Model Checking Service Compositions under Resource Constraints

Howard Foster[1], Wolfgang Emmerich[2],
Jeff Kramer[1], Jeff Magee[1], David Rosenblum[2] and Sebastian Uchitel[1]
London Software Systems
[1] Dept. of Computing, Imperial College London
180 Queen's Gate, London SW7 2BZ, UK
[2] Dept. of Computer Science, University College London
Gower Street, London WC1E 6BT, UK
[1] {hf1,jk,jnm,su2@doc.ic.ac.uk} [2] {w.emmerich,d.rosenblum@cs.ucl.ac.uk}

ABSTRACT

When enacting a web service orchestration defined using the Business Process Execution Language (BPEL) we observed various safety property violations. This surprised us considerably as we had previously established that the orchestration was free of such property violations using existing BPEL model checking techniques. In this paper, we describe the origins of these violations. They result from a combination of design and deployment decisions, which include the distribution of services across hosts, the choice of synchronisation primitives in the process and the threading configuration of the servlet container that hosts the orchestrated web services. This leads us to conclude that model checking approaches that ignore resource constraints of the deployment environment are insufficient to establish safety and liveness properties of service orchestrations specifically, and distributed systems more generally. We show how model checking can take execution resource constraints into account. We evaluate the approach by applying it to the above application and are able to demonstrate that a change in allocation of services to hosts is indeed safe, a result that we are able to confirm experimentally in the deployed system. The approach is supported by a tool suite, known as WS-Engineer, providing automated process translation, architecture and model-checking views.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Validation

General Terms

Design, Languages, Verification

Keywords

Web Services, Resource Modelling, Validation, BPEL4WS

1. INTRODUCTION

The Business Process Execution Language (BPEL4WS) [2] has become the de-facto industry standard for orchestrating web service invocations. Such orchestrations effectively compose several web service invocations. This combination can be given its own interface in a Web Service Description Language and then becomes a web service in its own right. Thus, we expect BPEL to take a central role in any service oriented architecture.

BPEL supports a number of synchronisation primitives to invoke other web services. Such invocations can be done in a synchronous, deferred synchronous or asynchronous manner. BPEL also includes a primitive to determine the concurrent execution of a block of statements. In order to avoid concurrency problems, such as lost updates or inconsistent analyses, the language supports locking primitives so that variables that maintain state can be accessed in mutual exclusion. The combination of these primitives mean that BPEL orchestrations that are not written carefully may deadlock or exhibit other safety or liveness property violations.

In [8], we have shown how such safety property violations of BPEL orchestrations can be detected. The approach rests on a translation of BPEL processes into Finite State Processes [16] and subsequently into labelled transition systems, which can then be model checked using, for example, our Labelled Transition System Analyser to establish presence or absence of deadlocks or other safety or liveness properties. We have used this approach to establish deadlock-freedom of a web service orchestration that implements a scientific workflow to search for polymorphic crystal structures of complex organic molecules described in detail in [7].

Following implementation and deployment of these processes in the open source ActiveBPEL process engine, we were very surprised to find that every so often the BPEL process would not terminate for no obvious reason. The BPEL process that implements this scientific workflow is computationally quite demanding. It spawns several thousand sub-processes and has tens of thousands of web service invocations. It turned out that a complex interaction between the allocation of web services to servlet containers, the size of thread pools of the BPEL engine and the threading policy adopted by the host server that processed invocations and replies caused deadlocks of a hierarchy of processes that was itself deadlock-free.

The main contribution of this paper is two-fold. We firstly give a detailed account how a process that provably satisfies safety and liveness properties can still be unsafe, and can deadlock or livelock in a resource constrained environment. We deduce from this insight that resource constraints need to be considered when reasoning about safety or liveness properties of web service orchestrations. Secondly, we describe a technique for model checking service compositions under resource constraints and evaluate this approach using the polymorph case study.

In Section 2, we provide a background to service orchestration, resource management and discuss related work. In Section 3, we describe the case study that raised the issues we are addressing in our work. In Section 4, we discuss a method of modelling processes under resource constraints and our abstraction of the problem domain. Section 5 illustrates a validation of the models and suggests resolution alternatives, whilst section 6 concludes this paper with a summary and an indication of our future work in this area.

2. BACKGROUND

A Web Services Architecture (WS-A) is a set of conceptual elements which define a common set of standards between interoperating components, running on different platforms and/or frameworks. The W3C definition of WS-A [3] includes a note that there is no restriction on how these services are implemented or combined to provide more complex compositions. The web services standards stack (including specifications for data, interface, service description, orchestration and choreography) is evolving to support various aspects of creating a web services architecture, yet there remains an ambitious task of building systems on such architectures and it is closely aligned with the capabilities of technology support. Amongst these standards is the Business Process Execution Language for Web Services (BPEL) [2], for service composition and orchestration. BPEL has gained significant support by both industry and academia. The notation in this specification aims to provide a standard description of processes which interact with a number of service partners, and therefore research has explored modelling these orchestrations to provide both design and implementation support in building complex processes.

Other work in the analysis of web service orchestrations includes mapping BPEL to Petri Nets [19], for control logic checking of BPEL and describes addition analysis for isolating *"redundant messages"* that are not necessary if a certain activity has been performed. This appears to be an advantage for efficiency in composition processing although the level of benefit of this ability is difficult to measure. Alternatively, [12] uses Petri net-based models to represent web service composition flows independently of a particular specification. In this work they define a *"web service algebra"* (in BNF-like notation). There is a little coverage however, of how this maps to current standard web service composition languages (such as BPEL4WS or WS-CDL). In [22] web service compositions are described in the Language of Temporal Ordering Specifications (LOTOS). The authors extend a mapping between the algebra and BPEL4WS by providing rules for a partial two-way process, but again there is no easily accessible mechanism for web service engineers to perform this analysis. Fu [10] provides an analysis tool based upon translation of BPEL4WS descriptions to Promela and analysed using the SPIN tool. They also apply limited XPath expressions for state variable access analysis.

There has been little published on the design of web services architectures, compositions and resource management. Closely related to these topics however, are the works published on component resource management and evaluation of their resource usage. In [5] the authors present a formalism for specifying component interfaces that expose requirements on limited resources. Their formalism permits an algorithmic check if a composition of components exceeds the available resources (e.g. network buffer overflows) and suggests an optimal configuration. [24] models the states of a set of printer components which uses memory resources for pages, fonts etc. Using probabilistic calculus their approach also permits compositional resource usage reasoning. Alternatively, [13] uses Labelled Transition Systems (LTSs) in a theoretical framework capable of conducting analyses of discrete-event processes that interact through shared, discrete resources. The approach defines a simulation language (called DEMO), which mimics the acquisition and release of available resources in both synchronous and asynchronous processes.

