Kenneth J. Turner and Koon Leai Larry Tan. Rigorous Development of Composite Grid Services. Network and Computer Applications, 35(4):1304-1316, February 2012.

Rigorous Development of Composite Grid Services

Kenneth J. Turner, Koon Leai Larry Tan

Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK

Abstract

CRESS (Communication Representation Employing Systematic Specification) is introduced as notation, a methodology and a toolset for service development. The article focuses on rigorous development of composite grid services, with particular emphasis on the principles behind the methodology. A straightforward graphical notation is used to describe grid services. These are then automatically specified, analysed and implemented. Analysis includes formal verification of desirable service properties, formal validation of test scenarios, testing of implementation functionality, and evaluation of implementation performance. The case study that illustrates the approach is document content analysis to compare two pieces of text. This involves two composite services supported by two partner services. The usability of the service design notation is assessed, and a comparison is made of the approach with similar ones. These show that the CRESS approach to developing services is usable and more complete than other comparable approaches.

Keywords:
BPEL (Business Process Execution Logic), Grid Service, LOTOS (Language Of Temporal Ordering Specification), Service Composition, Service Orchestration, Validation, Verification, Workflow Modelling

1. Introduction

1.1. Objectives

The overall goal of this work is an integrated methodology for developing a wide variety of services. A case study is presented of developing composite grid services, but the approach has also been used to create telephony services, interactive voice services, device services and composite web services. The grid [11] has emerged as an important approach to distributed computing. Much like the electricity grid, a computing grid provides computational capacity on demand. Grid services can be seen as an extension of web services, with features such as dynamic service selection and distributed resource access.

Email addresses: kjt@cs.stir.ac.uk (Kenneth J. Turner), larrytkl@gmail.com (Koon Leai Larry Tan)

Communications services are of increasing importance to industry and are therefore becoming quality-critical. The authors believe that practical formal methods such as described in this paper will make a useful contribution to service quality. The methodology described in this article covers service description and specification, verification and validation, implementation, testing and performance evaluation. In industrial practice, service development tends to be pragmatic: services are designed, coded and tested using traditional (and manual) software engineering techniques. However, online services are becoming increasingly mission-critical in many applications. Services are often combined with others for B2B communication (Business-to-Business). The services can be complex, concurrent, and risk unexpected interference. There is a strong need for techniques that deliver dependable services exhibiting desired properties, coupled with automatic code generation to turn these into reality.

Formal approaches are not so common in industry, except in a few specialised application areas such as safety-critical systems. The authors have tried to automate the methodology as far as possible and to make it accessible to non-specialists. Although researchers have aimed at improving service development, they have generally tackled only certain aspects of design and in ways that require detailed technical knowledge.

This article describes CRESS (Communication Representation Employing Systematic Specification). CRESS offers a notation for services, a methodology for service development, and a comprehensive toolset. Currently CRESS handles services in seven different domains, and supports code generation for five different languages. The foundational work in [33] introduced a notation for telephony features. This was subsequently adapted in [34] to describe web services and in [31] to describe grid services. The service development methodology has recently been rounded out with capabilities for convenient formal verification and implementation evaluation. Relative to previous publications on CRESS, this article covers the complete methodology, grid applications, development principles, and practical formal verification and validation.

1.2. Structure of The Article

Section 2 gives the technical background to the work, placing CRESS in context. Related work is described on specifying, implementing and testing composite web and grid services.

Section 3 explains service development with CRESS. The methodology is illustrated with respect to grid service development, including the subset of CRESS used in this article. The techniques for formal verification and formal validation are discussed.

Section 4 describes a case study that makes use of grid services. A document matching service makes use of a scoring service to compare texts for similarity. The diagrammatic service descriptions are automatically specified, verified and validated. Once confidence has been built in the service design, it is automatically implemented and deployed. In a final step, the implementation is automatically evaluated for correct functional behaviour and adequate performance.

Section 5 evaluates usability of the CRESS notation and compares CRESS to similar approaches. Section 6 rounds off the article with a summary of the approach.

2. Background

2.1. Modelling Services with CRESS

A service is an abstraction of the functionality provided by an application. SOA (Service Oriented Architecture) has become popular as a means of creating systems from loosely coupled components. A service offers a black-box, interface-oriented view of application functionality. As well as dealing with individual services, SOA also supports composing services into new ones. This offers new business opportunities, but with the added complexity of having to integrate a number of services seamlessly.

Service composition is also called service orchestration or workflow definition. A survey of workflow languages and an assessment of standards in this area are given by [28]. WS-BPEL (Web Services Business Process Execution Logic [2]) is a widely used standard for achieving this. BPEL provides the logic that links calls of individual services. Service choreography, e.g. WSCDL (Web Services Choreography Description Language), is a complementary approach that describes how services interact with each other rather than how their combined execution is achieved.

A composite service is a 'business process' that exchanges messages with partner services. A business process is itself a service with respect to its users. Services have communication ports where operations are invoked. An unsuccessful operation gives rise to a fault that may need compensation to undo prior work.

BPMN (Business Process Modeling Notation [4]) is a graphical notation for describing business processes in general. It is relevant here because it can be mapped to BPEL and thus to web service composition. Compared to CRESS, BPMN supports a wider range of capabilities. However, this means that BPMN is a much more complex notation. It also lacks the capabilities of formal analysis conferred by CRESS. Furthermore, modelling composite services is only one of many applications of CRESS, whereas BPMN is specialised for business processes.

The original grid architecture was OGSI (Open Grid Service Infrastructure), with extensions to WSDL (Web Services Description Language). Grid services now make use of resources through WSRF (Web Services Resource Framework) and also employ GSI (Grid Security Infrastructure). Many researchers have investigated *web* services, but CRESS is one of few methodologies that support *grid* services.

CRESS respects the principles of SOA and service composition. The emphasis of this article is on formal aspects, here using LOTOS (Language Of Temporal Ordering Specification [17]). LOTOS is a standardised formal language that was originally designed for specifying networked and distributed systems, but has found application in many other areas. LOTOS is a process algebra combined with algebraic data types that supports concurrency, verification, validation and application-defined data types.

CRESS is appropriate for domains that can be modelled as a flow of activities; this encompasses a broad range of applications. CRESS is not particularly designed for real-time aspects, though there is some support for time and timers. This potential limitation mainly depends on the underlying formalism used. However, the LOTOS tools used with CRESS support timing and stochastic extensions. Where the implementation language supports time directly, CRESS can make use of this. CRESS is also not strongly oriented towards performance evaluation. Stochastic aspects (e.g. probabilistic behaviour) are not supported.

CRESS supports formal verification (i.e. proof) and formal validation (i.e. rigorous testing). Formal verification requires a finite state space to be practicable, often requiring data type values to be constrained in some way. Formal validation does not have this limitation.

CRESS differs from other approaches in a number of respects:

- Most approaches deal with the design of just one kind of service (e.g. voice, web). CRESS uses a common set of techniques and tools to support the design of many different kinds of services.
- Most approaches handle only part of service development (e.g. analysis or testing). CRESS provides a complete methodology that covers service description, specification, analysis, implementation, testing and performance.
- Other approaches typically need expert knowledge (e.g. formal methods, specialised software). Thanks to a high degree of automation, CRESS covers many aspects of development with minimal effort by the service designer.
- Comparable approaches often use non-standard techniques. CRESS emphasises the use of standards, with the advantages of wide acceptance, availability of reference and tutorial material, tool support, and attractiveness to industry.
- Although many approaches support the design of *web* services, CRESS supports the design of *grid* services as well.
- Formal approaches often abstract services to make them tractable, handling only simple types such as booleans and integers. CRESS supports a full range of data types and data structures that are likely to be required in realistic services.

