A Rigorous Methodology for Composing Services

Kenneth J. Turner and Koon Leai Larry Tan

Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK
{kjt,klt}@cs.stir.ac.uk

Abstract. Creating new services through composition of existing ones is an attractive option. However, composition can be complex and service compatibility needs to be checked. A rigorous and industrially-usable methodology is therefore desirable required for creating, verifying, implementing and validating composed services. An explanation is given of the approach taken by CRESS (Communication Representation Employing Systematic Specification). Formal verification and validation are performed through automated translation to LOTOS (Language Of Temporal Ordering Specification). Implementation and validation are performed through automated translation to BPEL (Business Process Execution Logic) and WSDL (Web Services Description Language). The approach is illustrated with an application to grid service composition in e-Social Science.

1 Introduction

1.1 Motivation

Workflows have been widely adopted to create new services by composing existing ones. Grid services are similar to web services, so it is not surprising that common mechanisms can be used with both to combine services. Such composite services are becoming increasingly common in commercial and scientific applications. They can require complex logic to combine independently designed services. Compatibility with third-party services can also be an issue.

It is therefore desirable to have a rigorous methodology for creating and analysing composed services. However, formal approaches are mostly restricted to computer scientists and are hard to sell to industry. This paper reports on work to encourage use of formal methods in the field of grid and web services:

- an accessible graphical notation is used to describe composite services
- formal models are automatically created, validated and verified without requiring detailed knowledge of formal methods
- implementations are automatically created and deployed once services have been validated and verified.

1.2 Composing Services

Grid computing allows heterogeneous systems and resources to interoperate following the paradigm of SOA (Service Oriented Architecture). New services can be created by combining existing ones. The terms ‘composition’, ‘orchestration’ and ‘workflow’ are

all very similar, and are used interchangeably in this paper. BPEL (Business Process Execution Language [1]) is a standardised approach for orchestrating *web* services. The authors and others have investigated techniques for orchestrating *grid* services.

Service composition raises a number of issues. The logic that combines services can become complex. Sophisticated error handling may also be required. Compatibility of component services may be a concern, especially if the services are defined using only WSDL (Web Services Description Language). Since WSDL describes just interface syntax and not semantics, deeper issues of compatibility can arise.

A methodology is hence desirable for developing composed services. Defining compositions should be made straightforward since the developer may well have a limited computing background. Verification ('doing the thing right') should allow the service composition to be automatically checked against desirable properties. Once confidence has been built in the design, implementation and deployment should be fully automatic. Validation ('doing the right thing') should be possible during specification (to build confidence in the design) and also after implementation (to check non-functional properties such as performance and dependability).

Grid and web services differ in their emphasis on use of resources, e.g. for processing, distributed data and specialised devices. Resources are often used by grid services to offer stateful services to clients. It is therefore necessary to formally model interactions with dynamic resources in grid service composition. For flexibility, it should also be possible to dynamically allocate the partners that support a composite service. Partners are third-party services that are combined through workflow logic to offer a new, composite service.

1.3 Service Composition Methodology

Surprisingly little attention has been given to rigorous composition of grid services (though more has been done on composed web services). Even where this has been studied, formal models are usually developed separately from their implementations. In contrast, the work reported here is a complete methodology that handles all aspects of service creation, from initial design through to system testing. This approach is called CRESS (Communication Representation Employing Systematic Specification, *www.cs.stir.ac.uk/~kjt/research/cress.html*). In fact, CRESS was designed for modelling many kinds of services and has been applied in many domains. For grid and web services, CRESS can be viewed as a workflow language for specifying composite behaviour.

Early work by the authors demonstrated that grid service composition could be achieved by adapting BPEL. However, there were significant limitations in BPEL that required work-arounds (e.g. for EndPoint References). Composed grid services were also not much more than simple web services, e.g. there was no support for service resources and dynamic partners in the style that grid services commonly use. Only formal validation was supported. The new work reported in the present paper has resulted in a rounded methodology for orchestrating services. Formalisation has been extended to deal with full grid services. Formal verification and implementation validation have also been added to the methodology.

There are several advantages to this approach. A composite service need be described only once, using an accessible graphical notation. The formal specification and

the implementation code can then be automatically generated from this single description. Automatic formal verification and formal validation can be used to ensure that the service composition is functionally correct. Errors at the design stage can be cheaply corrected by modifying how the composition is described. Implementation and deployment are then fully automatic. Although further checking might appear unnecessary, a range of practical issues make implementation validation desirable. For example performance bottlenecks may arise, or factors such as dependability and reliability might need attention. Although the methodology described in this paper supports all aspects of composing services, the emphasis here is on a new and distinct facet: how formal verification and validation can be supported.

1.4 Relationship to Other Work

Specifying Composed Services Formalising *web* services has been studied by the formal methods community. LTSA-WS (Labelled Transition System Analyser for Web Services [8]) is a finite state method. 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 limited to handling data types but not their values. This restricts the formal analysis of service composition since data values are often used in conditions that influence behaviour. CRESS differs in generating the formal model and the service implementation from a single abstract description. CRESS uses LOTOS (Language Of Temporal Ordering Specification [12]) to model service composition, and can therefore model data types as well as their values.