To date, existing enterprise application component server technology, such as the Java 2 Enterprise Environment (J2EE) compliant server framework and Microsoft .NET framework, have extended capabilities to host components as services. The deployment of web services into a web service container means that the resource demands of a web service during an invocation are controlled by the container in cooperation with the underlying operating system. These resources include memory, file descriptors, database connections and threads. Containers typically terminate if they exhaust memory or if the number of file descriptors that can be open at any one time is exceeded. The behaviour for the management of database connections and threads is different though and a container would have a fixed and configurable pool of these resources, allocate resources from these pools to web services and if this pool is exhausted it would force web service executions to wait until a resource becomes available again.

Web service compositions using BPEL are executed by a specialist container, sometimes called a BPEL engine or a BPEL run-time environment. These containers will again use resources. BPEL engines will, for example upon receiving a SOAP message to start a BPEL process, instantiate this process and execute it in a separate thread concurrently with other ongoing BPEL processes. Again BPEL engines typically have configurable database connections and thread pools and they would delay the start of a BPEL process until they can assign a thread from a pool. Both Web service and BPEL containers typically map these threads efficiently to a set of operating system threads. The amount of operating system threads however, is finite due to the finite amount of memory required to handle the stack segment of the thread. Administrators must therefore carefully configure the thread pools to avoid exhaustion of the operating system resources.

3. SAFETY AND LIVENESS VIOLATIONS DUE TO RESOURCE CONSTRAINTS

3.1 Case Study

We have modelled a sizable theoretical chemistry workflow using BPEL. The workflow is used to predict organic crystal structures from the chemical diagram. In order to

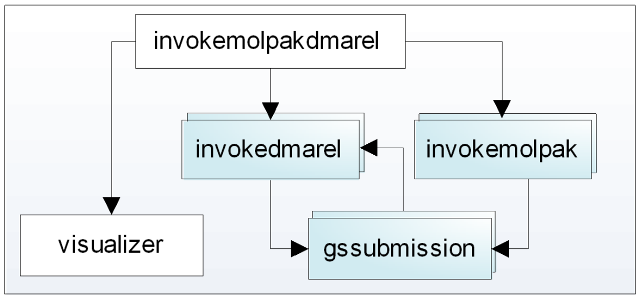


Figure 1: Case Study Workflow Hierarchy

structure the complex workflow, we have decomposed it into a number of component workflows, which each expose a web services interface. This decomposition is shown in Figure 1. Arrows in the diagram indicate the synchronous invocation of the web service that is implemented by a BPEL process. Some client component invokes the root workflow, which then invokes up to 38 InvokeMolpak workflows in parallel. These, in turn, invoke the GSSubmission workflow in order to submit jobs to a grid resource manager. The GSSubmission workflow uses the GridSAM web service [14] to submit a job and then polls the GridSAM monitoring service until the results are returned. Once these jobs are completed a further 200 parallel InvokeDmarel workflows are instantiated that then in turn each again submit a job to the GSSubmission workflow, causing the same process of job submission and polling for results. The workflow and its orchestration in BPEL are defined in detail in [7].

3.2 Naive deployment

We deployed all BPEL processes in the same instance of the ActiveBPEL workflow engine and hosted the engine in a Servlet container on a powerful Linux server. We hosted the GridSAM web service in the same Servlet container. As we had previously established deadlock freedom using the approach and tools described in [8], we were considerably surprised to find that the process would deadlock on a regular basis and it took us quite some time to understand the cause of these deadlocks.

The deadlocks were caused by a complex interplay between allocation of web services and BPEL processes onto execution hosts, the synchronisation behaviour of the web service and BPEL processes on the one hand and the threading behaviour and thread pool configuration of both the servlet engine and the BPEL engine on the other hand. We had configured the Linux operating system to provide 2048 threads. We allocated 750 threads to the BPEL engine and another 900 threads to the Servlet container. The concurrency constraints were such that easily more than 750 process instances could be active at any one time. Under those circumstances the ActiveBPEL engine enqueues the process instances to wait until a thread from the engine's thread pool becomes available. Similar behaviour occurs with the processing of SOAP messages by the Axis servlet that is contained in Tomcat. Tomcat allocates a thread to process for each incoming SOAP message and the thread is released into the thread pool once the matching SOAP reply message is set. In our workflow it is again possible for the pool of 900 threads to be exhausted and in that case, the SOAP messages get queued in the TCP stream until threads become available to process them. Taken together, this behaviour means that there is a chance for deadlocks because when no threads are available and GSSubmission workflows are polling for results using the monitoring service then these invocations will not return, which means that the respective GSSubmission workflows will not terminate, which also means that no further threads will be released and the system enters a state of deadlock.

3.3 Key insight

The deadlocks described above, or more generally safety and liveness property violations, can be caused by resource allocation decisions of the underlying distribution middleware. To ascertain safety and liveness of distributed systems it is therefore necessary to integrate models of the abstract process behaviour with models that describe the policies used for resource allocation policies of the underlying middleware. We also note that fundamentally different technology stacks are competing with each other for operating system resources and these need to be reconciled with each other, too.

4. RESOURCE AWARE PROCESS MODELLING

The activity of providing resource aware process modelling is formed from considering the architecture, the processes and the resource allocation. To concisely build models of these concepts, we use the Finite State Process (FSP) notation developed by Jeff Magee and Jeff Kramer of the Distributed Software Engineering Group at Imperial College London. We begin by briefly describing this notation.

4.1 FSP, LTS and Behaviour Models

Our approach uses an intermediate representation to undertake analysis of web service compositions and choreography. The FSP notation [16, 17] is designed to be easily machine readable, and thus provides a preferred language to specify abstract processes. FSP is a textual notation (technically a process calculus) for concisely describing and reasoning about concurrent programs. The constructed FSP can be used to model the exact transition of workflow processes through a modelling tool such as the Labelled Transition System Analyzer (LTSA) [23], which provides a compilation of an FSP into a state machine and provides a resulting Labelled Transition System (LTS). FSP supports a range of operators to define a process model representation. A summary of the operators for FSP is given as follows. **Action** prefix "->": (x->P) describes a process that initially engages in the action x and then behaves as described by the auxiliary process P; **Choice** "|": (x->P |y->Q) describes a process which initially engages in either x or y, and whose subsequent behaviour is described by auxiliary processes P or Q, respectively; **Recursion**: the behaviour of a process may be defined in terms of itself, in order to express repetition; **Sequential composition** ";": (P;Q) where P is a process with an END state, describes a process that behaves as P and when it reaches the END state of P starts behaving as the auxiliary process Q; **Parallel composition** "||": (P ||Q) describes the parallel composition of processes P and Q; **Trace equivalence minimisation** "deterministic": deterministic P describes the minimal trace equivalent process to P. If no terminating traces are proper prefixes of other traces, then it also preserves END states; **Weak semantic equivalence minimisation** "minimal": minimal

P describes the minimal weak semantic equivalent process to P; **Relabelling** "/": Re-labelling is applied to a process to change the names of action labels. The general form of re-labelling is / {newlabel/oldlabel}; **Hiding** "\": When applied to a process P, the hiding operator \{action1, actionx} removes the action names from the alphabet of P and makes these concealed actions "silent".