2.2. Specifying Composite Services

The SENSORIA project (Software Engineering for Service-Oriented Overlay Computers [27]) has studied a number of aspects of service design, including larger case studies using web service orchestration. UML (Unified Modeling Notation) is used to describe service structure, evolution and activities. A number of process calculi were created for modelling web services [38]. These are coupled with techniques for functional or performance analysis. UML has also been been used in [18] to describe web services at a high level, e.g. using activity diagrams or state diagrams. These are translated into FSP (Finite State Processes) for model checking to find deadlocks or problems with synchronisation. Although the use of UML agrees with the authors' preference for standards, the above formalisms for analysis are not standard (unlike LOTOS). CRESS also tries to support a methodology, techniques and compact toolset that do not require specialised knowledge.

LTSA-WS (Labelled Transition System Analyser for Web Services [10]) is a finite state approach to specifying web services. Abstract service scenarios and actual service implementations are generated through behavioural models in the form of state transition systems. Verification and validation are performed by comparing the two systems. The approach is restricted in its handling of data types. CRESS differs in generating the formal model and the implementation from a single abstract description, and in allowing arbitrary data types.

PROPOLS (Property Specification Pattern Ontology Language for Service Composition [39]) is a pattern-based specification language for temporal business rules. A behavioural model combines rules using their respective finite state automata. The process model can then, in principle, be transformed into BPEL. The approach does not, however, deal with data types. CRESS differs in generating both the specification and the implementation from the same description, dealing fully with data.

WSAT (Web Services Analysis Tool [12]) is used to analyse and verify composite web services, particularly focusing on asynchronous communication. Specifications are translated into Promela and model checked using SPIN. WSAT is able to verify synchronisability and realisability. However, the tool does not support the full range of capabilities found in BPEL (e.g. error handling and compensation).

[7, 9] automate translation between BPEL and LOTOS. CRESS differs in that no specification is required of either BPEL or LOTOS. Instead a graphical notation, accessible to the non-specialist, supports abstract service descriptions that are translated into BPEL and LOTOS automatically.

jABC (Java Application Building Center [29]) allows services to be created with reusable building blocks. The approach supports automates specification and verification. Mapping to BPMN and BPEL is also possible, including support for web services.

StAC (Structured Activity Compensation [6]) is a process algebra that has been used to specify the orchestration of long-running transactions. This can be used with the B notation to allow specification of data types. Most of BPEL can be translated into StaC, but the emphasis is on reasoning about transactions rather than support for BPEL. [20] also focuses on verifying web transactions, but is even further from BPEL.

2.3. Implementing Composite Services

Pragmatic aspects of web service implementation are well supported through packages such as ActiveBPEL [1] and Oracle BPEL Process Manager. Grid services are implemented by Globus [14], OMII (Open Middleware Infrastructure Institute), etc. The OMII-BPEL project [37] support scientific workflows using an adaptation of ActiveBPEL. Distinctive features include support for security and for long-running processes.

JOpera [24] 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 [23] was developed for web services, particularly for workflows in bioinformatics. The underlying language SCUFL (Simple Conceptual Unified Flow Language) is intended to be multi-purpose, including applications in grid computing. However, the authors believe that conformance to the widely accepted BPEL standard is desirable for acceptance.

2.4. Testing Composite Services

[3] gives a comprehensive overview of techniques for testing web services. Some aspects of this derive from work on test case generation, such as in protocol conformance testing. Model-based testing uses some kind of formal model of the system to derive rigorous tests. Of particular relevance to this article are techniques based on LOTOS (e.g. [32]). Because others have worked on test case generation, CRESS can build

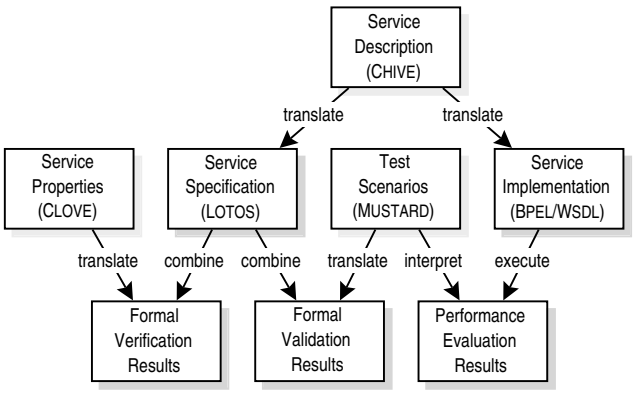

Figure 1: CRESS Methodology

on this. That is, CRESS focuses on convenient representation and execution of test cases. CRESS also offer a unique advantage: the same scenarios are used to validate a specification *and* its implementation.

[22] maps BPEL to PEPA (Performance Evaluation Process Algebra). Stochastic aspects are not defined by BPEL and have to be added manually. The approach extends WSDL with estimated latency attributes for each operation. The PEPA workbench then calculates the throughout of the model. Fault handling is not addressed by the approach, and factors such as communication cost and load are not considered.

3. CRESS Approach

3.1. Service Development with CRESS

Figure 1 shows the overall methodology for CRESS (Communication Representation Employing Systematic Specification [36]). All the major elements in the methodology are due to the authors, though use is made of third-party tools for BPEL and LOTOS. It is not feasible in this article to give a full description of the methodology and its applications. Interested readers can find a more detailed discussion in [30].

Services are described manually using the CRESS graphical notation. Several graphical editors are supported, but the preferred one is CHIVE (CRESS Home-Grown Interactive Visual Editor, *www.cs.stir.ac.uk/~kjt/software/graph/chive.html*). Diagrams are automatically translated into a formal specification that describes one or more services. The core CRESS notation is independent of the application domain and the verification language. In this article, grid services are translated to LOTOS for formal analysis. The meaning of a CRESS diagram is given denotationally through its representation in LOTOS (which has a formal semantics).

Business processes are automatically specified in full from their CRESS diagrams, but only outline specifications can be generated of partner services. This is sufficient to prove useful properties of a service design, such as correct use of interfaces. However, for fuller analysis it is desirable for the developer to provide complete specifications of partner services. These are automatically imported by CRESS and combined with the process specifications.

Properties a specification should respect are defined manually in CLOVE. Certain properties such as deadlock freedom, livelock freedom and guaranteed termination are automatically handled and do not need explicit definition. Application-specific properties have to be specified by hand, though it will be seen that CLOVE simplifies the process for non-technical users. Additional work is required prior to verification with the CADP toolset (Construction and Analysis of Distributed Programs [13, 16]). For example, CADP does not handle parameterised ('formal') types in LOTOS and needs annotations to specify how data types should be realised. The LOTOS generated by CRESS is automatically annotated for CADP, allowing automated verification of CLOVE properties.

CLOVE is independent of how exactly properties are checked. In this article, grid service properties are automatically translated into μ-calculus [5] and model checked. The meaning of a CLOVE property is given denotationally by its representation in μ-calculus (which has a formal semantics). Results from verification are presented in a way that is meaningful to the non-technical user. Although the procedure makes use of techniques such as on-the-fly and compositional verification, state space explosion often limits what is practical (a common issue with all model-checking techniques).

For this reason, formal validation is also supported as it copes with large (even infinite) state spaces. As a form of testing, validation is necessarily incomplete – but it complements what is possible through verification. The LOTOS specifications generated by CRESS can be used immediately for formal validation of test scenarios with the LOLA tool (LOTOS Laboratory [25]).

Test scenarios are created manually using MUSTARD (Multiple-Use Scenario Test and Refusal Description [35], *www.cs.stir.ac.uk/~kjt/research/mustard.html*).In this article, test scenarios for grid services are automatically translated into LOTOS and formally validated. The meaning of a MUSTARD test is given by its denotation in LOTOS.

