. For example, the following defines the variable *scores*. This is a record with fields: float *length* and string array *frequency*. A typical value would be the string *scores.frequency[2]*.

{ **Float** length [**String** word] frequency } scores

2.2 Content Analysis using Grid Services

The examples in this paper are drawn from the field of document content analysis (e.g. [13]). This is used for many purposes such as investigating disputed authorship of a

document, analysing different versions of a document to identify likely antecedents, or comparing two documents for plagiarism. This is a rich field, so only a simplified version is described in order to illustrate how orchestrated grid services can be used.

In the example of this paper, documents are compared for similarity using the following two metrics that lie in the range [0, 1]. For both of these, identical documents have a 'distance' of 0. Documents with a 'distance' of 1 are maximally different.

Clause Length: The average number of words per clause is computed for each document. Suppose the numbers are 6 and 8. The 'distance' between the documents is the difference between these divided by the larger value: $\frac{8-6}{8}$, i.e. 0.25.

Word Frequency: the instances of each word are counted (disregarding common words) and the words are placed in order of decreasing frequency. This gives an ordered list of words for each document (truncated to some practical limit such as 50 words). The 'distance' between the two word lists is then computed from the relative positions of each word in the two lists (counting the first as 0). Suppose 'grid' is the second most frequent word in one list (i.e. position 1) but the fourth most frequent in the other (i.e. position 3). The distance for this word is the difference between their positions: $3 - 1$, i.e. 2. If a word does not appear in the other list, its position there is notionally the length of that list. Thus if 'grid' did not appear in the second list (of size 50), the distance would be $50 - 1$ or 49. This ensures that if a more frequent word is missing, it has a greater distance. The total distance between two word vectors is the sum of the distances for all the individual words, normalised to yield a value between 0 and 1.

The content analysis example makes use of two external partner grid services that could exist already or should be developed separately because they are generally useful:

Counter: This calculates various measures over a document. The *clause* operation computes the average clause length. The *word* operation determines the words in decreasing frequency. The *distance* operation computes the metrics explained above from the raw clause and word information.

Parser: This handles word lists for a document. The *parse* operation takes a document as a string of text and splits it up into words (consecutive letters and possibly digits), disregarding white space. Consecutive punctuation marks (e.g. ':-') are also grouped as 'words'. Like many grid services, the parser holds its results in persistent storage and just returns an endpoint reference for the word list. This reference can be used by other services to perform further analyses. The *delete* operation removes a stored word list.

2.3 CRESS Description of The Scorer Service

The scorer is an auxiliary service that supports the main content analysis application. Its CRESS description appears in figure 2. The rule-box to the bottom right of the figure defines types and variables. The raw data is *words* – a reference to the word list being analysed. The result is *scores* – the average clause length and word frequency list.

Initially the scorer receives a request to perform a *score* operation on the words list (node 1). Since calculating the two distance metrics may be time-consuming, each is

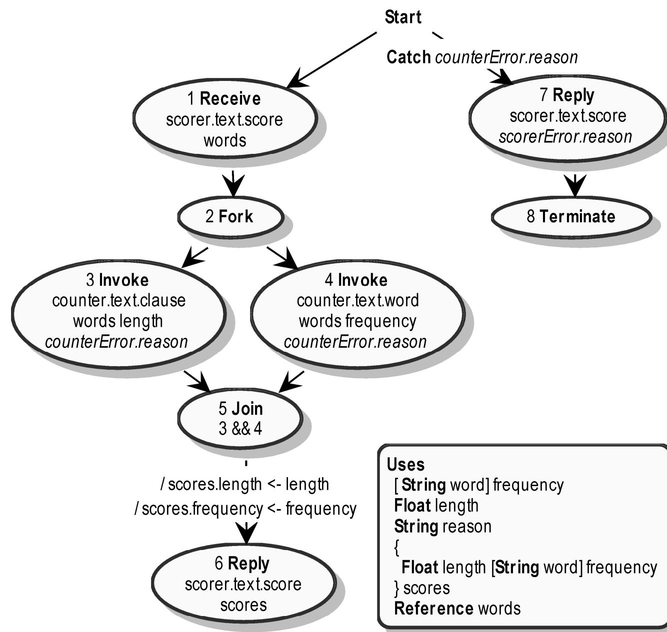

Fig. 2. CRESS Description of The Scorer Service

computed concurrently (node 2). In one parallel branch, the counter service is invoked to calculate the average clause length (node 3). In another parallel branch, a different instance of the counter service is invoked to determine words in decreasing order of frequency (node 4). Where both paths converge at node 5, they must have produced a successful result ('3 && 4'). The two metrics are combined into one record (arc leading to node 6). Finally, the scores are returned by the scorer to its caller (node 6).