4.2 Modelling the architecture

The technology architecture we have used in our case study (described in section 3.1) is that based upon the Apache Tomcat 5.5 Servlet/JSP Container [23] and the ActiveBPEL Engine [1] which runs on the J2EE environment. The server is used to host two BPEL processes, and other web services, to carry out the intensive processing of molecule breakdown analysis. Both Tomcat and ActiveBPEL provide a "maxThreads" attribute to limit the number of threads available if requested, although it is a general characteristic of these engines types that they provide unlimited threads to support a growing need as client requests are made to the services. Such technology implementations require an architecture to model how certain operations acquire or release resources. In the case of ActiveBPEL we have found that the architecture described in the ActiveBPEL Engine documentation required further analysis of the engine logs to determine when threads are acquired or released. Furthermore, different technologies built to support the service orchestration implementations may carry out the specification with their own strategy, which effects the way in which resource management is utilised. Our model of resource management is focused on the thread pool utilisation and as such, is sufficiently generic enough to represent varied differences in architecture configuration.

A model of the architecture can describe the characteristics of the host server and orchestration as follows. There are a number of Architecture Description Languages (ADLs) we could use to describe this architecture, including DARWIN [15], ACME/ADML [11], and UML [18]. We selected an open format in the form of the eXtensible ADL (xADL) [6] as it is general, maps to many ADLs, and has good tool support. xADL 2.0 supports run-time and design time modeling, architecture configuration management and model-based system instantiation. Additionally, xADL 2.0 has a set of extensible infrastructure tools that support the creation, manipulation, and sharing of xADL 2.0 documents. An architecture model produced in ArchStudio 4 (an architecture meta-modeling environment built upon the xADL 2.0 specification) is illustrated in Figure 2. The underlying document for this model is a xADL 2.0 XML description which can be used later in our work to provide a source for generating the server architecture process model. The process of abstracting these elements from the underlying document is quite detailed, and the focus of this work is on the analysis of the behaviour exhibited, therefore we only provide a summary here of how the types are mapped to process elements. In our illustration the two BPEL process components are connected to a servlet component through "hosted" connector interfaces. The servlet component itself is connected to a threadpool component through a "manages" connector interface, and the threadpool component has an interface which includes a property of "Size=10" to demonstrate a threadpool size limitation. The architecture description illustrated supports an abstraction of architec-

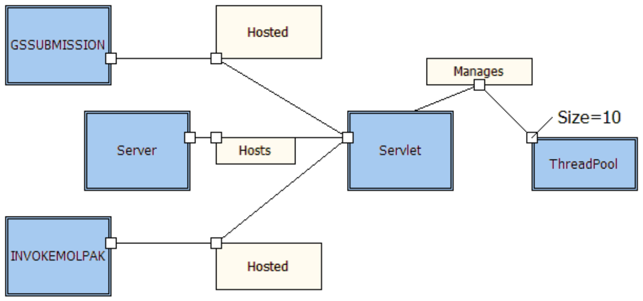


Figure 2: Server, Process and Resource Architecture

ture elements which are necessary for our analysis, i.e. components, connectors and links. As xADL 2.0 is extendable, we foresee that further attributes of the architecture supported in analysis can be added at a later stage.

We now begin our process modelling by considering the resource management aspects of the architecture.

4.3 Modelling resource management

A resource, or more specifically a system resource, is any physical or virtual component of limited availability within a computer system. Every device connected to a computer system is a resource. Every internal system component is a resource. Process related resources are typically defined in one of three groups [20], that of Processor (thread pools, priority mechanisms and intraprocess mutexes), Communication resources (protocol properties, connections etc) and Memory (buffering requests in queues and bounding the size of a thread pool). One such resource that is commonly used with multiple processes and interactions is that of a shared thread pool. We model the shared thread pool in FSP as a sequence of processes which "get" or "put" a resource from a container (Figure 3). The shared thread pool (TPOOL) is a container for N number of threads (in the architecture example given previously this was the "SIZE=10" property of the threadpool component), and represents the service orchestration server technology stack for allocating and releasing threads as required by the orchestration processes. When a process is composed with this thread pool, those interactions which acquire a thread (represented by the first conditional statement of "(t>0) get ->TPOOL[t-1]") decrease the pool by one thread if there exists unallocated threads in the pool. Alternatively a completed interaction may free a thread which is represented by the statement "put ->TPOOL[t+1]", adding a thread back to the pool.

4.4 Modelling technology stack interactions

We now have a model of a generic shared thread pool, and an architectural description with allocations of the processes and resource pool to server instances. We must now model the behaviour of the processes (hosted on the server environment) and link these with the servlet and resource process models.

4.4.1 *Orchestration Behaviour and Resources*

Figure 4 illustrates how a simple BPEL process acquires and releases threads as the process is carried out, instigated by an initial request message. A host servlet (in which the process is contained) acts as a generic provider for listening

```
TPOOL(N=10) = TPOOL[N],
TPOOL[t:0..N] =
(when (t>0) get -> TPOOL[t-1]
|put -> TPOOL[t+1]).
```

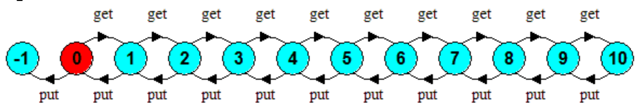


Figure 3: FSP and LTS for a shared Thread Pool model.

and managing to service requests. The servlet also requires a thread, and is allocated one when the server starts.

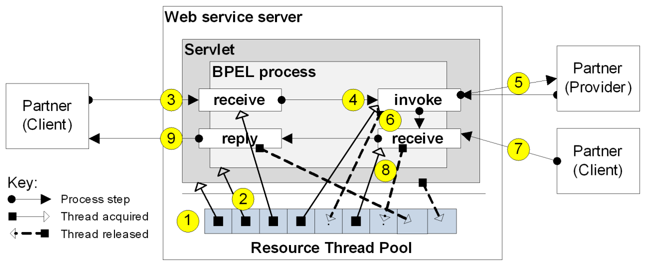


Figure 4: Web Service Server, Servlet and BPEL Process Thread Acquisition and Releases.