The result of verification and validation is a service description in which the developer can have a high degree of confidence. The penultimate step is automatic generation and deployment of operational code. For grid services, this involves creating BPEL, WSDL and deployment descriptors. It would seem that the rigorous methodology ought to deliver dependable implementations. However, various issues can arise in deployment. For example, performance limitations may require implementation tuning, and the implementation may not operate as expected due to resource conflicts. The methodology therefore has a final step to evaluate performance of the actual implementation. This re-uses the test scenarios that were formally validated against the specification, giving a confidence check on functionality and performance under load.

3.2. Notation for Grid Services

A CRESS diagram is a directed graph that shows the flow of activities. Numbered nodes in a CRESS diagram define inputs and outputs (communication with other services) or actions (internal to the service). In an orchestrated service, an activity can terminate successfully or can fail (due to a fault). Branches in a CRESS diagram normally reflect alternatives, but parallel paths can also be defined. Although BPEL has separate constructs for sequence, iteration and graph-like flows, CRESS models these in a uniform way.

Construct	Meaning
(*diagram*:)?*name*	a variable or fault name defined by a particular diagram (the current diagram if no prefix is given)
service.port.operation	an operation for the given service and port
name(*.variable*)? | *.variable*	a fault name with optional variable or just a fault variable
/ variable <– value	an assignment associated with an arc or node
Catch *fault*	how to handle a fault; a fault unmatched in the current scope is sought in higher-level scopes
Compensate	used after a fault to undo previous work by calling compensation handlers in reverse order of activities
Compensation	undoes work due to a fault; enabled once the corresponding activity completes successfully
Fork	introduces parallel paths; may be nested to any depth
Invoke *operation output* (*input faults**)?	one-way for output, or two-way for output and input; potential faults are declared statically (but happen dynamically)
Join *condition*	matches a **Fork**; an explicit join condition refers to termination of prior activities, e.g. '1 && (2 || 3)'
Receive *operation input*	an initial **Receive** creates a new process instance, being matched by a **Reply** for the same operation
Reply *operation* (*output* | *fault*)	typically provides an output response at the end of a business process, though a fault may also be signalled
Terminate	ends a process abruptly

Table 1: Summary of CRESS Constructs used in Article ('?' optional, '*' zero or more, '|' choice)

The subset of CRESS activities appearing in this article is explained in table 1, with concrete examples appearing in sections 4.4 and 4.5. (CRESS supports a wider range of constructs than is described here.)

In a CRESS service diagram, arcs (service flow) join nodes (activities) shown as ellipses. CRESS arcs and nodes may have associated assignments. Arcs may be labelled with expression guards (alternative choices) or event guards (conditional on events occurring).

As an example, figure 2 shows part of an online loan service. Node 1 performs a **Receive** for service *lender*, port *loan*, operation *quote*, and input *request*. If the requested amount is 10000 or more, the request is copied into variable *proposal* (arc from node 1 to 2). The *approver* service is then asked to approve the loan request, resulting in the loan *rate* or a *refusal* with some *error* value (node 2). Otherwise, the *assessor* service is asked to perform a full assessment of the loan request, resulting in a *risk* assessment (node 3).

A CRESS rule box is shown as a rounded rectangle. It defines (among other things)

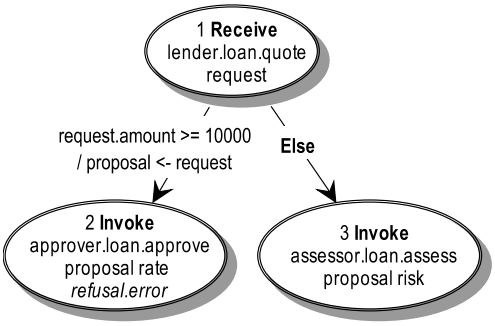

Figure 2: CRESS Notation Example

simple variables, structured variables and their types, and subsidiary diagrams. The format is **Uses** *declarations / diagrams*. A type name precedes the variable(s) of that type. Simple variables have the types used in XML.

As an example, figure 3 shows some of the definitions for a document content analysis service. **Float** *length* and **String** *reason* are simple variables. **Reference** *words* is an endpoint reference to a service resource (a list of analysed words). CRESS can also define structured types. Here, *scores* is a structure containing a *length* field (a float) and a *frequency* field (an array of string elements called *word*). These fields might be accessed as *scores.length* or *scores.frequency[3]*.

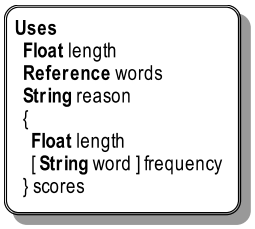

Figure 3: CRESS Rule Box Example

A BPEL handler deals with faults and events. In BPEL a handler may be defined inside any scope of a process, but in CRESS the 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.)

3.3. Formal Verification

CLOVE (CRESS Language Oriented Verification Environment, *www.cs.stir.ac.uk/~kjt/research/clove.html*) is a language for expressing desirable system properties for any application domain or specification language. Properties can be translated into multiple languages (μ-calculus in this article). The Patterns project has catalogued common verification properties; an online repository [26] gives example mappings to several temporal logics. Most properties apply to five scopes: **global** (always applies), **before** (some event), **after** (some event), **between** (always applies), and **after until**

Construct	Meaning
'string	a literal string
?type	a wild card for any value of the given type
\pattern	a regular expression; '\?type' prefixes a pattern value that is later referenced by number '*\n*'
service.port.operation	a grid service operation
always(*behaviour*)	all specification paths respect the behaviour
choice(*behaviour...*)	alternative choices among behaviours
exists(*behaviour*)	some specification path respects the behaviour
property(*name,behaviour*)	a property to be verified
response(*scope,behaviour*)	a response to some request, e.g. a **global** one that applies to all behaviours
signal(*operation,parameters*)	an input or output action

Table 2: Summary of CLOVE Constructs used in Article

(from one event to another). Pattern mappings have been created for μ-calculus (*www.inrialpes.fr/vasy/cadp/resources/evaluator/rafmc.html*). The subset of CLOVE constructs used in this article is summarised in table 2 and illustrated concretely in section 4.6.2.

Verification checks that desired properties are respected by a specification. Counterexamples that contradict a property can be generated by the analysis. Verification can analyse the entire behaviour, whereas validation is less general. A model is automatically generated from the specification as an LTS (Labelled Transition System) that is then checked against the desired properties.

As an example, figure 4 defines a property to be satisfied by the lender service. The property describes a global service response (i.e. in all circumstances). If a loan quotation request is made with any proposal details, the service must give one of two quotation responses: a number as the loan rate, or a refusal fault with some reason as a string.

```
property(Any_Loan_Response,
  response(global,
    signal(lender.loan.quote,?Proposal),
    choice(
      signal(lender.loan.quote,?Number),
      signal(lender.loan.quote,Refusal,?String))))
```

Figure 4: CLOVE Example

The CADP toolset (Construction and Analysis of Distributed Programs [13, 16]) uses finite-state model checking to verify specifications. A LOTOS specification is automatically translated by CADP into C code that is executed to generate the system LTS. CRESS automates the annotation of data types for CADP to generate the model. CADP verifies properties specified in RAFMC (Regular Alternation-Free Mu Calculus

[21]), a logic that expresses temporal properties including data values and also supports regular expressions.

Properties are automatically verified under control of the CLOVE tool. The automated translation turns data value enumerations into C and properties into μ-calculus. CADP supports on-the-fly verification with the EVALUATOR tool, managing systems with a large state space by constructing and exploring the state space incrementally. CLOVE uses EVALUATOR to verify the translated μ-calculus properties, with the C code and annotated LOTOS as inputs. For the non-specialist, verification outcomes are shown in CLOVE terms. If a property does not hold, counterexamples are provided. Common properties (e.g. deadlock, livelock, termination) are automatically checked.