The scorer must allow for the counter process faulting. For example, the word list may be empty or may contain only punctuation. Both invocations of the counter statically declare that a *counterError* may occur (node 3 and 4). If this happens, the fault is caught (arc leading to node 7). The scorer then returns the fault reason to its caller (node 7) and terminates (node 8).

2.4 CRESS Description of The Matcher Service

The matcher offers the primary content analysis service to the user. Its CRESS description appears in figure 3. The rule-box at the bottom right again defines types and variables. The raw data is *texts* – text strings containing the two documents. The analysis yields *metrics* – the clause length and word frequency distances. The final entry in the rule-box '*/ SCORER*' indicates that the matcher depends on the scorer service.

Initially the matcher receives a request to perform the *match* operation on the texts (node 1). Since the documents are independent and may be large, their metrics are computed separately on two parallel paths (node 2). Each starts by setting the relevant

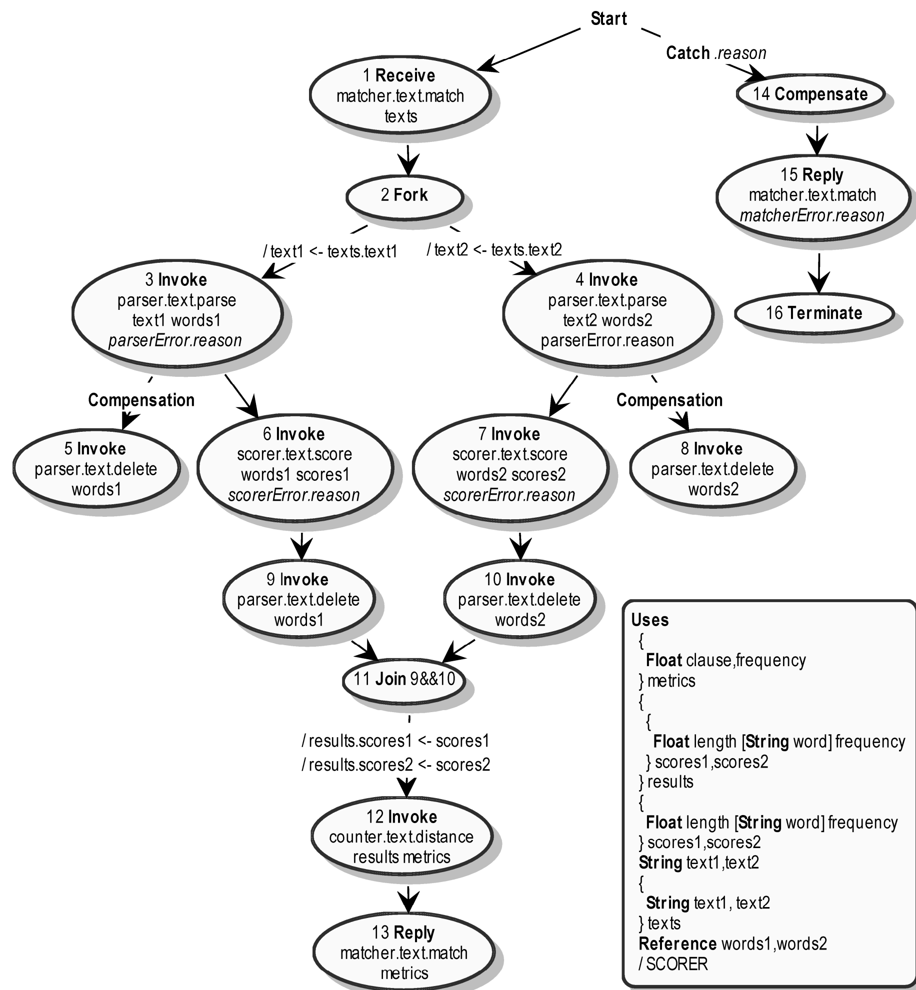

Fig. 3. CRESS Description of The Matcher service

text (*text1/text2* on the arc leading to node 3/4). The parser is invoked to create a word list from a document (node 3/4). The word lists are held by the parser, and returned as endpoint references (*words1/words2*). The scorer is then invoked to compute the metrics (*scores1/scores2* in node 6/7). The word lists have now served their purpose and are deleted (node 9/10). The converging paths must both be successful ('9 && 10' in node 11). The separately computed scores are combined (arc leading to node 12) and passed the counter to compute distances (node 12). The matcher returns the resulting metrics to its caller (node 13).