A scenario of thread resource management for host, container and process (depicted in figure 4) is a follows. As discussed, initially the servlet acquires a thread for its own purposes (1). On a new request, the servlet attempts to direct the message to the appropriate point in an existing BPEL process, however, in the event there is no running process then a new process is created and assigned a thread (2). The "receive" construct of BPEL usually provides an entry point in to the process for a particular service operation request and is also assigned a thread (3). Immediately following the receive is an invocation (4), defined by a "invoke" construct to another partner of the process. The invocation is assigned a further thread (5). In this case the invocation is synchronous and the invocation awaits a reply from the partner. When a reply is given, the assigned invocation thread is released back to the pool (6). As the orchestration is of a composition of service interactions, the process expects to receive a further request from another partner (7). When this occurs another "receive" requires an additional thread (8) and the message is accepted. The process does not reply to this request (the servlet responds with a standard confirmation) and therefore the thread is released back to the pool. A "reply" statement completes the process, and consequently releases the thread from the initial "receive" construct (9). On process completion the process handler also frees its allocated thread. The remaining thread allocated for the servlet, remains allocated for the lifetime of the servlet instance on the host server.

4.4.2 Mapping Orchestration Activities

In more complex orchestrations, or where there are multiple processes hosted in a servlet, the assessment of resources required becomes increasingly difficult to estimate. A process model however, can provide a formal specification of the interactions and can be composed with resource models to detect where possible deadlocks may occur given a

```
// --- partial FSP for process transitions ---
GSSUB_SUBMITJOB = (GSSUB_
     jobsubmissionpartner_invoke_submitjob->END).
GSSUB_SUBMITJOB_REPLY = (GSSUB_
     jobsubmissionpartner_reply_submitjob ->END).
WAIT1_PT120S = (wait1_pt120s->END).
GSSUB_SEQ2 = (GSSUB_jobmonitoringpartner_
     invoke_getjobstatus->END).
GSSUB_REPLY = (GSSUB_jobmonitoringpartner_
     reply_getjobstatus ->END).
GSSUB_SEQ = GSSUB_SEQ2; GSSUB_REPLY; END.
GSSUB_SEQUENCE= WAIT1_PT120S; GSSUB_SEQ2; END.
||WHILE2 = (GSSUB_SEQUENCE).
// --- create and terminate additions ---
GSSUB_CREATEINSTANCE = (
    createInstance_gssubmission->END).
GSSUB_TERMINATEINSTANCE = (
    terminateInstance_gssubmission->END).
```

Figure 5: Partial FSP for GSSUBMISSION BPEL Process.

limited number of resources available. In [8] we described a mapping of BPEL processes to the FSP notation, and here we extend this to support mapping interactions to resource allocation activities. Each of the BPEL interaction activities of Invoke (synchronous), Invoke (asynchronous), Receive and Reply, are represented in FSP as an activity label. The format of this label is in a template of "process_partner_operation_activity", where "process" is the name of the BPEL process, "partner" is the name of the partner role for which the interaction is taking place, "operation" is the method of the activity (e.g. newOrder, sellBook etc), and "activity" is the interaction form (i.e. invoke, receive or reply). Using these rules and as an example, we translate each of the BPEL processes to FSP. This activity maps each of the BPEL constructs to FSP operators and activity labels. A sample of the FSP generated is listed in Figure 5 along with a LTS Model (Figure 6) for the process (short names have been used for presentation).

The key additions to our mapping for resource allocation are as follows. Firstly we add an action for the create and terminate process instances. These are represented simply as GSSUB_CREATEINSTANCE and GSSUB_TERMINATEINSTANCE in the example. In the BPEL specification, a process may be instantiated by containing at least one "start activity". This may either be designated on a "receive" activity or a "pick" (which resembles the switch..case statement in traditional structured programming languages) through the use of a "createInstance" attribute. There is no restriction for the number of activities which may create an instance of a process, and there are further semantics for how these correlate on a given process. Therefore, a createInstance action can occur in multiple activities, but only one may actually create an instance of a process. In our current mapping capability, we assume that one activity will be designated to create an instance of a process. In the example given for the GSSUBMISSION process, this is immediately following the "client_GSSUB_receive_runGSSUB" activity.

4.4.3 Linking Service Orchestrations

In our case study, the INVOKEDMAREL orchestration calls the GSSUBMISSION orchestration to request a "RungSubmission" operation. In this way, the two orchestrations are interacting and our modelling approach needs to reflect this in order to evaluate interactions accurately. We leverage

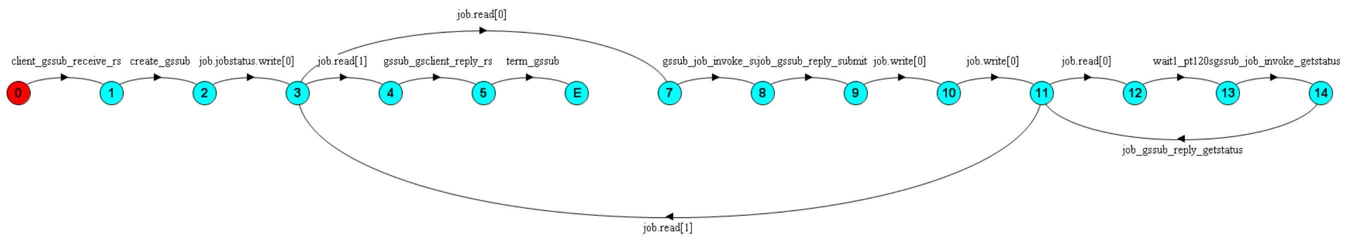


Figure 6: LTS Model for GSSUBMISSION BPEL Process.

our previously reported work on "compatibility verification" in [9] for Web Service choreography. To model the interaction between processes we require a process link between the "invoke", "receive" and "reply" actions of the BPEL4WS processes and a model of how these interactions are buffered across the system. Partner process activity interactions can be represented in FSP by using the notion of a connector, which encapsulates the interaction between the components of the service architecture. This is implemented in FSP as a monitor, allowing us to combine the concepts of information hiding and synchronization. "Rendezvous" (Request-Reply) invocations are specified in BPEL4WS with the "invoke" construct, with both input and output container attributes. To model these types of interactions, we use a generic synchronous port model for each process port. Synchronous invocations specified with the "invoke" construct and only an input container attribute declare an interaction on a request only basis (there is no reply expected). With both of these invocation model types, the connection interaction for invoke activities in BPEL4WS can be modeled effectively using transition links for send, receive and reply processes in FSP. The task of modeling the invocation process and port is completed by using the re-labeling feature of FSP linking the appropriate activities between process and port. An example of this port connector model is listed in FSP for the INVOKEDMAREL and GSSUBMISSION interactions as follows;

```
// --- port reply model ---
INVOKEDMAREL_GSSUBMISSIONPARTNER_RUNGSSUBMISSION
 _PORT_REPLY = (gssub_gsclient_reply_rungssub-
  mission->gssubpartner_invokedmarel_reply_rungs-
  submission->INVOKEDMAREL_GSSUBMISSIONPARTNER_
  RUNGSSUBMISSION_PORT_REPLY).
// --- port invoke model ---
INVOKEDMAREL_GSSUBMISSIONPARTNER_RUNGSSUBMISSION
  _PORT_INVOKE = (invokedmarel_gssubpartner_
  invoke_rungssubmission->gssubmissionclient_
  gssubmission_receive_rungssubmission->
  INVOKEDMAREL_GSSUBMISSIONPARTNER_
  RUNGSSUBMISSION_PORT_INVOKE).
```