Compositional verification is used to avoid state space explosion through divide-and-conquer. A large specification is automatically divided into smaller behaviours that are then composed. The CADP language SVL (Script Verification Language) is used by CLOVE for compositional verification tasks. This is automated for CRESS-generated specifications by identifying behavioural units. A composed service is divided into its service partners (recursively if they are business processes themselves).

3.4. Formal Validation

MUSTARD (Multiple-Use Scenario Test and Refusal Description [35], *www.cs.stir.ac.uk/~kjt/research/mustard.html*) is a language for expressing test scenarios for any application domain or specification language. The same tests can be translated into multiple languages (LOTOS and BPEL in this article). MUSTARD supports acceptance tests (the system behaves as expected) and refusal tests (the system refuses undesirable behaviour). MUSTARD also supports a variety of test behaviours: sequential or concurrent, deterministic or non-deterministic, modular, conditional or dependent on certain features, and using test fixtures (putting the system into a known state).

The subset of MUSTARD constructs used in this article is summarised in table 3 and illustrated concretely in section 4.6.3. MUSTARD is intentionally similar in style to CLOVE in order to facilitate learning and usage. A complete test scenario is defined recursively by combining simpler behaviours. The most basic behaviours are **read** and **send**, with most other constructs being combinators that build on these.

As an example, figure 5 gives a test to be satisfied by the lender service. The test must successfully be able to make a loan quotation request for a particular person, address and loan amount. However, as the amount is too large for this applicant, the response must be a refusal with reason ‘loan unacceptable’.

```
test(Lots_ Exceeds_ 15000,
  succeed(
    send(lender.loan.quote,Proposal(′Ian Carey,′Croydon England,15000.)),
    read(lender.loan.quote,Refusal,′loan unacceptable)))
```

Figure 5: MUSTARD Example

MUSTARD scenarios are validated by translating them into a target language (e.g. LOTOS) and then combining them with the specification to be validated. A language-specific tool (LOLA for LOTOS) then explores the state space of this test composition.

Construct	Meaning
'string	a literal string
?type	a wild card for any value of the given type
service.port.operation	a grid service operation
interleave(*behaviour,...*)	concurrent execution of behaviours
offer(*behaviour,...*)	the system offers a choice of alternative behaviours
read(*signal,parameters*)	inputs a signal from the system
send(*signal,parameters*)	outputs a signal to the system
sequence(*behaviour,...*)	sequential behaviour with abrupt termination
succeed(*behaviour,...*)	sequential behaviour with successful termination
test(*name,behaviour*)	a test scenario for the given name and behaviour

Table 3: Summary of MUSTARD Constructs used in This Article

Since scenarios always have finite behaviour, a scenario can be validated efficiently. Tests that fail are diagnosed and presented for the non-specialist in MUSTARD terms.

Much as for verification, specifications can be shown through validation to have desirable properties. Particular properties can be validated, e.g. the specification reacts in *this* way to *that* input. However, properties cannot be validated in general for classes of inputs. Properties like safety, liveness and starvation freedom also cannot be proven. However, validation is practicable when verification is not due to state space explosion.

4. Case Study – Document Content Analysis

This section presents a case study that uses grid services to perform document content analysis. Development begins with graphical description of the services. A formal specification is then automatically generated, verified and validated. Finally, a running implementation is automatically generated, deployed and evaluated.

4.1. Support for Grid Services

Grid services are an extension of web services. Differences include the following. CRESS supports all these aspects, and so is appropriate for grid services.

- Although both kinds of service have interfaces defined by WSDL (Web Services Description Language), grid services use extensions such for services properties. These are required for the grid service partners in the case study of this section.
- Grid services typically make use of resources through WSRF (Web Services Resource Framework [15]). These are identified through endpoint references and a particular addressing standard. Resources and endpoint references are used in this case study.
- Grid services can use dynamic partner binding. This means that a partner of a business process need not be fixed at design time. Rather, the partner is selected at run time (perhaps in response to selection criteria such as location, cost and

reputation). Although dynamic partners do not appear in this case study, they have been used by the authors for other services. CRESS handles these quite simply: a dynamic partner instance is an endpoint reference that is assigned to a **Partner** variable.

- Grid services often make use of GSI (Grid Security Infrastructure) to ensure that communication and access are secured. Grid security is not required in this case study, but has been used by the authors for other services. The CRESS configuration diagram (not illustrated in this article) defines the key characteristics of services. For a secure service, this gives the necessary credentials (typically username and password). CRESS also supports **Certificate** variables corresponding to the X.509 certificates that are widely used for security. These can be passed to a service for authentication or for credential delegation.

4.2. Content Analysis

Document content analysis (e.g. [19]) is used for many purposes such as investigating disputed authorship of a document, analysing different versions of a document, or comparing two documents for plagiarism. This is a rich field, so only a simplified version is given to illustrate how orchestrated grid services can be used. In this article, 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: Instances of each word are counted (disregarding common words) and the words are placed in order of decreasing frequency. This gives an ordered list 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 in one list does not appear in the other list, its position is considered to be at the end of that list. Thus if 'grid' were in position 1 of one list but not in the other list (of size 50), the distance would be $50 - 1$ or 49. 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.

Figure 6 shows the call structure for this example. The user invokes the matcher with two documents to be compared. This calls the parser to parse the documents, the scorer to compare them, and the counter to compute the similarity metrics.

4.3. Partner Services

The main services use external partner grid services that could exist already, or should be developed separately because they are generally useful:

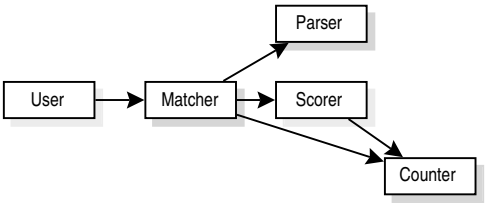

Figure 6: Call Structure for Document Content Analysis

Counter: This calculates the two metrics that compare documents. 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. The *delete* operation removes a stored word list.

4.4. Scorer Service

The scorer is a composite service that supports the main content analysis; its CRESS description appears in figure 7. The rule-box to the bottom right of the figure defines types and variables. The raw data is *words* (a reference to the documents being analysed). The result is *scores* (a structure containing the clause and frequency metrics). These variables are supplemented by *frequency* (an ordered word list), *length* (average clause length) and *reason* (for a fault).

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 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, nodes 3 and 4 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 spaces. Both invocations of the counter statically state 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).

4.5. Matcher Service

The matcher offers the primary content analysis service to the user. Its CRESS description appears in figure 8. The rule-box at the bottom right again defines types

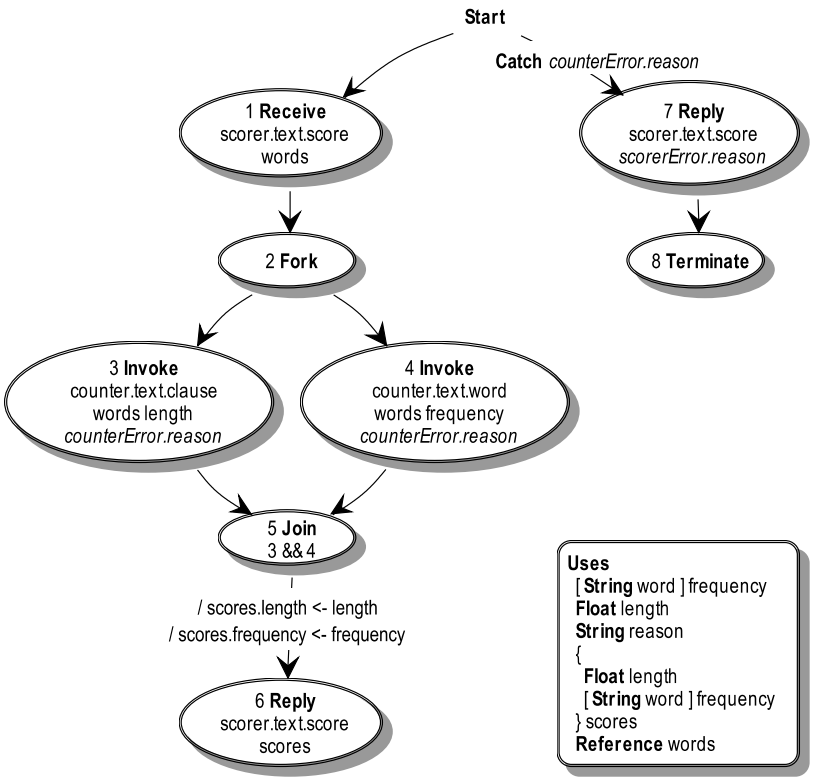

Figure 7: CRESS Description of The Scorer Service

and variables. The raw data is *texts* (text strings containing the two documents). The analysis yields *metrics* (clause length and word frequency distances). These variables are supplemented by *results* (metric scores for the two documents), *words1* and *words2* (references to the two word lists). The final entry in the rule-box '*/ scorer*' indicates that the matcher depends on the scorer diagram; dependencies on partner services are not shown as they are automatically inferred by CRESS.