The matcher allows for faults in the services it calls: either of two invocations of the parser or the scorer may fail. Any such fault is caught (arc leading to node 14). The use of a fault variable (*reason*) without a fault name means that only a fault value is required: either *parserError* or *scorerError* is caught. Compensation is invoked by the

fault handler to undo any actions that have been taken (node 14). The matcher returns the fault to its caller (node 15) and terminates (node 16).

Compensation may be needed after invoking an external partner, since this is often where work needs to be undone after a fault. The parser invocations to store data (node 3/4) make permanent changes and so have associated compensation: the corresponding word list is deleted (node 5/8). A compensation handler is enabled once its associated activity completes. If compensation is invoked without an explicit scope (node 14), compensation handlers are invoked in reverse order (most recent first). If one parser invocation succeeds but the other fails, only the former will be compensated.

As has been seen, the matcher service orchestrates the actions of two external partner services (counter and parser) as well as the scorer service (figure 2). In turn, the scorer service orchestrates further operations of the counter partner. Although four services now have to cooperate, the user of the matcher service sees it as a whole. This is a major advantage, because the detailed design of the service is then hidden.

The major issue is whether the services work together smoothly, or whether there are interoperability problems. Even though this is a comparatively small example, it will be appreciated that there are many possibilities for error. It is very easy to make a mistake when calling a service, for example supplying an integer where a float is expected. Deadlocks are also a risk. Many more subtle problems can arise from semantic incompatibilities among the services. For these reasons, it is highly desirable to embed grid service development within a rigorous methodology.

2.5 The CRESS Service Configuration

Now that the various services have been introduced, the CRESS configuration diagram can be shown. Figure 4 shows how the services here are described. The **Deploys** clause lists the tool options and, following '/', the services to be deployed. Although only *MATCHER* is named, this implicitly includes all of the other services because of the inferred dependencies. The parameters of each service then follow in the configuration diagram. All services, such as *COUNTER*, have a namespace prefix ('cntr'), a namespace URI (Uniform Resource Name, 'CounterPoint'), and a base URI where they are deployed ('localhost:8880/wsrf'). As can be seen, in this case the services were deployed on the local computer. However, they can be deployed anywhere in the network.

Deploys *tool options* / MATCHER

COUNTER	cntr	urn:CounterPoint	localhost:8880/wsrf	-
MATCHER	mtch	urn:MatchMaker	localhost:8080/active-bpel	
PARSER	pars	urn:WordSmith	localhost:8880/wsrf	**String** textName
SCORER	scor	urn:UnderScore	localhost:8080/active-bpel	

Fig. 4. CRESS Description of The Service Configuration

Grid services (counter, parser here) may have resources, declared after the other parameters. The counter has no resources (shown as '-'). The parser has a resource: the

word list it stores, identified by *textName*. Every instance of the parser has a unique resource value, identified by its *resource key* in grid terminology. A composite service may also have resources. For example, if the matcher service were stateful then it too would have resource declarations.

2.6 Translation of The CRESS Diagrams

Translating the CRESS representation of *web* services has been described previously for BPEL [20]. However, the work reported in this paper has considerably extended and specialised this to handle *grid* services:

- A wider range of data types is now supported, including arrays and arbitrarily nested structured types. Specialised types have been added for dealing with grid services, such as certificates and endpoint references.
- Additional orchestration constructs have been added to match BPEL better.
- Support has been introduced for external partners shared amongst a number of services. Special treatment is needed to merge such descriptions in different diagrams.
- Grid service resources are now handled.