Temporal business rules have been used to synthesise composite service models [21]. The pattern-based specification language PROPOLS (Property Specification Pattern Ontology Language for Service Composition) expresses these rules. Each rule has a predefined finite state automaton to represent it. A behavioural model is then generated by composing the rules using their respective finite state automata. This can be further iterated with additional rules until a satisfactory model is generated. The process model can then be transformed into BPEL, although this aspect appears to be under development. The approach does not, however, deal with data types. CRESS differs in generating both the implementation and the formal specification from the same CRESS description, dealing fully with data types and values.

WSAT (Web Services Analysis Tool [9]) is used to analyse and verify composite web services, particularly focusing on asynchronous communication. Specifications can be written bottom-up or top-down, finally being translated into Promela and model-checked using SPIN. For composite web services that interact asynchronously, WSAT is able to verify the concepts of synchronisability and realisability. However, the tool does not support the full range of capabilities found in standards such as BPEL. A composite web service specification often deals with error handling, compensation and correlation – things that are not yet handled by WSAT.

[4, 7] use a process algebraic approach to 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. This is an advantage as the service developer may well not be familiar with either BPEL or LOTOS.

Implementing Composed Services Web service orchestration has been actively studied and supported in a number of pragmatic developments. There are several implementations for modelling and executing service workflows.

JOpera [14] is a service composition tool for building new services by combining existing ones. It provides a visual composition language and also a run-time platform to execute services. JOpera claims to offer greater flexibility and expressivity than BPEL. Although JOpera initially focused on web services, support for grid service composition has also been investigated.

Taverna [13] was developed to model web service workflows – specifically for bioinformatics. It introduced SCUFL (Simple Conceptual Unified Flow Language) to model grid applications in a specialised workflow language.

BPEL has been investigated by several researchers for orchestrating grid services. [16] developed BPEL extensibility mechanisms to orchestrate services based on OGSI (Open Grid Service Infrastructure) and WSRF (Web Services Resource Framework [11]). [22] used specialised constructs to achieve interoperability with WSRF services. These efforts showed that grid service orchestration was possible, but restricted.

Since web services may vary dynamically, partner services may become inconsistent with respect to workflows that rely on them. ALBERT (Assertion Language for BPEL Process Interactions [2]) is a language for expressing (non)-functional properties of workflows. The continued validity of these properties can be monitored at run time.

OMII-BPEL (Open Middleware Infrastructure Institute BPEL [19]) aims to support the orchestration of scientific workflows with a multitude of service processes and long-duration process executions. It provides a customised BPEL engine, and supports a set of constructs desirable for specification of scientific workflows.

The OMII-BPEL work is the closest to CRESS. The authors strongly believe that implementations should be created in standard languages (BPEL, WSDL, XSD) which are already widely used. For example, this allows the use of a variety of orchestration engines. Where CRESS differs from similar BPEL approaches is that it takes a more abstract (and even language-independent) view. Specification, implementation and analysis can therefore be integrated in a single methodology.

2 Background

2.1 Service Composition and Grid Services

SOA (Service Oriented Architecture) treats capabilities or functions as individual services. Service composition is a key feature of SOA for creating new services by combining the behaviour of existing ones. BPEL (Business Process Execution Language [1]) is one of the most popular languages for specifying composite *web* services. Although early work on composing *grid* services using BPEL showed promise, this was not straightforward. Fortunately, the latest standard for BPEL supports WSRF (Web Services Resources Framework [11]) and is hence appropriate for grid services.

WSRF allows a service instance to be associated with arbitrary numbers and types of resources. 'Resource pairs' are identified by an EPR (EndPoint Reference [20]). Grid services promote virtual collaboration among users of distributed systems. A grid environment can be highly dynamic, with resources, partners and services being created, added, shared and removed over time.

Grid computing initially developed through applications in the physical sciences. The trend is now towards use in other areas such as e-Social Science, which has been recognised as a promising application of grid computing. The authors are formalising support for workflows on the DAMES project (Data Management through e-Social Science, *www.dames.org.uk*).

To illustrate the methodology for developing workflows, this paper tackles a common task performed by social scientists: representing occupations in different classification schemes. Occupational data researchers are interested in analysing questions such as how jobs affect social position, social interaction patterns, etc. There are many occupational classification schemes, some of them international standards. As each classification scheme favours certain types of analysis, occupational researchers have to map datasets to particular schemes to perform the analysis. This might involve several intermediate mappings to arrive at the desired encoding. As a result, translation is often performed using computer scripts or paper indexes that map between (usually) two schemes. Sections 3 and 4 discuss how an occupational translation service was rigorously developed using service composition.

2.2 CRESS

CRESS is a domain-independent graphical notation for describing services. CRESS takes an abstract approach in which a high-level service description automatically generates a formal specification and an executable implementation. In other work, it has been used to describe a variety of voice services and also web services. CRESS can be used as a graphical workflow notation for grid and web services.