4.4.4 *Linking Orchestration and Resource Management*

Now that we have provided methods for producing a resource management process model for thread pools, a translation of BPEL to an LTS process model and interaction linking between processes, we can further define a mapping between these models and include a model of the acquisition and release of system thread resources. Specifically for each orchestration architecture, we scan each of the orchestration process interactions and gather those which are resource-operator activities. In the case of the BPEL notation, these are receive, invoke and reply. A pseudocode algorithm for undertaking this is given as follows.

```
For each Server Resource Pool
  For each BPEL process model assigned
    Gather the process interaction labels
    For each interaction label
      Determine operation of interaction
      if operation is resource-operator then
        add interaction label to resource-operator-list
      add resource-operator-list to pool-operator-list
    For each pool-operator-list item
      if item is receive or invoke operator then
        relabel item with pool get operator
      otherwise
        relabel item with pool put operator
      add item-relabel to pool-resource-map
  add createInstance to pool-resource-map
  add terminateInstance to pool-resource-map
  generate pool-resource-map process
END.
```

Additionally, we add a resource and activity mapping for the createInstance and terminateInstance activities. These are added as a resource "get" and "put" respectively. The last activity stated is that of generating a pool-resource-map process and is the result of the linking algorithm. To generate this, we need to define a process stub for each combination of orchestrations sharing the server resource pool and represent that a number of instances of these processes can exist at a given time. For example, in our case study we have two BPEL orchestrations "GSSUBMISSION" and "INVOKEDMAREL". Both of these utilise the pool resources and undertaking the algorithm provides us with a process illustrated as FSP in Figure 7.

Note that each of the "get" and "put" relabelling is assigned to a given process "proc[P]". This represents one or many instances of the processes on the server. "proc" is process label as a set of P interacting processes. So for example, "proc[2]" would model two instances of both the GSSUBMISSION and INVOKEDMAREL orchestrations interacting. In this way, we can simulate a number of clients requesting the orchestrations (where an initial request is made to start either of the orchestrations).

4.4.5 *A Complete Model for Validation*

A complete model is represented in FSP in figure 8. The FSP for the entire process is detailed, and therefore a summary illustrates the individual elements that have been listed previously with an architecture model representing a composition of these elements. Firstly, we specify two compositions for the orchestrations

```
// --- pool mapping to acquire a resource ---
 /{proc[P].{createInstance_gssubmission,
 gssubmissionclient_gssubmission_receive_rungs-
 submission,gssubmission_jobsubmissionpartner_
 invoke_submitjob,gssubmission_jobmonitoring
 partner_invoke_getjobstatus,createInstance_
 invokedmarel,invokedmarelclient_invokedmarel_
 receive_invokedmarel,invokedmarel_molpak2cml-
 partner_invoke_preparedmarel,invokedmarel_gs-
  submissionpartner_invoke_rungssubmission}
  /get,
// --- pool mapping to release a resource ---
  proc[P].{terminateInstance_gssubmission,
  jobsubmissionpartner_gssubmission_reply_
  submitjob,jobmonitoringpartner_gssubmission_
  reply_getjobstatus,gssubmission_gssubmission
  client_reply_rungssubmission,terminateIns-
  tance_invokedmarel,molpak2cmlpartner_invoked
  marel_reply_preparedmarel,gssubmissionpartner
  _invokedmarel_reply_rungssubmission,invoked
  marel_invokedmarelclient_reply_invokedmarel}
  /put}.
// --- configure set of client processes ---
 const Max_Processes = 2
 range P = 1..Max_Processes
```

Figure 7: Resource Mappings for BPEL interactions and Thread Pool

(GSSUBMISSION and INVOKEDMAREL) representing their process models. We then compose these orchestrations with their related port connector models, to represent the linked interactions between them. This linked model is a parallel composition named INVOKEDMAREL_GSSUBMISSION_RUNGSSUBMISSION_PORT, which combines invocation and reply models for these interactions of both orchestrations. A complete interaction model for the orchestrations is then provided (named CompositionArchitecture) which combines the port connector model and the process models described previously. The last process model to combine with the orchestration models is the deployment architecture. This is simply a combination of both CompositionArchitecture and the server resource pool which is allocated to the designated deployment server for the orchestrations. In the example given, this complete model is named "DEPLOY_ARCH". Note that each combined model is linked by the mapping of process activity labels.

4.4.6 Abstraction and Limitations

Inherent in model checking is a scaleablity issue relating to the degree of abstraction required, the state space of the models generated and the type of analysis performed [4]. The complete model discussed previously was based upon a number of abstractions for workflow and resource modelling. Firstly, in our orchestration mapping, our case study workflows used 200 parallel invocations of the same requirement and thus we were able to abstract to 2 client invocations (of a single invocation type) to check concurrency on a smaller scale. We have also assumed that only a single activity causes a new process instance to be created (for example the receive activity from a service client to the InvokeMolpak orchestration is modelled as the only process instance creation trigger) yet in more complex orchestrations there could be a number of such activities. We have also abstracted from data analysis (which may also affect behaviour of the orchestration). More specifically this means

```
// orchestration models
||GSSUBMISSION_BPELModel=(GSSUBMISSION_SEQUENCE1).
||INVOKEDMAREL_BPELModel=(INVOKEDMAREL_SEQUENCE1).
// orchestration port connector models
||GSSUBMISSION_JOBSUBMISSIONPARTNER_SUBMITJOB_PORT
  = (GSSUBMISSION_JOBSUBMISSIONPARTNER_SUBMITJOB_
  PORT_INVOKE || GSSUBMISSION_JOBSUBMISSIONPARTNER
  _SUBMITJOB_PORT_REPLY).
||INVOKEDMAREL_GSSUBMISSION_RUNGSSUBMISSION_PORT =
  (INVOKEDMAREL_GSSUBMISSIONPARTNER_
  RUNGSSUBMISSION_PORT_INVOKE || INVOKEDMAREL_
  GSSUBMISSIONPARTNER_RUNGSSUBMISSION_PORT_REPLY).
// composition architecture (process and ports)
||CompositionArchitecture = (GSSUBMISSION_
  BPELModel || INVOKEDMAREL_BPELModel || PORTS_
  GSSUBMISSION_JOBSUBMISSIONPORTTYPELINK_
  SUBMITJOB || PORTS_GSSUBMISSION_JOBMONITORING
  PORTTYPELINK_GETJOBSTATUS || PORTS_INVOKEDMAREL_
  MOLPAK2CMLPORTS_PREPAREDMAREL || PORTS_
  INVOKEDMAREL_GSSUBMISSIONPLINKTYPE_
  RUNGSSUBMISSION).
// deployment architecture of processes, ports
// and shared thread pool
||DEPLOY_ARCH = (proc[P]:CompositionArchitecture
  || TPOOL(10))
```

Figure 8: A summary of the complete model.

that a process may be influenced by the content of the messages rather than just by traditional control logic, which we currently do not observe. Our architecture modelling is based upon the configuration known to the specific technologies in our case study (namely Tomcat and ActiveBPEL), and it is likely that alternative technologies will exhibit different process and resource management patterns. This also relates to the size of the resource pool, which was predicted on a brief visual calculation of the process interaction types. We consider how we may address these limitations in our conclusions and further work.