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). For simplicity the similar parallel code is repeated explicitly, but could be commoned up. Each starts by setting the relevant 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 from nodes 9 and 10 must both be successful ('9 && 10' in node 11). The separately computed scores are combined (arc leading to node 12) and passed to the counter to compute distances (node 12). The matcher returns the resulting metrics to its caller (node 13).

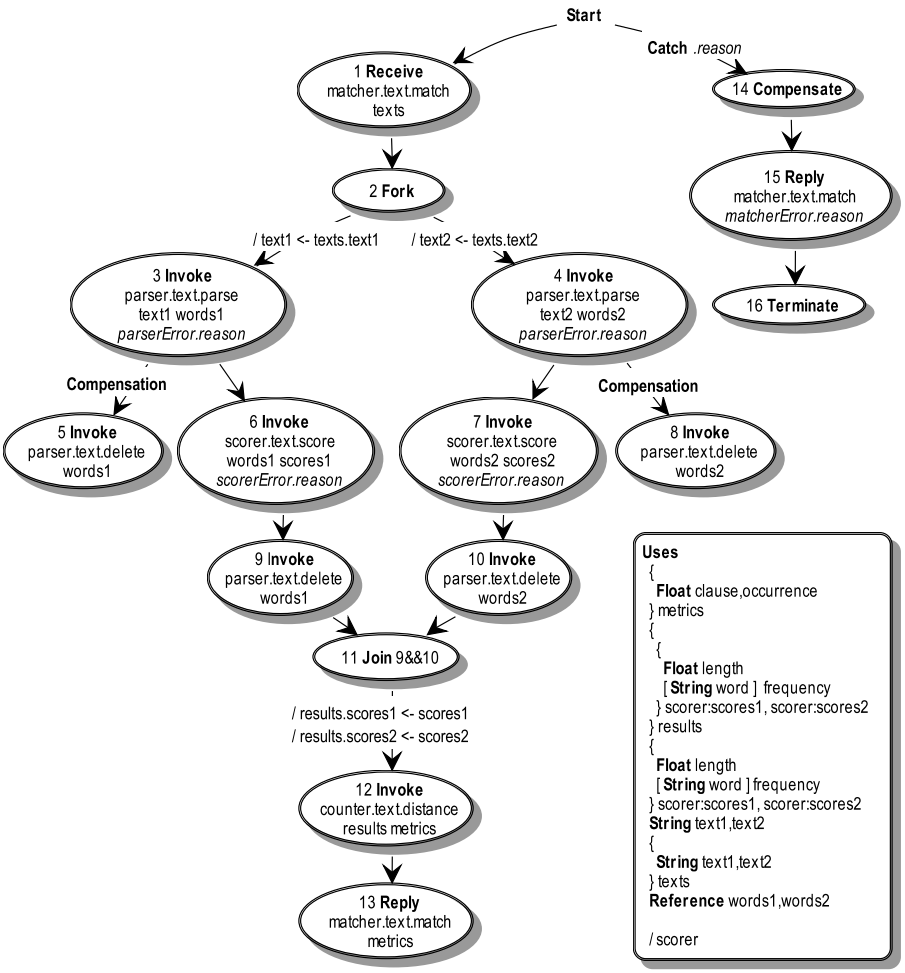

Figure 8: CRESS Description of The Matcher service

The matcher allows for faults in the services it calls: an invocation 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

```
property(Any_ Metric_ Response,
  response(global,
    signal(matcher.text.match,?Texts),
    choice(
      signal(matcher.text.match,?Metrics),
      signal(matcher.text.match,MatcherError,?String))))

property(Specific_ Texts,
  response(global,
    signal(matcher.text.match,
      Texts('Let sleeping dogs lie,'At night all cats are gray)),
    signal(matcher.text.match,Metrics(0.333333,1.0))))

property(Same_ Texts,
  exists(
    signal(matcher.text.match,Texts(\?String,\1)),
    always(signal(matcher.text.match,Metrics(0.0,0.0)))))
```

Figure 9: Example CLOVE Verification Properties

one parser invocation succeeds but the other fails, only the former will be compensated.

The matcher service orchestrates the use of two external partner services (counter and parser) as well as the scorer service (figure 7). 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.

A key question 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. With more complex data types as in this example, incompatibility is a bigger issue. Deadlock is a risk, as are more subtle problems due to semantic incompatibilities among the services. For these reasons, it is highly desirable to embed grid service development within a rigorous methodology.

4.6. Specification and Analysis

4.6.1. Automated Formalisation

The diagrams in figures 7 and 8 are automatically combined and translated into a LOTOS specification; the translation strategy is described in the early work of [34]. The details are not given here as they are likely to interest only LOTOS experts. However, the case study files can be found at *www.cs.stir.ac.uk/~kjt/software/download/gs-examples.zip*.

The service designer chooses LOTOS as the target language and clicks *Realise* in the CRESS diagram editor. This generates 1339 lines of LOTOS code: 21 processes and 24 data types, not including the library of data types for grid services.

4.6.2. Automated Verification

The CLOVE notation and approach for formal verification were described in section 3.3. Figure 9 (explained below) gives examples of CLOVE properties that are verified against the case study services. The service designer chooses LOTOS as the target language and clicks *Verify* in the CRESS diagram editor. The following is an extract of verification results:

Generating properties for SCORER ...	CPU Time	(Real Time)
Generating graph for SCORER ...	22.1 secs	(57.0 secs)
Verifying SCORER Livelock Freedom ...	6.2 secs	(10.0 secs)
Verifying SCORER Initials Safety ...	6.2 secs	(7.0 secs)
Verifying SCORER Always Exit ...	6.0 secs	(7.0 secs)

Generating properties for MATCHER ...	CPU Time	(Real Time)
Generating graph for MATCHER ...	22.4 secs	(1.7 mins)
Verifying MATCHER Any Metric Response ...	6.4 secs	(7.0 secs)
Verifying MATCHER Specific Texts ...	6.3 secs	(8.0 secs)
Verifying MATCHER Same Texts ...	6.3 secs	(7.0 secs)

Since verification requires a finite and manageable state space, free values in the specification must be constrained (typically those provided by the user to the system). The following CLOVE defines values of the *String* type used for document contents:

literals(**strings**,
  ′A stitch in time saves nine, ′At night all cats are gray, ′Barking dogs seldom bite, ...)

These are, of course, just small pieces of text used to check key properties of the specification. In practice, document comparison is performed with much larger texts. Although not required in this case study, data is often defined by regular expressions to allow a compact representation of a wide range of values. Like CRESS, CLOVE also supports values of structured types – though enumerated structures do not happen to be required in this case study. Values are automatically translated by CRESS into C to save manual coding (which is tedious for complex data types). CADP uses the enumerated values when generating the state space.