CRESS service descriptions are graphical, making the approach accessible to non-specialists. The focus is on high-level description, abstracting away the technical details required in an actual implementation. CRESS is designed as an extensible framework where support for new domains and target languages can be added like plug-ins.

The CRESS representation for service composition is intentionally close to BPEL. A brief description of the subset of CRESS notation used in this paper is given here. Refer to figures 1 and 2 for the examples cited below.

A CRESS diagram typically includes a rule box, numbered nodes, and arcs that link nodes. A rule box is a rounded rectangle which (for grid and web services) defines variables and their types, as well as dynamic partners. Complex data structures can be defined, e.g. '{...}' for records. As an example, the following defines two variables *mapping1* and *mapping2* whose type is a record with two string fields:

{ **String** job **String** scheme } mapping1:allocator, mapping2:allocator

Variables and their types are normally associated with the diagram that define them. It is possible to be explicit about this by qualifying a variable with its owning diagram (*Allocator* in the above).

A rule box can also indicate which other services are required, e.g. '/ Allocator' in the description of the *Lookup* service shows it depends on the *Allocator* service.

The activities in a composed service are described in numbered ellipses. A typical composition starts with **Receive** as an incoming request that specifies the service, port and operation names, as well as the input variable. A typical composition ends with a **Reply** as an outgoing response that returns an output variable or a fault. There can be alternative **Reply** activities for one **Receive**, and even several **Receive** activities. **Invoke** is used to call an external partner by service, port and operation. Invocation specifies an output variable, an optional input variable, and optional faults that may be thrown. Examples of all of these are:

Receive lookup.job.translate schemes	(input *schemes*)
Reply lookup.job.translate codes	(output *codes*)
Reply lookup.job.translate allocatorError.reason	(fault *allocatorError.reason*)
Invoke allocator.job.translate mapping1 code1	(output *mapping1*, input *code1*)

Faults can be defined with just a name (*allocatorError*), with just a value (*.reason*), or with both elements.

Other activities include **Terminate** (to end behaviour), **Compensate** (to undo work following a failure), and **Fork/Join** (for concurrency). For the latter, a fine degree of control over concurrency can be specified. In general, each activity may complete successfully or may fail. A join condition such as '3&&4' means that activities 3 and 4 must succeed before behaviour continues. Activities as well as arcs can contain assignments such as '/ mapping1.job <– schemes.job'.

Branches in CRESS diagrams normally represent choices. A deterministic choice has labels on arcs for conditions that govern which path is followed. A non-deterministic choice has unlabelled arcs. Event choices are not made immediately, but rather when some event happens. For example, a '**Catch** *.reason*' branch is followed only when a fault with some *reason* value occurs. A **Compensation** branch is taken only when a **Compensate** activity is used to undo previous work. Typically, compensation is defined after an **Invoke** since a failure may mean that changes already made have to be undone.

3 Formal Specification and Analysis of Composed Grid Services

3.1 Describing Service Composition

The service developer starts by drawing a CRESS diagram that describes the logic used to combine the functions of external service partners. Typically these partners have already been created by others, though new partner services might also be created for the purposes of the orchestration. In a complex development, a number of CRESS diagrams may be defined to realise the orchestration. CRESS also supports feature diagrams for common functions that can be added automatically to service descriptions.

During the work reported in this paper, CRESS was extended to treat EPRs (End-Point References) as first-class values, to support grid resources fully, to handle dynamic service partners, to formally verify properties, and to validate implementations. These aspects are all illustrated using the following example.