5. VALIDATION

5.1 Detecting deployment deadlocks

A safety property Q in FSP is represented by an image of the LTS of the process expression that defines the property. The image LTS has each state of the original LTS and has a transition from each state for every action in the process alphabet of the original. Transactions added to the image LTS are to the error-state and signify a failure in verification. The LTSA tool has an inbuilt safety check to determine whether a specified process is deadlock free. Deadlock analysis of a LTS model involves performing an exhaustive search of the LTS for deadlock states (i.e. states with no outgoing transitions). Firstly, if we check our individual orchestration models for deadlock freedom they show positive (i.e. no deadlocks detected). Secondly, if we check the linked orchestrations, combining these models with port connector models, they show positive (again no deadlocks detected). Thirdly we check the complete model which includes all the elements of orchestration models, port connectors, server and orchestration allocation, and the server thread resource pool. The complete model is assigned a number of client instances, in this case we choose to verify against 2 concurrent client requests of the service orchestration. This time a deadlock is detected and is illustrated as a set of interactions in a Message Sequence Chart (MSC) in Figure 9. Note that normally the semantics of MSCs show an ordered set of

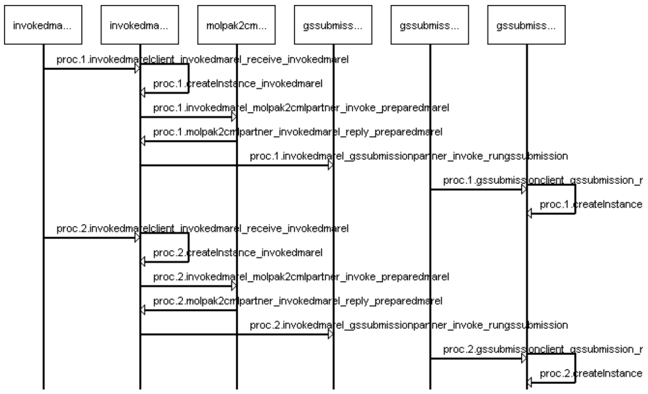

invokedma...
invokedma...
molpak2cm...
gssubmiss...
gssubmiss...
gssubmiss...
proc.1.invokedmarelclient_invokedmarel_receive_invokedmarel
proc.1.createInstance_invokedmarel
proc.1.invokedmarel_molpak2cmlpartner_invoke_preparedmarel
proc.1.molpak2cmlpartner_invokedmarel_reply_preparedmarel
proc.1.invokedmarel_gssubmissionpartner_invoke_rungssubmission
proc.1.gssubmissionclient_gssubmission_r
proc.1.createInstance
proc.2.invokedmarelclient_invokedmarel_receive_invokedmarel
proc.2.createInstance_invokedmarel
proc.2.invokedmarel_molpak2cmlpartner_invoke_preparedmarel
proc.2.molpak2cmlpartner_invokedmarel_reply_preparedmarel
proc.2.invokedmarel_gssubmissionpartner_invoke_rungssubmission
proc.2.gssubmissionclient_gssubmission_r
proc.2.createInstance

Figure 9: Trace to DEADLOCK in polymorph case study for 2 concurrent client requests and 2 orchestrations deployed on a server with a shared thread pool

interactions. For ease of illustration, we have included two sets of interactions on one chart, whereby the parallel execution of client requests 1 and 2 are shown with 2 following 1 - although they are infact concurrent.

The reason for this deadlock is an exhausted resource thread pool allocation, whereby a request to create a new instance of the GSSUBMISSION orchestration process is never fulfilled due to the interaction causing an activity of requesting for a new thread. A summary of this thread management against activity and process is listed in Table 1. The concurrent allocation of thread resources for client requests 1 and 2 is the cause for the pool limit to be reached, with each request waiting for a response which can never be received. For each request a thread is allocated upon the process “receive” activity. A new instance of the InvokeMarel orchestration process is then created, and an invocation of the Molpak2CMLPartner is invoked (creating 2 more threads). Replies to these invocations are accepted (which release the allocated threads for the invocation to the Molpak2Partner service), and two further invocations are created to the GSSUBMISSION orchestration process (allocating two threads in the sequence). The invocations are received by the GSSUBMISSION host (creating 2 new threads) and an attempt to create two new instances of GSSUBMISSION is undertaken. For the first request a thread is allocated, however for the second request there are no more available thread resources available in the pool. A deadlock situation has occurred whereby neither of the invocations can be replied to as the GSSUBMISSION would also require further threads to undertake its activities (invocations of other services).

Activity	*ToPartner*	*P1*	*P2*
receive	invokedMarel	9	8
createProcess	invokedMarel	7	6
invoke	molpak2cmlpartner	5	4
reply	invokedMarel	5	6
invoke	gssubmission	5	4
receive	gssubmission	3	2
createProcess	gssubmission	1	0