In figure 9, *Any_Metric_Response* verifies a global response property: all behaviours of the service must result in a given response to some request. Here, the request asks for a match of some unspecified pair of texts (*?Texts*). The response must be a choice of some metrics (*?Metrics*) or a matcher fault with some string value (*MatcherError*, *?String*). The following example, *Specific_Texts*, is again a global response property. This states that the service will give particular results (clause metric 0.333333, word metric 1.0) for a certain pair of texts. Finally, *Same_Texts* verifies that the same pair of texts will give a clause metric of 0.0 and a word metric of 0.0 (indicating identical texts). This makes use of the regular expression that is saved for the first text (\?*String*) and then re-used for the second text (\1).

As an example, the *Any_Metric_Response* property is automatically translated into the following μ-calculus:

```
[ true* . ′MATCHER !TEXT !MATCH !TEXTS (.*)′ ]
  mu X. (<true> true and [not(((′MATCHER !TEXT !MATCH !METRICS (.*)′) or
    ((′MATCHER !TEXT !MATCH !MATCHERERROR
      !"[−a−zA−Z0−9%^&*=+{}@∼#\\<>.\t]*"′)))) ] X)
```

```
test(No_ Clauses,
  succeed(
    send(matcher.text.match,'Each day we see,'),
    offer(
      read(matcher.text.match,matcherError,'No Clauses),
      read(matcher.text.match,matcherError,'No Words))))

test(All_ Shared,
  succeed(
    send(matcher.text.match,
      Texts('Sator Arepo Tenet Opera Rotas,'Arepo Sator Opera Rotas Tenet)),
    read(matcher.text.match,Metrics(0.0,0.4))))

test(Concurrent_ Use,
  succeed(
    interleave(
      sequence(
        send(matcher.text.match,Texts('Go West,'West Side Story)),
        read(matcher.text.match,Metrics(0.333,0.727))),
      sequence(
        send(matcher.text.match,
          Texts('Each day we see,'Round our ship terns and sea swallows)),
        read(matcher.text.match,Metrics(0.428,1.0))))))
```

Figure 10: Example MUSTARD Validation Scenarios

A typical formal approach to verification requires the designer to write properties like this. Evidently, μ-calculus is much more impenetrable than the CLOVE syntax, hence the value of higher-level property description using CLOVE.

If a property does not hold, a counter-example is automatically displayed in CLOVE form. This helps the designer to see why the specification does not behave as expected and to correct the problem.

Compositional verification (i.e. proving things in smaller pieces) is desirable for larger specifications such as the case study in this article. CLOVE automatically generates a script to perform compositional verification of the matcher. This is broken down into the scorer (composed in turn with the counter and the parser). The script generates the state space against which the translated μ-calculus properties are verified.

4.6.3. Automated Validation

The MUSTARD notation and approach for formal validation were described in section 3.4. Figure 10 (explained below) gives examples of MUSTARD scenarios to be validated against the case study services. The service designer chooses LOTOS as the target language and clicks *Validate* in the CRESS diagram editor. The following is an extract of validation results:

```
Test MATCHER No Clauses ...        Pass    0.7 secs
Test MATCHER All Shared ...        Pass    0.6 secs
Test MATCHER Concurrent Use ...    Pass    1.5 secs
```

In figure 10, *No_Clauses* checks for correct error handling behaviour. If the second text is empty, the matcher should report an error (either reason 'No Clauses' or 'No Words'). *All_Shared* checks for the case of two texts with the same words in a different order. The clause metric should be 0.0, while the frequency metric should be 0.4 due to the different ordering. *Concurrent_Use* exercises two concurrent calls of the matcher.

If a scenario fails its test, the behaviour up to the point of failure is displayed. This helps the designer to see why the specification does not behave as expected and to correct the problem.

4.7. Implementation and Evaluation

4.7.1. Automated Implementation and Deployment

Once verification and/or validation have been completed, a high degree of confidence has been obtained in the service design. The diagrams in figures 7 and 8 are then automatically combined, translated into a BPEL implementation, and deployed. The translation strategy is described in [34]. The details are not given here as they are likely to interest only BPEL experts, but the case study files can be found at *www.cs.stir.ac.uk/~kjt/software/download/gs-examples.zip*.

The service designer chooses BPEL as the target language and clicks *Realise* in the CRESS diagram editor. For the implementation, CRESS generates 9031 lines of source code and 68 source files. The large amount of code reflects the complexity of creating grid services, which need interface definitions and catalogues, deployment descriptors, property files, type classes, service and partner code, etc. The matcher and scorer are automatically compiled and deployed to ActiveBPEL [1] as the engine used to execute orchestrated grid services, while the counter and parser are automatically compiled and deployed to Globus [14] as the container for grid partner services.

4.7.2. Automated Implementation Evaluation

Implementation performance is evaluated by the *same* MUSTARD scenarios as used to validate the specification. The service designer chooses BPEL as the target language and clicks *Validate* in the CRESS diagram editor. This time the scenarios are translated into a form suitable for validating BPEL and are executed against the implementation by MINT (MUSTARD Interpreter). This evaluates consistency of service response times and pinpoints bottlenecks. When the authors performed this evaluation, it highlighted resource limitations in the Tomcat container that runs ActiveBPEL and required tuning for better performance.

Implementation validation is similar to specification validation (see section 4.6.3). However, additional results are produced if performance tests are requested. These can be executed sequentially (for checking consistent implementation behaviour) or in parallel (for loading the service). Additional statistics on test performance are reported, e.g. the averages and standard deviations of test execution times.

5. Evaluation

5.1. Usability of The CRESS Notation

Although CRESS describes services with a simple graphical notation, this does not necessarily mean that it is usable. A mixed empirical evaluation was therefore

conducted to check the following hypothesis: someone with experience of software development, with 45 minutes of training on the approach and the CRESS diagram editor, can define small services (up to five activities), with 80% accuracy, in at most 15 minutes per service.

Five software developers were recruited without previous experience of the CRESS approach: two female, three male, average age 25 (range 22 to 31). The participants were given written instructions to follow in their own time, without training or advice from the authors. A copy of the CRESS editor was provided for local installation, along with a 'palette' of typical symbols used in constructing services.

The first part of the instructions gave a three-page explanation of the approach and the CRESS editor, including three diagrams that the participants were asked to study and then to reproduce themselves using the diagram editor. 45 minutes was suggested as appropriate for this phase, though no time limit was imposed.

In the next part of the instructions, the participants were given five specific tasks to perform. Each task required a service diagram to be drawn (somewhat different from the examples), based on a natural language description. The participants were asked to record how long tasks took, and to save their diagrams on completion (or after 15 minutes if a task was not completed). The participants were asked to rate five statements about the approach on a five-point Likert scale. They were also given the opportunity to provide a free-form qualitative evaluation of the exercise.

All collected information was submitted by email. Task times and questionnaire answers were collected and analysed. Participant attempts at service diagrams were scored, comparing these against previously created sample solutions. Each possible element was given one mark (e.g. number, name, activity and parameters for a diagram node). This resulted in a percentage score for the accuracy of each diagram.

The participants spent an average of 34 minutes (range 10 to 60) on the familiarisation phase. This compares well with the expectation of 45 minutes. The shortest period (10 minutes) may reflect this participant's preference for learning by doing rather than extended prior study. Overall, participants completed the five service designs in an average of 5.7 minutes per task, with an average accuracy of 88% (compared to the hypothesis of 15 minutes and 80%).