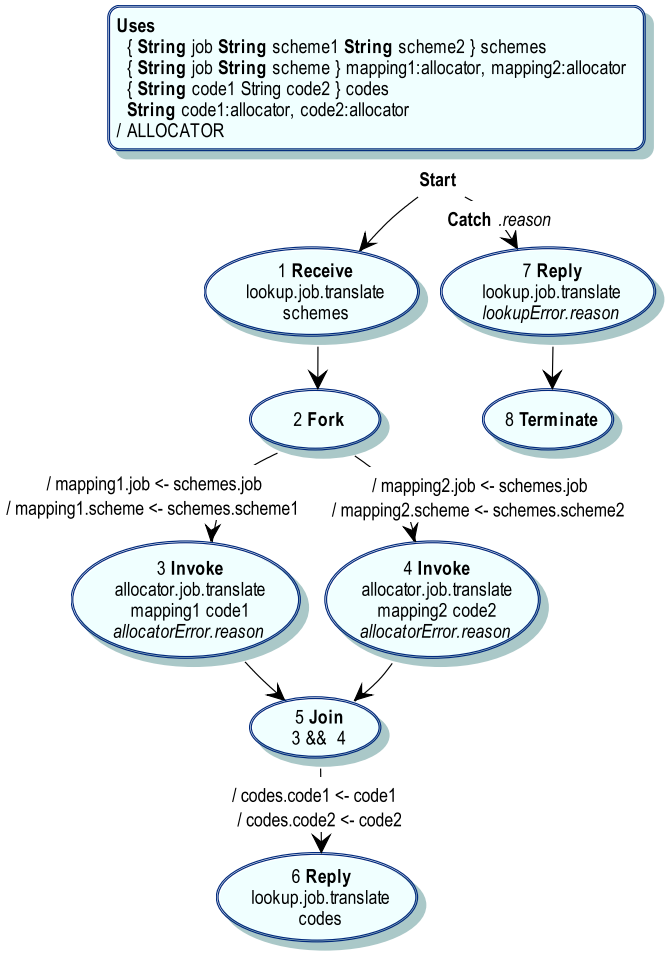

Fig. 1. CRESS Description of The Occupation Lookup Service

The diagrams in figures 1 and 2 show the use of CRESS to describe an e-Social Science workflow. This supports the classification of occupations mentioned in section 2.1. The services involved in this example are as follows:

Lookup: This is the top-level workflow that takes a request to translate a job title into two occupational schemes. It uses the *Allocator* partner to perform these translations in parallel, and returns the combined result.

Allocator: This partner service is itself a workflow that takes a request to map an occupation into some scheme. It uses the *Factory* partner to find a suitable resource (i.e. a *Mapper* service) to perform this translation and then return the occupation code.

Factory: This partner service accepts a request to find an occupational classification translator. It dynamically allocates a resource for performing this task, and returns a reference to it. If no suitable resource can be allocated, a fault is thrown.

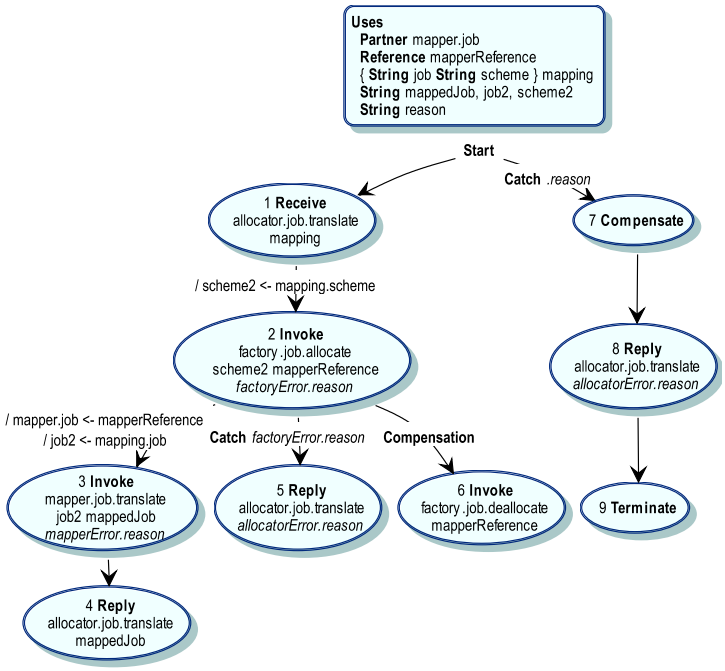

Fig. 2. CRESS Description of The Occupation Allocator Service

Mapper: This partner service is selected dynamically, so it represents a class of translation services. The given job title is translated into a particular occupational scheme.

Lookup Service The service in figure 1 defines translation logic that makes use of the partner *Allocator* service. Initially the service proceeds along the arc from **Start** to node 1. The service then accepts a request to translate a job title into two specified schemes (node 1). The translations may be automated (e.g. through an online service) or may be manual (e.g. requiring a researcher to look up a classification scheme). Since the delay in translation for each scheme is unknown, both translations are performed in parallel (node 2 to node 5).

Both parallel branches are similar. For example, the left-hand branch copies the *job* and *scheme* names into the *mapping1* structure (arc from node 2 to node 3). The *Allocator* service is then called to translate this into a job code (node 3). This service may give rise to a fault if the translation cannot be performed (fault name *allocatorError*, fault value *reason*). Both parallel branches require to complete successfully before further action is taken, as specified by the join condition in node 5. At this point, the two job codes are combined into a *codes* structure (arc from node 5 to node 6). The *Lookup* service ends by sending this to its caller (node 6).

In passing, note that CRESS supports long-running transactions that are fairly common in grid computing. In the case of *Lookup*, the parallel invocations may take as long as required. This is not desirable if the service instance could otherwise make progress

on other tasks. CRESS therefore also allows a one-way **Invoke** that immediately returns to the calling service. An asynchronous response is matched by a **Receive**.

Concurrency requires proper handling of faults. For example, if one of the parallel branches fails then the other cannot be left hanging. The top-level error handler (arc from **Start** to node 7) catches a fault from either parallel invocation. It replies to the caller with the reason why the *Allocator* failed, and terminates the whole workflow.

Allocator Service The service in figure 2 initially proceeds along the arc from **Start** to node 1. Here it accepts a request for a particular job mapping. The classification scheme is extracted into *scheme2* (arc from node 1 to 2). Since it is necessary to find a suitable translation service, the *Factory* partner is called to find one for the particular scheme (node 2). This returns a *Mapper* reference for a suitable service instance. If no suitable service is found, a fault is thrown.