The CRESS diagrams (scorer, matcher, configuration) hold all that is needed to automatically generate a BPEL implementation and a LOTOS specification. Figure 5 compares translations of the content analysis example in figures 2 to 4:

- The fixed code is the framework common to all grid applications. This is substantial in the case of LOTOS because it contains many complex data types.
- The automatically generated code is shown for data types and behaviour. The BPEL translation yields many files: one BPEL file per service, one WSDL file per service/partner, and several deployment files. In addition, the WSDL files are automatically converted into Java. The LOTOS translation is a single file.
- The code for the external partners (counter, parser) has to be written manually. The Java coding conventions for grid services require several files per partner.

Target	Fixed Code	Generated Code			Partner Code		Total
		Files	Types	Behaviour	Files	Behaviour	
BPEL	20	51	14570	1640	10	2830	**19060**
LOTOS	840	1	530	400	2	290	**2060**

Fig. 5. Comparison of BPEL and LOTOS Translations *(lines of code except for Files columns)*

The BPEL implementation is substantially larger than the LOTOS specification, despite the fact that the LOTOS has a significant common overhead in data types. LOTOS has to explicitly specify functions on numbers, strings, etc. that would be expected in an implementation language. With larger examples, LOTOS is even more compact compared to BPEL. The most striking difference is in the large number of files required to support BPEL.

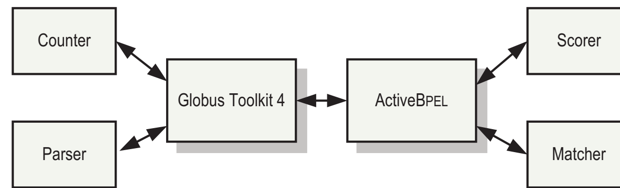

Fig. 6. Content Analysis Service Deployment

3 Translating Web Services to BPEL

Once the CRESS service diagrams have been created, their translation into BPEL/WSDL is automatic. The principles behind translating *web* services are outlined in [20]. Only a high-level description is given here, particularly covering where *grid* services differ.

3.1 Service Creation

Orchestrating grid services require a considerable amount of XML that is generated automatically by CRESS. Translation and deployment of a CRESS diagram is entirely automated, except for the one-off implementation of partner grid services. Partner services are automatically deployed using GT4 (Globus Toolkit version 4), while the orchestrating process is automatically deployed using ActiveBPEL.

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

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

3.2 Service Deployment

The deployment architecture is shown in figure 6. The grid services (counter, parser) are executed with GT4. Their orchestration (matcher, scorer) is handled by ActiveBPEL 2.0. Both GT4 and ActiveBPEL deploy services within a container that uses AXIS (the Apache SOAP engine). In principle, GT4 and ActiveBPEL could be executed within the same Apache Tomcat container. In practice, this is not feasible with the current versions. GT4 presently uses an older version of AXIS that is incompatible with ActiveBPEL; an updated version of GT4 is required before this can be resolved. For now, GT4 and ActiveBPEL are run in separate containers. Actually, this is reasonable since BPEL can coordinate grid services running on completely different computers. This would be quite likely in a realistic deployment of the content analysis example described in this paper.

GT4 currently imposes another limitation on the orchestration of grid services. The most desirable form of security is the so-called WS-SecureConversation that allows *credential delegation* in grid terminology. Unfortunately the current implementation

of GT4 requires all services to use the same container for delegation to work. The authors have developed a solution combining GT4 and ActiveBPEL, but the current AXIS incompatibilities mean this cannot be used yet. A newer version of GT4 will allow credential delegation to be realised.

Current limitations of ActiveBPEL mean resources have to be treated transparently. It is intended to make resources directly available to the orchestrating process. Endpoint references cannot be used directly by ActiveBPEL. It is planned to make BPEL processes behave more like grid services and less like web services.

3.3 Service Flow

BPEL 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 with **Join** to check whether a linked activity has terminated successfully.

A CRESS handler is translated into the corresponding type of BPEL handler. For example, **Catch** and **CatchAll** introduce a fault handler, while **Compensation** introduces a compensation handler. In principle, handlers may be defined in any scope including the global one. In fact, WS-BPEL does not allow global compensation handlers. CRESS regularises this situation by allowing handlers at two levels. Global handlers are translated as part of the top-level flow. The other place where CRESS allows handlers is in association with **Invoke**. Although this is a restriction compared to BPEL, it is where a handler is mostly likely to be required anyway.

3.4 Supporting Orchestration

Data types in CRESS are either simple ones defined by XML schemas (e.g. float, string) or are arbitrarily nested structures of records and arrays. Built-in types are used for the former, while complex types are generated for the latter. CRESS automatically handles the rather different ways in which BPEL uses variables: as message variables (input, output) or as data variables (assignment, expression).