The commonest errors in diagrams were omitting a node number (which two participants reasonably argued should be irrelevant or automatically generated), and simple syntax errors (such as using '/' rather than '\' before an assignment). In fact this was a knowingly demanding evaluation:

- The participants were given only a short written briefing and not an extended technical manual or training course. They had no opportunity for classroom instruction or one-to-one advice before undertaking the formal evaluation. This was deliberate, to see how readily the approach could be used with minimal instruction.
- Participants were asked to create diagrams without any way of machine-checking for errors. The full CRESS toolset (as opposed to the diagram editor) does, of course, check for syntactic and static semantic correctness. Indeed, all the syntax errors in the participant diagrams would have been readily identified and corrected in this way. Not providing the full CRESS toolset was again a delib-

erate decision, in order to discover the extent to which the approach exhibited syntactic idiosyncrasies that would trip up novices.

Participants were asked to rate five statements about the approach on a scale from 1 (strongly disagree) to 5 (strongly agree):

Statement 1: *I was able to create the service diagrams without too much difficulty*: average score 3.8 (range 3 to 4).

Statement 2: *I found it fairly straightforward to translate the English descriptions into diagrams*: average score 3.2 (range 1 to 4).

Statement 3: *I found it fairly straightforward to create and edit diagrams using* CRESS *editor*: average score 3.6 (range 3 to 4).

Statement 4: *I think the approach would be usable by people with experience of software development*: average score 4.0 (range 3 to 5).

Statement 5: *I think the approach could be useful in practice for developing services*: average score 3.2 (range 2 to 5).

The rating of statement 1 suggests that the approach is usable by the planned type of user (software developers), though the diagram editor could benefit from some technical improvements. The authors had expected statement 2 to be least agreed with, since significant mental effort is required to translate a natural language requirement into any formal representation (including programming languages). Like statement 1, the scoring of statement 3 offers encouragement – though improvements to the diagram editor are desirable. The evaluation of statement 4 suggests that an appropriate class of user has been targeted. Based on the accompanying free-form comments, the lack of a more positive response to statement 5 appears to reflect the need for improvements in the diagram editor rather than doubt over the general approach.

Given the short time that the participants spent in familiarisation (average 34 minutes), their performance was impressive. Although the limited number of participants does not allow statistically valid conclusions, the results of the evaluation are encouraging and favour the stated hypothesis.

In their free-form comments, the participants also provided valuable feedback on how the approach could be improved. In some cases the observations arose from the shortness of the written briefing, e.g. it was not mentioned that the editor indicated page boundaries with gray lines, and the syntax of assignments and conditions was only briefly illustrated. These points can readily be addressed through more extended training notes. Concrete suggestions to be considered include automatic node numbering, and use of a toolbox with typical grid service symbols.

5.2. Comparison with Other Approaches

5.2.1. General Capabilities

Implementation-oriented approaches like JOpera [24] and OMII-BPEL [8] support most of the constructs required by grid service orchestration, but they are generally not rigorous. BUnit (the BPEL analogue of JUnit for Java) offers a degree of systematic testing in ActiveBPEL.

Construct	WSAT	Ferrara	PEPA	LTSA	CRESS
Basic Activities	✓	✓	✓	✓	✓
Structured Activities	✓	✓	✓	✓	✓
Scoped Activities		✗	✗	✗	✗
Data Handling	✓	✓			✓
Dynamism					✓
Process Interaction	✓			✓	✓
Grid Services					✓
BPEL V1.1	✓	✓	✓	✓	✓
BPEL V2.0					✓

Table 4: Support of Service Orchestration Constructs (see text for explanation)

Some approaches have a rigorous aspect, though the degree of rigour varies. Since rigour is a key claim of CRESS, a comparison has been made with approaches that also support rigorous development. Of the techniques discussed in section 2.2, the following are the most directly relevant (though they are mostly for *web* services): WSAT (Web Services Analysis Tool [12], the work of Ferrara (translation of BPEL to LOTOS [9]), PEPA [22], and LTSA-WS (Labelled Transition System Analyser for Web Services [10]). Specifically, these have been compared with respect to coverage of orchestration, support for interacting processes, the level of abstraction, formalisation, verification, validation, implementation, deployment, and testing.

Few researchers have worked on complete methodologies for designing *grid* services. As a result, there is no standard example for a comparison of CRESS with other approaches. Instead, an example commonly used by *web* service developers has been chosen. This is a loan approval service [2] that offers online loans at a rate depending on the risk assessment.

5.2.2. Service Orchestration Coverage

Table 4 gives a comparison showing the coverage of BPEL constructs in the approaches considered. Here, ✓ means full support, ✗ means partial support, and blank means no support. The coverage of constructs is based on that stated by authors of the related work.

The lender service does not require some orchestration constructs, such as compensation and correlation. Basic activities include service input and output. Structured activities include alternatives, iteration, sequences and flows. Scoped activities include error, event and compensation handling. Data handling refers to data types, variables, assignments and expressions. Dynamism means that partner services can be chosen or changed at run time. Interaction allows multiple BPEL processes to interact with each other. Support for grid services is indicated. Finally, orchestration constructs may be supported from BPEL version 1.1 or 2.0.

Phase	WSAT	Ferrara	PEPA	LTSA	CRESS
Abstraction				✓	✓
Specification	✓	✓	✓	✓	✓
Verification	✓		✓✗	✓	✓
Validation				✓	✓
Implementation	✓	✓			✓
Testing					✓
Performance			✓		✓✗

Table 5: Support of Service Development Phases (see text for explanation)

5.2.3. Service Development Support

Table 5 compares the approaches with regard to coverage of grid services, abstraction, formal specification, formal verification, formal validation, implementation, testing and performance analysis. Although several approaches directly handle rigorous analysis, they differ in the extent and ease that this is supported.

WSAT has automated support only for translating BPEL into Promela for verification. The WSAT developers manually created two properties in Linear Temporal Logic and verified these using the SPIN model checker.

Ferrara's approach automates the translation between a LOTOS specification and a BPEL implementation. However, there is no automated tool support for verification. There are also only general hints about how this might be achieved, such as checking for bisimulation.

LTSA-WS abstracts the underlying techniques and tools in an effort to simplify and make analysis more accessible. This was achieved by using high-level notations, automated specification and analysis. LTSA-WS uses UML for design, specifically MSCs (Message Sequence Charts). Deadlock freedom can be checked for the Labelled Transition System created from BPEL code. Trace equivalence is automatically checked between models created from the MSCs and BPEL. Other properties have to be manually specified (e.g. request-response, safety). Although not demonstrated in [10] for the lender, the automated check of trace equivalence can detect errors such as interface incompatibilities. Validation is performed interactively and manually by animating the model. Analysis does not deal with data semantics. There is also no support for implementation validation.

The PEPA approach deals with translation and annotation from BPEL and WSDL to PEPA. Performance evaluation can then be performed with respect to latency and execution times. However the analysis does not focus on functionality. [22] also does not demonstrate how the approach helped to improve service quality.

CRESS allows verification of properties much as WSAT does, but the CRESS approach is more complete through the use of typical data values. In addition, there is support for abstract property specification and tool automation which WSAT does not provide. CRESS provides verification templates for well-known property patterns. The underlying temporal logic syntax is hidden, and realistic data values are supported by verification.

Depending on the specific data values, the time and state space for verification can vary widely. Verification with CRESS takes comparable times to WSAT. Properties of the lender service are typically verified by CRESS in minutes, for a range of 2000 distinct numbers and 5 distinct strings in loan requests. Due to wide variations in data inputs, this requires a state space with 104,000 states, 204,000 transitions and 50,000 transition labels. CRESS validation is automated for a range of scenarios that can include realistic data values.

The CRESS approach has been found in practice [30] to detect interesting errors in both specifications and implementations. Implementation validation uses black-box testing and load testing, neither of which is supported in the other approaches cited. Formal performance analysis in the style of PEPA is not currently supported, although it would be possible as a future extension since CADP supports this for LOTOS.

6. Conclusion

The CRESS notation, methodology and tools have been discussed, with a focus on formal aspects. Document content analysis case study has been used to illustrate the approach. The methodology supports automated specification, analysis and implementation.