The *Allocator* then dynamically sets the reference for the *Mapper* service, and extracts the job title into *job2* (arc from node 2 to node 3). When the *Mapper* is called to translate the job title, this dynamic partner is used (node 3). In normal circumstances, the *Allocator* replies with the job code to its caller (node 4).

Various error conditions are handled by the *Allocator*. If the *Factory* invocation in node 2 fails, the error is caught in the local scope and returned to the caller (node 5). A mapping failure in node 3 needs to be handled differently. Since no fault handler is defined for this invocation, the global fault handler is used (arc from **Start** to node 7). This requires compensation because simply terminating the *Allocator* would leave the translation resource allocated. **Compensate** in node 7 requests global compensation. All subsidiary compensation activities are then called in reverse order of completion. In this example, there is only one such activity (arc from node 2 to node 6). The effect is to deallocate the service instance that the *Factory* had allocated (node 6). Following compensation, the *Allocator* returns the fault *reason* to its caller and terminates the workflow (nodes 8 and 9).

Collectively, figures 1 and 2 define a composite service with four partners. However, a client of the whole translation service sees just a single grid service; the internal design of this is intentionally hidden, and could be changed in future.

3.2 Formalising Service Composition

A CRESS diagram is automatically translated into LOTOS (Language Of Temporal Ordering Specification [12]), including support for developer-defined data types and behaviour. (A number of formal approaches to grid or web services support only elementary data types such as booleans and integers.) Service behaviour is represented by interacting LOTOS processes. As the focus is on service composition, CRESS fully specifies the logic that combines external partner services. CRESS does not normally have enough information to specify these partners, and instead defines only their interfaces. However if a partner service is itself a composition, CRESS will specify it fully.

Since partner services are usually defined by others, it is likely that no formal specification exists of them. Indeed, the design of a partner service may be proprietary and hidden. The automated interface specifications generated by CRESS are sufficient for

basic compatibility checks of partners. For a more thorough analysis it is desirable to have more complete (though still abstract) specifications of partner services. These specifications have to be created manually, by the developer of the partner service or by the developer of the composite service. However, having a formal specification of all services is good practice anyway.

Handling of dynamic resources in LOTOS has been added for the work reported here. For static service partners, interactions between a composite service and a service partner are via LOTOS events that specify the service, port and operation. For dynamic service partners, synchronisation is specified with a resource prior any interaction. This is reasonable as an actual implementation also does the same thing.

The **Partner** type in CRESS is a unique key that identifies a resource pair. An assignment to *partner.port* is performed prior to invoking a dynamic partner. In LOTOS, this is translated as an assignment to the corresponding EPR variable. Synchronisation with a dynamic partner specifies the EPR required. It is only after this that a dynamic partner instance can be invoked. This approach can also be used with web services, since dynamic web partners operate in an identical fashion.

Figure 3 shows how the various specification elements are combined in the CRESS methodology. The generated specification of a composite service normally dominates the specifications of the partner services. The composite LOTOS specification that results is sufficient for use with several LOTOS tools, e.g. LOLA (LOTOS Laboratory [15]). However, some LOTOS tools such as CADP (Construction and Analysis of Distributed Processes [10]) require the specification to be preprocessed first.

A CRESS service description is rigorously analysed through formal validation and verification of the automatically generated LOTOS. Once the composition has been checked to have the desired properties, an implementation can be created automatically.

3.3 Validating Service Compositions

Formal validation can be directly performed on the composite specification produced by CRESS. This makes use of a test notation and tool called MUSTARD (Multiple Use Scenario Test and Refusal Description [18]). Although not illustrated here, MUSTARD can be used to test partner services as well as service compositions.

As a simple example of validation with MUSTARD, the following acceptance test checks the translation of job title 'nurse' into the SOC2000 and SIC92 classifications (codes '3211' and '95.14' respectively). The test succeeds if it is possible to send a translation request and then to read the expected response. Strings in MUSTARD are preceded by a single quote.

```
test(Nurse_Translation,
  succeed(
    send(lookup.job.translate, schemes('Nurse,'SOC2000,'SIC92)),
    read(lookup.job.translate, codes('3211,'95.14))))
```

Acceptance tests check only what a system must do. MUSTARD is also used to define refusal tests that check what a system must not do. Concurrent behaviour can be checked as well. In the following test, parallel requests are performed to translate a job title using different occupational schemes:

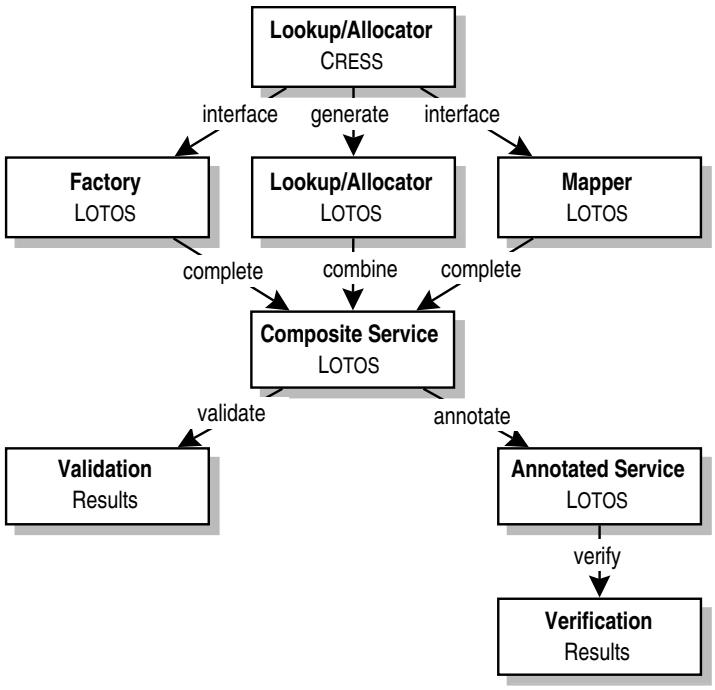

Fig. 3. Formal Validation and Verification with CRESS

```
test(Parallel_Translation,
  succeed(
    interleave(
      sequence(
        send(lookup.job.translate, schemes('Cab Driver,'SOC2000,'SIC92)),
        read(lookup.job.translate, codes('8214,'60.22))),
      sequence(
        send(lookup.job.translate, schemes('Private Detective,'SIC92,'SOC2000)),
        read(lookup.job.translate, codes('74.60/1,'9241))))))
```

MUSTARD translates such tests into LOTOS, adds them to the composite specification generated by CRESS, and uses the validation facilities of LOLA to formally check that the specification passes its tests. This is achieved through abstract execution of the specification, constrained by the test behaviours. The tests above are simple examples. In practice, MUSTARD is used for a variety of tests that may include alternatives, conditions, non-determinism, variables, wild-card values, service dependencies (whether a particular service is deployed), fixtures (common preambles for tests), and reset actions (to put a service into a known state). As will be seen in section 4, MUSTARD tests are used to validate implementations as well as specifications.

Although such validation is formally based, testing is necessarily limited. Its main advantage is that validation is practical; automated validation of even a complex service is performed in seconds or minutes. However, formal verification is desirable as a com-

plement to this. Rather than showing that the specification exhibits desirable behaviour on certain test cases, it is preferable to prove properties in general for classes of tests.

3.4 Annotating LOTOS

For formal verification of LOTOS, the toolset of choice is CADP (Construction and Analysis of Distributed Processes [10]). However, CADP places a number of restrictions on the form of LOTOS that it will accept. In particular, data types need to be extensively annotated. Verification with CRESS is performed only after a further automated stage to annotate a LOTOS specification for CADP. This requires a tool that knows about standard LOTOS data types as well as the data types that CRESS generates.

CADP does not allow parameterised data types, so they must be instantiated first. The authors developed a tool to 'flatten' and annotate data types: all data types are collapsed into one, and CADP pragmas are created. CADP also does not support infinite sorts. Annotations in the form of special LOTOS comments are therefore added to a specification prior to verification, e.g. to identify constructor operations and external implementations of data types.

CADP can verify a LOTOS specification through model checking. Abstract data types with infinite values have to be limited to a finite range for verification. Most data types in grid and web services have finite (although possibly large) ranges whose size may depend on the programming language or platform. Several CRESS library data types such as *Number* have an infinite range.

In previous work, finite ranges were manually specified for CRESS data types (e.g. *Char*, *Number*, *Text*). In the work reported here, the automated annotation tool also deals with restricting ranges. C implementation skeletons are created automatically for user-defined data types (e.g. record structures in CRESS). For the occupation translation example, C skeleton files are created automatically for CRESS types like *schemes* and *mapping*. As it happens, this particular example does not need any special implementations for data types – CADP supplies default implementations. However, specific implementations can be created manually to replace the default ones.

Roughly speaking, each CRESS diagram node corresponds to a LOTOS process. A LOTOS process communicates using events at gates. Processes synchronise their communications at gates, which may be selectively hidden from external view. Processes may run independently in parallel or may synchronise on specific gates.

Factory and *Mapper* are normal partners, and are instantiated inside the *Allocator* where they are used. The *Allocator* is actually instantiated twice: once inside *Lookup*, where it is used, and again at the global level. This is because *Allocator* is a composite service that can be used in its own right. A *Resource* partner implicitly represents the set of dynamic resources that may be allocated by the *Factory*. In implementation terms, this is called the 'resource home'. In CRESS this is a 'phantom partner', and is instantiated at the global level for use by all services.

3.5 Verifying Service Compositions

Verification allows general properties to be checked, whereas validation can check only specific cases (though these are usually selected to be the critical ones). Model check-

ing requires a finite (though possibly large) state space, and so will not be practicable in some cases. Validation can deal with very large or infinite state spaces. The two techniques are therefore complementary, and help to ensure that the methodology for service development is both rigorous and practical.