BPEL processes orchestrate external partner services. In fact these may be web services or grid services (more precisely, stateless or stateful). The WSDL for partners is automatically generated from the CRESS diagrams, along with service deployment descriptors. If a partner service already exists, it can be used directly. The CRESS view is likely to be a subset of the partner WSDL, since an orchestrating 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 GT4 tool *wsdl2java*. The skeleton partner service must then be implemented manually.

3.5 Compatibility of ActiveBPEL and GT4

Resource addressing is a key issue for grid services. State information is handled separately from the service itself. A WS-Resource pair (service plus state) is encoded in an endpoint reference, as defined by the WS-Addressing schema. GT4 handles this implicitly, meaning that the ports used by clients are bound to a service and resource. To use another resource with the same service, a separate endpoint reference is created with the relevant resource key.

However, ActiveBPEL is not able to handle such a resource implicitly. Endpoint references thus have to be passed explicitly as parameters to grid service partners, allowing them to infer resource pairs. This requires compatibility of the WS-Addressing used by GT4 and ActiveBPEL. Unfortunately, the endpoint references generated by GT4 do not currently conform to the usual schema. Instead a variant schema with a *ReferenceParameters* element is used, leading to incompatibility. By altering the schemas in use, it is possible share endpoint references consistently. However, work remains to allow ActiveBPEL to use resources directly.

Grid services supported by GT4 require a *document/literal* SOAP binding. This is one of the binding styles that complies with the WS-Interoperability standard. However, this binding does not convey the operation name. Instead, the structure of the SOAP message body must be used implicitly to identify the operation being invoked. This causes ambiguity when a service has several operations with the same input signature, forcing use of distinct message parts even though they are not logically necessary.

4 Translating Grid Services to LOTOS

CRESS also translates grid services in LOTOS. Only the rigorous analysis this permits is discussed here. LOTOS was originally standardised for specifying and analysing communications standards (Open Systems Interconnection). However, LOTOS is a general-purpose language that supports precise specification of both behaviour and data: it is a process algebra supplemented by algebraic data types.

A LOTOS specification is automatically generated from the *same* CRESS diagrams that are translated into BPEL/WSDL. A default specification is provided for external partner services, though this just respects their operation interfaces. For more detailed analysis, the partners are specified manually.

Because CRESS is graphical, it is more understandable and compact than the corresponding code. Although this paper is focused on practical development of composite grid services, the use of a formal method is an important first step in their design. Apart from giving a precise definition of what orchestration means, it allows rigorous analysis of services prior to implementation. The use of formal methods is thus integrated into more conventional development techniques.

In practice, grid services are manually debugged. The generated LOTOS can, of course, be manually simulated as well. However, an important benefit of the formalisation is that it supports a wide variety of automated analyses.

An important issue in orchestrating grid services is to ensure their interoperability. Problems arise from simple misinterpretation of interfaces or from more subtle semantic

incompatibility. Such problems often lead to deadlock in LOTOS terms, as determined by automated behaviour exploration or through model checking.

Service properties can also be model checked. Safety and liveness properties of grid services can be formulated in ACTL (Action-based Computational Temporal Logic). For example, the matcher service must not fault (safety), and an invocation of it must eventually receive a response (liveness). Unfortunately the complex data types and infinite data sorts make model checking somewhat impractical. For this reason, the authors favour the use of rigorous validation instead of verification.

MUSTARD (Multiple-Use Scenario Test and Refusal Description [21]) has been developed as a language-independent and tool-independent approach for expressing use case scenarios. These scenarios are automatically translated into the chosen language (here, LOTOS) for automatic validation against the specification. This is useful for initial validation of a specification, and also for later 'regression testing' following a change in the service description. Scenario-based validation is also good for checking interference among supposedly independent services – the so-called feature interaction problem. Interactions may arise for technical reasons (e.g. conflicting services activated by the same input) or for resource reasons (e.g. services sharing a resource or external partner).

A major advantage of MUSTARD is that the use of an underlying formal method is entirely hidden from the user. An automated procedure translates CRESS and MUSTARD into LOTOS, validates the scenarios, and reports the analysis in language-independent terms. In other words, the use of LOTOS (or any other formal language) is invisible. In fact, the tool user merely draws diagrams and clicks a button to check their integrity.