Table 1: Thread allocation in trace to deadlock

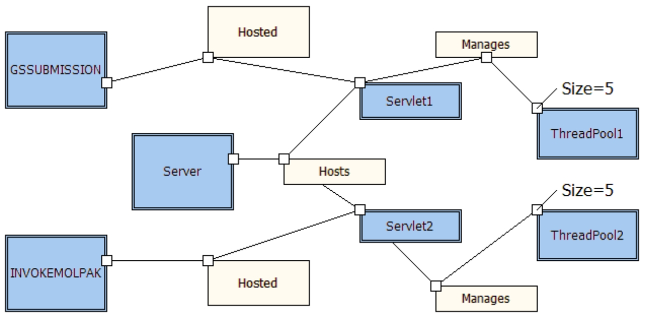


Figure 10: Refined Architecture Model

5.2 Resolving deployment deadlocks

There are a number of options to resolve this deadlock. Firstly, we could simply increase the number of threads available in the shared resource pool. If we double the pool to 20 threads available, the deadlock is resolved. However, in terms of the architecture this may eventually lead to a limit of the maximum number of threads available for a server. This option could also impact performance of the server overall, with other shared services suffering as a consequence. An alternative option is to split the shared pool and host the orchestration on a separate instance. In this case the total number of resources reserved remains the same however, we now model two thread resource pools and model that each of the two orchestrations in our case study are allocated to individual thread pools. The FSP for this solution changes the main composition and process P relabelling as listed below for P1 ”get” and P2 ”put”:

```
||DEPLOY_ARCH2 = (proc[P]:CompositionArchitecture
        || p1:POOL(5)|| p2:POOL(5))/
  {proc[P].{createInstance_gssubmission,
        gssubmissionclient_gssubmission_receive_
        rungssubmission,gssubmission_jobsubmission
        partner_invoke_submitjob,gssubmission_
        jobmonitoringpartner_invoke_getjobstatus}
        /p1.get,
   proc[P].{terminateInstance_invokedmarel,molpak2
        cmlpartner_invokedmarel_reply_
        preparedmarel,gssubmissionpartner_
        invokedmarel_reply_rungssubmission,
        invokedmarel_invokedmarelclient_
        reply_invokedmarel}
        /p2.put}.
```

Note that each of the interactions for each orchestration are split between a thread ”get” (to allocate a thread) and a thread ”put” (to release the thread back to the pool) and that these are assigned to each instance of a thread pool ”p1” and ”p2”. Thus, for each client request, each orchestration will utilise a thread pool designated by the orchestration and activity mapped. The architecture model is similarly updated to reflect this split, and is illustrated in Figure 10. In summary, we have introduced a new orchestration host servlet (container) and a new associated shared thread pool.

If we perform our deadlock analysis again, the interactions of 2 concurrent client requests does not cause a deadlock situation where one is blocking the other from completing a set of interactions. This mirrored the solution taken in the case study (section 3.1) in which the deployment architecture was reconfigured with a new servlet to host one of the two service orchestrations.

5.3 Different resource types

In our analysis we have considered the thread pool management as a significant resource to model as it was the cause of the initial deadlock in our case study. There are of course, a number of resources which are managed by service technology architectures including memory allocation, processor instances, and port availability. Our approach can be used to define and model these constraints and include them in analysis. The broader scope of analysis could contribute to consideration of Service Level Agreements (SLAs) in a service architecture being both from a perspective of the level of service quality to the client, and also that the provider is not subjected to a request load that they cannot fulfil. In [21] an approach to monitor the timeliness of service provision (the duration in which requests can be completed) at runtime is discussed. Reasoning about the provision of services to clients is critical in maintaining a SLA, particularly where an exhausted service resource pool leads to a violation of the SLA.

5.4 Tool Support in WS-Engineer

A tool, illustrated in (Figure 11), is known simply as "WS-Engineer" and is built on the existing LTSA tool suite [13]. The tool offers an extendable framework to support multiple editor and view types using the core Eclipse framework. The WS-Engineer plug-in provides editors for BPEL4WS and FSP and FSP code is automatically generated for the BPEL processes supplied. From an FSP description, the tool generates an LTS model. The user can animate the LTS by stepping through the sequences of actions it models, and model-check the LTS for various properties, including deadlock freedom, safety and progress properties. The WS-Engineer plug-in architecture leverages our previous work and is extended to support the approach in this paper through Deployment views. In this work we have used XML, WSDL, BPEL4WS and xADL, but we also support modelling WS-CDL for service choreography models. The WS-Engineer Eclipse plug-in is available for download from the following web page: http://www.doc.ic.ac.uk/ltsa.

6. CONCLUSIONS AND FURTHER WORK

In this paper we have presented an approach to the modelling, analysis, detection and resolution of deadlocks for a web services composition under a number of resource constraints. Our approach applies a modelling of web service compositions in the form of a translation of BPEL4WS services to FSP and a representation of architectures with resources. Within our approach however, are a number of assumptions. In this first case we have abstracted several areas of the composition, and in particular to scale the processes for modelling efficiency. We plan to experiment with large process compositions to test the approach and technology. We also assume that the user can describe the behaviour of service orchestration engines (for example the semantics for the ActiveBPEL engine), yet once defined these could be gathered to provide a suite of alternative architectures to test certain configurations upon iteratively prior to deployment (and perhaps find an optimal architecture). We also currently have a limited representation of BPEL4WS semantics, in that we assume a single activity marks the instance creation of a service component. This leads us to explore further work to expand the coverage of multiple activities that can create a new instance of a service component.

We plan to expand the approach to consider other resource constraints. The scope is also defined to cover other properties such as dynamic analysis of policies for service interactions in service choreography and also the analysis of composition deployments on distributed architectures.

Acknowledgements

We are grateful to Liang (Ben) Chen and Bruno Wassermann for identifying the problem and helping us understand where the problem originated. We thank Sally Price for providing a web service orchestration that was sufficiently resource demanding for the problems to occur. This work has been funded partly by the EU project IST-2005-16004 (SENSORIA), EPSRC project GR/S90843/01 (OMII-BPEL) and by an IBM Eclipse Innovation Award (2006).

7. REFERENCES

[1] Active-Endpoints. Activebpel engine, 2005. Available at: http://www.activebpel.org.

[2] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business process execution language for web services version 1.1, 2004.

[3] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture, w3c working group note 11 february 2004, 2004. Available at: http://www.w3.org/TR/ws-arch/.

[4] T. Bultan and A. Betin-Can. Scalable Software Model Checking Using Design for Verification. In *Proceedings of the IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, volume 989, Zurich, Switzerland, 2005.

[5] Chakrabarti, A., Alfaro, L.D., Henzinger, T.A., and Stoelinga, M. Resource interfaces. In *Third International Conference on Embedded Software (EMSOFT 2003)*. ACM Press, 2003.

[6] Eric M. Dashofy, Andre Van der Hoek, and Richard N. Taylor. A highly-extensible, xml-based architecture description language. *wicsa*, 00:103, 2001.