Service properties are verified using the notation and tool called CLOVE (CRESS Language-Oriented Verification Environment). This supports the high-level formulation of properties, and provides a simple way of using the actual verification tools. To some extent, CLOVE is oriented towards the needs of verifying grid or web services. Verification is normally undertaken only by specialists. To fit in with the pragmatic aims of CRESS, CLOVE is designed for use by those with limited knowledge of formal methods. For example, common properties of services are automatically checked, and property templates are also supported. This allows the domain specialist the verify correctness of service descriptions.

The specification patterns repository (*patterns.projects.cis.ksu.edu*) builds on the fact that verification properties are often common across many application domains. This makes it possible to develop template properties that can be supported by different formal methods [6]. CLOVE supports this approach by embedding and extending these properties, using the LOTOS representations developed by Mateescu (*www.inrialpes.fr/vasy/cadp/resources/evaluator/rafmc.html*). In addition, CLOVE supports common properties such as freedom from deadlock and livelock, as well as specialised properties that are appropriate for services.

As examples, the following properties are desirable for the occupational translation service in figures 1 and 2:

- The service should always be available, i.e. free from deadlocks (a safety property).
- If the service receives a request, it must able to accept a new request at a future point (a liveness property).
- For correct service requests, the client should receive the translated job title or a fault due to partner failure.
- For incorrect service requests (an unknown job title or classification scheme), the service must throw a fault to the client.
- If no translation resource exists, the service must throw a fault to the client.

As a concrete example, the following CLOVE property deals with service requests and responses. A request to translate an occupation must always ('global') obtain the translated occupation code, or else a lookup fault with a string message. '?' means any value of the given type. If this property does not hold of the service description, the cause of the failure is analysed.

```
property(General_Response,
  response(global,
    signal(lookup.job.translate,?schemes),
    or(
      signal(lookup.job.translate,?codes),
      signal(lookup.job.translate,lookupError,?string))))
```

The *Nurse_Translation* test in section 3.3 checks only one translation. The following CLOVE property asserts that translating job title 'nurse' into the SOC2000 and SIC92 classifications should always yield the correct result.

```
property(Nurse_ Response,
  response(global,
    signal(lookup.job.translate,schemes('Nurse,'SOC2000,'SIC92)),
    signal(lookup.job.translate,codes('3211,'95.14))))
```

Common service properties are automatically verified without having to be specified explicitly. In addition, service-specific properties like the above can be formulated by the developer. The CLOVE notation is intended to be more accessible than the underlying formalism (μ-calculus). CLOVE is also designed to be similar to MUSTARD, allowing the developer to verify and validate services in a similar way. Although these example properties are simple, they are typical of service verification practice. CLOVE also supports other types of property, e.g. for safety or liveness (reachability).

Behind the scenes, CLOVE automatically translates the properties into μ-calculus [5] – a temporal logic that allows branching-time properties to be checked. CLOVE then invokes CADP to carry out property verification. The goal for verifying CRESS service descriptions is to make this as 'push button' as possible, especially since the service developer may not be a computing specialist. In fact it is possible for the developer to create composed services without leaving the CRESS diagram editor. A service composition can be described graphically, validated, verified, implemented and deployed from within this graphical tool.

The CADP tools used for verification are CAESAR (behaviour compiler), CAESAR.ADT (data type compiler) and Evaluator. CAESAR.ADT generates a C header file from the LOTOS specification, including references to the C skeleton files generated by CRESS. CAESAR is then used to generate a BCG (Boundary Components Graph) for the specification. The Evaluator tool verifies properties of a specification in LOTOS or BCG form. Verification steps are defined by an automated script written in SVL (Script Verification Language). Desirable properties include deadlock freedom, consistency of service behaviour, and reachability of service states.

The CRESS specification generated from figures 1 and 2 was verified against these properties after some corrections. For example, the original description had deadlocks due to an error in dealing with requests with an invalid occupational scheme. As a result the system, could not proceed and did not respond to the client request.

4 Implementing and Deploying Composed Grid Services

The main emphasis of this paper is on formal aspects, so the automated implementation will be described only briefly. The *same* CRESS description as used for specification is automatically implemented through translation into BPEL/WSDL and is packaged for deployment. Services that are part of the composition have their interfaces and data types generated in WSDL and XSD respectively. The BPEL, WSDL interface, WSDL catalogue, deployment descriptor and common definitions are automatically generated for the composite service and its partners. CRESS generates outline implementations of partners that are completed manually for use in the final implementation.

Service orchestration (for *Lookup* and *Allocator* in this example) is performed by the ActiveBPEL engine (*www.activebpel.org*). The composed service is automatically created and deployed as a BPEL archive. If orchestration makes use of partner *web*

services, these are also deployed in ActiveBPEL. More typically, the partners are *grid* services (*Factory* and *Mapper* here). These are automatically packaged and deployed as grid service archives using the Globus Toolkit (*www.globus.org*).