Graphical descriptions of grid services are automatically translated into LOTOS. Specification properties expressed using CLOVE are then model checked. This hides the underlying complexities of μ-calculus and CADP from the service developer. To complement verification, formal validation uses scenarios expressed in MUSTARD. This hides the underlying complexities of LOTOS and LOLA. CLOVE and MUSTARD are abstract, language-independent and more straightforward for the non-specialist than the underlying formalisms. After confidence has been built in the service design, an implementation is automatically built and deployed. The same MUSTARD scenarios can then be used to evaluate the performance of the implementation.

From the designer's point of view, development mainly requires drawing flow diagrams that describe services. This provides a convenient route to service implementation if only service creation is needed. As has been seen, CRESS automates the creation and deployment of a large amount of implementation code. However, this code would then have to be debugged in conventional ways.

With some additional effort it is possible to do much better. Implementation and performance testing can be automated through MUSTARD by defining test scenarios in a reasonably straightforward language. For the same effort, the service design can then be evaluated at a much earlier stage by validating the specification. This allows problems to be discovered much earlier during development.

However, testing can cover only a limited portion of a design. General properties (e.g. deadlock/livelock freedom, termination) can be verified by CLOVE without having to write any definitions. With additional work on defining desirable service properties, the design can be verified more thoroughly. This gives much wider coverage by checking general properties of the design rather than just specific scenarios. Although CLOVE is more accessible to non-specialists than the underlying formal techniques, it is acknowledged that formulating appropriate properties does need experience. For-

tunately CLOVE was able to build on the knowledge codified by the Patterns project about properties that are most likely to be useful in practice.

A modest evaluation of the CRESS notation has assessed its usability and found it to be satisfactory. In future work, the usability of CLOVE and MUSTARD notation will also be evaluated. Service development with CRESS has also been compared to other approaches and found to cover a wider range of capabilities.

Although this article has considered only grid services and formalisation using LOTOS, CRESS deals with other service domains (e.g. voice, device, web) and their associated languages (e.g. Call Processing Language, Specification and Description Language, VoiceXML). CRESS therefore offers a general-purpose approach to delivering verified and validated services.

Acknowledgements

Larry Tan was supported by an Overseas Research Studentship, by the University of Stirling, and by the Economic and Social Research Council (grant RES-149-25-1066).

[1] ActiveBPEL Community. ActiveBPEL workflow engine. *http://activebpel502.sourceforge.net*, Dec. 2011.

[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. Organization for The Advancement of Structured Information Standards, Billerica, Massachusetts, USA, Apr. 2007.

[3] M. Bozkurt, M. Harman, and H. Hassoun. Testing web services: A survey. Technical Report TR-10-01, Computer Science, King's College, London, Jan. 2010.

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

[5] J. Bradfield and C. Stirling. Modal mu-calculi. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*. Elsevier Science Publishers, Amsterdam, Netherlands, 2007.

[6] M. Butler, C. Ferreira, and M. Y. Ng. Specifying and verifying web transactions. *Universal Computer Science*, 11(5):712–743, May 2005.

[7] 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.

[8] 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.

[9] 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.

[10] H. Foster. *A Rigorous Approach to Engineering Web Service Compositions*. PhD thesis, Imperial College, London, Jan. 2006.

[11] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Supercomputer Applications*, 35(6), 2002.

[12] 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.

[13] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24, Aug. 2002.

[14] Globus Alliance. Globus toolkit. *http://www.globus.org*, Dec. 2011.
[15] 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.
[16] INRIA Vasy Team. CADP (Construction and Analysis of Distributed Processes). *http://www.inrialpes.fr/vasy/cadp*, Nov. 2011.
[17] 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.
[18] N. Kaveh and W. Emmerich. Validating distributed object and component designs. In M. Bernardo and P. Inverardi, editors, *Formal Methods for Software Architecture*, number 2804 in Lecture Notes in Computer Science, pages 63–91. Springer, Berlin, Germany, Sept. 2003.
[19] K. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. Sage, Thousand Oaks, California, USA, 2004.
[20] J. Li, H. Zhu, and J. He. Specifying and verifying web transactions. In K. Suzuki, T. Higashino, K. Yasumoto, and K. El-Fakih, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE 2008)*, number 5048 in Lecture Notes in Computer Science, pages 168–183. Springer, Berlin, Germany, June 2008.
[21] R. Mateescu. Property pattern mappings for RAFMC. *http://www.inrialpes.fr/vasy/cadp/resources/evaluator/rafmc.html*, May 2009.
[22] B. Mitchell and J. Hillston. Analysing web service composition with PEPA. In J. Bradley, editor, *Proc. 3rd Workshop on Process Algebras and Stochastically Timed Activities*, pages 33–44, Edinburgh, June 2004. University of Edinburgh.
[23] 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.
[24] 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.
[25] S. Pavón Gomez, D. Larrabeiti, and G. Rabay Filho. LOLA user manual (version 3R6). Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, Feb. 1995.
[26] Santos Laboratory. Specification patterns. *http://patterns.projects.cis.ksu.edu*, Dec. 2011.
[27] Sensoria Consortium. Software Engineering for Service-Oriented Overlay Computers. *http:/www.sensoria-ist.eu*, Sept. 2010.
[28] S. Staab, W. van der Aalst, V. R. Benjamins, A. Sheth, J. A. Miller, C. Bussler, A. Maedche, D. Fensel, and D. Gannon. Web services: Been there, done that. *IEEE Intelligent Systems*, 18(1):72–85, Jan. 2003.
[29] B. Steffen, T. Margaria, R. Nagel, S. Jörges, and C. Kubczak. Model-driven development with the jABC. In E. Bin, A. Ziv, and S. Ur, editors, *Hardware and Software, Verification and Testing*, number 4383 in Lecture Notes in Computer Science, pages 92–108. Springer, Berlin, Germany, May 2007.
[30] K. L. L. Tan. Case studies using cress to develop web and grid services. Technical Report CSM-183, Computing Science and Mathematics, University of Stirling, UK, Dec. 2009.
[31] K. L. L. Tan and K. J. Turner. Automated analysis and implementation of composed grid services. In D. Dranidis and I. Sakellariou, editors, *Proc. 3rd South-East European Workshop on Formal Methods*, pages 51–64. South-East European Reseach Centre, Thessaloniki, Greece, Nov. 2007.

[32] J. Tretmans. Test case derivation from LOTOS specifications. In S. T. Vuong, editor, *Proc. Formal Description Techniques II*. North-Holland, Amsterdam, Netherlands, Dec. 1989.

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

[34] 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.

[35] K. J. Turner. Validating feature-based specifications. *Software Practice and Experience*, 36(10):999–1027, Aug. 2006.

[36] K. J. Turner. CRESS (Communication Representation Employing Systematic Specification). *http://www.cs.stir.ac.uk/cress*, Dec. 2011.

[37] B. Wassermann, W. Emmerich, B. Butchart, N. Cameron, L. Chen, and J. Patel. Sedna: A BPEL-based environment for visual scientific workflow modelling. In I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors, *Workflows for E-Science*, pages 428–449. Springer, Berlin, Germany, 2007.

[38] M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch, and A. Schröder. Sensoria process calculi for service-oriented computing. In E. Najm and J.-F. Pradat-Peyre, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE 2006)*, number 4229 in Lecture Notes in Computer Science, pages 24–45. Springer, Berlin, Germany, Jan. 2006.

[39] J. Yu, J. Han, P. Falcarin, and M. Morisio. Using temporal business rules to synthesize service composition process models. In M. van Sinderen, editor, *Proc. 1st Int. Workshop on Architectures, Concepts and Technologies for Service Oriented Computing*, pages 86–95. INSTICC Press, Setúbal, Portugal, July 2007.