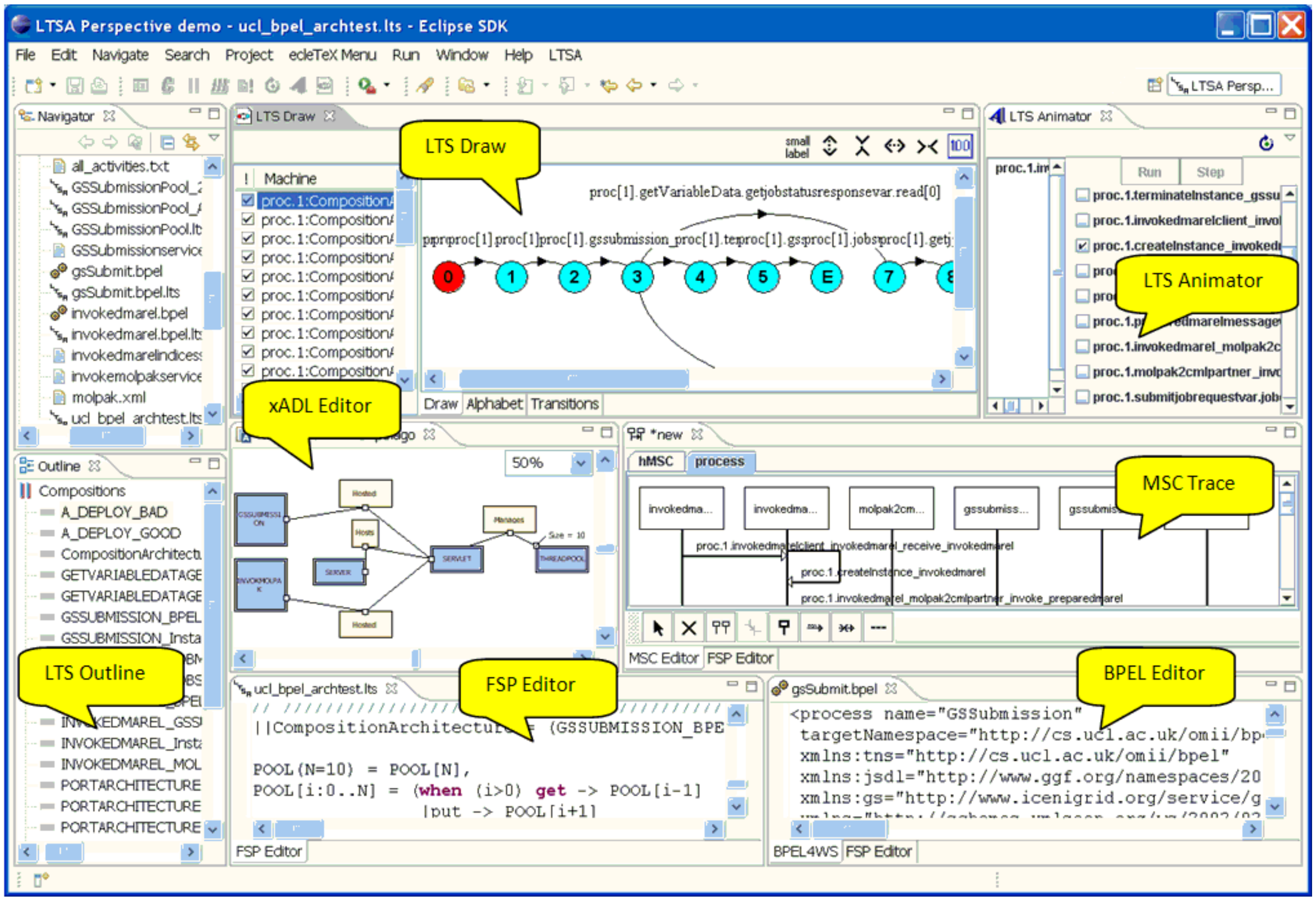
[7] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid Service Orchestration using the Business Process Execution Language (BPEL). *Journal of Grid Computing*, 3(3-4):283–304, 2005.

[8] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *Proc. of the* 18^{th} *IEEE Int. Conference on Automated Software Engineering*, pages 152–161. IEEE CS Press, 2003.

[9] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Compatibility for web service choreography. In *3rd IEEE International Conference on Web Services (ICWS)*, San Diego, CA, 2004a. IEEE.

[10] Xiang Fu, Tevfik Bultan, and Jianswen Su. Wsat: A tool for formal analysis of web services. In *16th International Conference on Computer Aided Verification (CAV)*, Boston, MA, 2004.

[11] C. Goyette. Xml applied to product line software development, 2005. Available from: http://www.mcc.com/projects/ssepp/papers.



```
// ///////////////////
||CompositionArchitecture = (GSSUBMISSION_BPE

POOL(N=10) = POOL[N],
POOL[i:0..N] = (when (i>0) get -> POOL[i-1]
              |put -> POOL[i+1]
```

```
<process name="GSSubmission"
 targetNamespace="http://cs.ucl.ac.uk/omii/bp
 xmlns:tns="http://cs.ucl.ac.uk/omii/bpel"
 xmlns:jsdl="http://www.ggf.org/namespaces/20
 xmlns:gs="http://www.icenigrid.org/service/g
```

Figure 11: The WS-Engineer Tool Environment

[12] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web services composition. In *3rd IEEE International Conference On Web Services (ICWS)*, San Diego, CA, 2004.

[13] Jonathan Haymann. The application of a resource logic to the non-temporal analysis of processes acting on resources (hpl-2003-194). 2003. Available at: http://www.hpl.hp.com/techreports/2003.

[14] W. S. Lee, A. S. McGough, S. Newhouse, and J. Darlington. A Standard Based Approach to Job Submission through Web Services. In S. Cox, editor, *Proc. of the UK e-Science All Hands Meeting, Nottingham, UK*, pages 901–905. UK EPSRC, 2004.

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[16] J. Magee and J. Kramer. *Concurrency - State Models and Java Programs - 2nd Edition*. John Wiley, 2006.

[17] Jeff Magee, Jeff Kramer, and D. Giannakopoulou. Analysing the behaviour of distributed software architectures: a case study. In *5th IEEE Workshop on Future Trends of Distributed Computing Systems*, Tunisia, 1997.

[18] OMG. Unified modelling language (uml) 2.1.1 specification, 2007. Available from: www.uml.org.

[19] C. Ouyang, W.v.d Aalst, S. Breutel, M. Dumas, A.t. Hofstede, and H. Verbeek. Formal semantics and analysis of control flow in ws-bpel (revised version) bpm-05-15. Technical report, BPMcenter. org, 2005.

[20] Pyarali, I., Spivak M., Cytron, R., and Douglas C., S. Evaluating and optimizing thread pool strategies for real-time corba. In *ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, Language, Compiler and Tool Support for Embedded Systems. ACM Press, 2001.

[21] F. Raimondi, J. Skene, L. Chen, and W. Emmerich. Efficient monitoring of web service slas (research note rn/07/01). 2007. Available at: http://www.hpl.hp.com/techreports/2003.

[22] G. Salan, L. Bordeaux, and M. Schaerf. Describing and reasoning on web servicesusing process algebra. In *3rd IEEE International Conference On Web Services (ICWS)*, San Diego, CA, 2004.

[23] The-Apache-Software-Foundation. Apache-tomcat 5.5, 2005. Available at: http://tomcat.apache.org/.

[24] Chris Tofts. Efficiently modelling resource in a process algebra (hpl-2003-181). 2003. Available at: http://www.hpl.hp.com/techreports/2003.