ActiveBPEL, Globus Toolkit, the composite service and its partners can all run on one system, though more typically they are distributed. This is defined by a CRESS configuration diagram (not shown here) that defines the locations and deployment characteristics of all services. The running implementation can then be validated again using the *same* MUSTARD tests as were used for the specification (section 3.3). In particular, this evaluates non-functional properties such as performance, dependability and reliability. This time, MUSTARD is translated into an intermediate form that is suitable for use in testing an implementation. MINT (MUSTARD interpreter) executes these tests in a similar kind of way as LOLA does for LOTOS. However, MINT has additional capabilities for evaluating an implementation. For example, it can perform stress testing by running many tests concurrently or sequentially to check implementation performance.

5 Conclusion

The CRESS methodology for composing services has now been rounded out to handle all the key characteristics of grid services. For example, service resources, EndPoint References and dynamic partner assignments are now fully handled in both the specification and implementation phases of CRESS. Formal validation of the generated LOTOS specifications was already possible using MUSTARD. New work has added automatic verification of desirable specification properties, allowing properties of a composite service to be proven in general. Automatic validation of the generated implementation is also now possible, using MINT to check (non-)functional characteristics.

Verification through model checking requires a finite state space. This is a reasonable restriction since the data types of an actual grid service implementation are finite. Though the state space can grow very large, the size of it can be constrained by using subsets of data values and by choosing significant values for verification. However, validation still has a useful role. For example, it can be used with infinite state spaces, can check specific interesting cases, and can be used for stress-testing the implementation.

Support for automated formal analysis will be further improved. It is planned to allow data types to be annotated in CRESS with regard to useful ranges and interesting values. A possible approach here is to use PCL (Parameter Constraint Language [17]) to specify significant values for validation and verification. This would allow MUSTARD test cases to be automatically defined. A rigorous methodology for developing composite grid services has been presented. This uses an accessible graphical notation and a high degree of automation to make it attractive to industry. An occupational classification service has been used to explain how interactions with dynamic resources and dynamic partners are supported by CRESS. Abstract CRESS descriptions are automatically translated into LOTOS for formal verification of desirable properties and for formal validation of significant test cases. These are almost 'push button' procedures. The CRESS descriptions can then be automatically translated into implementations with confidence. The same MUSTARD tests can again be used to check the characteristics of these implementations.

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).

References

1. A. Arkin *et al.*, editors. *Web Services Business Process Execution Language*. Version 2.0. Organization for The Advancement of Structured Information Standards, Apr. 2007.
2. L. Baresi *et al.* Validation of web service compositions. *Software*, 1(6):219–232, Dec. 2007.
3. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14(1):25–59, Jan. 1988.
4. A. Chirichiello and G. Salaün. Encoding abstract descriptions into executable web services: Towards a formal development. In *Proc. Web Intelligence 2005*. IEEE, Dec. 2005.
5. E. M. Clark, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
6. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. 21st Int. Conf. on Software Engineering*, pp. 411–420, 1999.
7. A. Ferrara. Web services: A process algebra approach. In *Proc. 2nd Int. Conf. on Service-Oriented Computing*, pp. 242–251. ACM Press, Nov. 2004.
8. H. Foster. *A Rigorous Approach to Engineering Web Service Compositions*. PhD thesis, Imperial College, London, Jan. 2006.
9. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. 13th. Int. World Wide Web Conf.*, pp. 621–630. ACM Press, May 2004.
10. 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.
11. S. Graham *et al.*, editors. *Web Services Resource*. Version 1.2. Organization for The Advancement of Structured Information Standards, Apr. 2006.
12. ISO/IEC. *LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807, 1989.
13. T. Oinn *et al.*. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
14. C. Pautasso. JOpera: An agile environment for web service composition with visual unit testing and refactoring. In *Proc. IEEE Symp. on Visual Languages and Human Centric Computing*. IEEE, Nov. 2005.
15. S. Pavón Gomez, D. Larrabeiti, and G. Rabay Filho. LOLA user manual (version 3R6). Technical report, Polytechnic University of Madrid, Feb. 1995.
16. A. Slomiski. On using BPEL extensibility to implement OGSI and WSRF grid workflows. In *Proc. Global Grid Forum 10*, Berlin, Mar. 2005.
17. K. J. Turner. Test generation for radiotherapy accelerators. *Software Tools for Technology Transfer*, 7(4):361–375, Aug. 2005.
18. K. J. Turner. Validating feature-based specifications. *Software Practice and Experience*, 36(10):999–1027, Aug. 2006.
19. B. Wassermann *et al.*. Sedna: A BPEL-based environment for visual scientific workflow modelling. In *Workflows for E-Science*, pp. 428–449. Springer, 2007.
20. W3C. *Web Services Addressing (WS-Addressing)*. World Wide Web Consortium, May 2006.
21. J. Yu *et al.*. Using temporal business rules to synthesize service composition process models. In *Proc. 1st Int. Workshop on Architectures, Concepts and Technologies for Service Oriented Computing*, pp. 86–95, INSTICC Press, July 2007.
22. M. Zager. SOA/web services – Business process orchestration with BPEL. *http://webservices.sys-con.com/read/155631_1.htm*, Oct. 2008.