Grid services are formally validated by MUSTARD scenarios that check critical aspects of their behaviour. It is possible to check services in isolation as well as in combination. This can effectively and efficiently detect service interactions, though failure to find interactions does not mean the services are interaction-free. MUSTARD supports scenarios with sequences, alternatives, non-determinism, concurrency and service dependencies. In addition, both acceptance tests and refusal tests may be formulated.

5 Conclusions

It has been seen how CRESS has been adapted to support orchestration of grid services. This offers the advantage that new composite services can be constructed from existing ones. As a realistic example, document content analysis has been used to explain how grid services can be orchestrated.

CRESS descriptions of composite grid services are translated into BPEL/WSDL for implementation. The orchestration is performed by ActiveBPEL, while the partner grid services are executed by GT4. The same CRESS descriptions are also translated into LOTOS for rigorous validation and verification. The whole development process is highly automated. The use of advanced software engineering techniques (visual programming, formal methods) has thus been integrated into the current grid computing practice.

Content analysis has been used as an example of how orchestration can be useful in grid computing. This is a realistic problem, although the illustration is a small one. The authors have also researched the use of grid computing in social science, specifically

grid services for occupational data analysis. Services from this domain are much more complex, and yet can be formalised and analysed rigorously using CRESS.

It has hopefully been demonstrated that CRESS is valuable in orchestrating grid services, implementing and analysing them.

Acknowledgements

Larry Tan's work was supported by the UK Economic and Social Research Council under grant RES-149-25-1015. The authors are grateful for the collaboration with their GEODE colleagues, particularly Paul Lambert (University of Stirling) and Richard Sinnott (University of Glasgow).

References

1. 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.
2. 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, Dec. 2005.
3. K.-M. Chao, M. Younas, N. Griffiths, I. Awan, R. Anane, and C.-F. Tsai. Analysis of grid service composition with BPEL4WS. In Y. Shibata and J. Ma, editors, *Proc. 18th. Advanced Information Networking and Applications*, volume 1, pages 284–289. Institution of Electrical and Electronic Engineers Press, New York, USA, 2004.
4. 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.
5. W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid service orchestration using the business process execution language (BPEL). *Grid Computing*, 3(3-4):283–304, Sept. 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. Compatibility verification for web service choreography. In M. Aiello, editor, *Proc. 2nd. International Conference on Service-Oriented Computing*, New York, USA, Nov. 2004. ACM Press.
8. I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Supercomputer Applications*, 35(6), 2002.
9. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. 13th. International World Wide Web Conference*, pages 621–630. ACM Press, New York, USA, May 2004.
10. S. Graham, A. Marmakar, J. Mischinsky, I. Robinson, and I. Sedukhin, editors. *Web Services Resource*. Version 1.2. Organization for The Advancement of Structured Information Standards, Billerica, Massachusetts, USA, Apr. 2006.
11. 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.

12. ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 2000.
13. K. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. Sage, Thousand Oaks, California, USA, 2004.
14. S. Majithia, D. W. Walker, and W. A. Gray. Automated composition of semantic grid services. In *Proc. 3rd. UK e-Science All Hands Meeting*. University of Nottingham, UK, Aug. 2004.
15. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
16. C. Pautasso. JOpera: An agile environment for web service composition with visual unit testing and refactoring. In *Proc. IEEE Symposium on Visual Languages and Human Centric Computing*. Institution of Electrical and Electronic Engineers Press, New York, USA, Nov. 2005.
17. S. Rosario, A. Benveniste, S. Haar, and C. Jard. Net systems semantics of web services orchestrations modeled in ORC. Technical Report PI 1780, IRISA, Rennes, France, Jan. 2006.
18. A. Slomiski. On using BPEL extensibility to implement OGSI and WSRF grid workflows. In *Proc. Global Grid Forum 10*, Berlin, Germany, Mar. 2005. Humboldt University.
19. 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.
20. K. J. Turner. Representing and analysing web services. *Network and Computer Applications*, Mar. 2006. In press.
21. K. J. Turner. Validating feature-based specifications. *Software Practice and Experience*, 36(10):999–1027, Aug. 2006.
22. World Wide Web Consortium. *Web Services Description Language (WSDL)*. Version 1.1. World Wide Web Consortium, Geneva, Switzerland, Mar. 2001.