# Towards a Document Composition Framework

**Mike Rogers, Sheikh Ghafoor, Brett Harper**
Department of Computer Science, Tennessee Technological University, Cookeville, TN, USA

**Abstract**— *This paper reports the design and evaluation of a declarative language based document creation frame work called Braid. Braid can create new XML documents from other xml documents distributed throughout the Web. Recurring patterns of steps are required to compose a single complex document from multiple distributed documents. We conducted study of forty six workflows and, combined, they used sixteen workflow control patterns to compose documents. Braid supports almost all the patterns that are required to compose documents in our study.*

**Keywords:** Declarative language, Document Composition, Workflow Pattern, Web Services

## 1. Introduction

Service Oriented Computing has become a popular paradigm for building complex internet applications, largely due to the introduction of several well-defined technologies and standards, such as SOAP [1], UDDI [2], WSDL [3], WSSec [4], BPEL [5], and a multitude of ESBs [6], [7], [8], [9], [10]. These technologies implement messaging between services, compose basic services into compound services, provide for the discovery of web services, and describe web service interfaces and protocols. Additionally, workflow languages provide a mechanism for describing the necessary steps to compose complex services out of multiple simple services.

Recurring patterns of steps needed to compose web services have emerged, and various workflow languages exist that can implement these patterns. Workflow languages are usually XML based and allow the programmer to specify workflows that compose web services. Many of these workflow languages such as BPEL are designed to model general purpose business processes. They are imperative languages with which the programmer describes each step of an instance of a workflow known as a *process*. It requires the programmer to specify the complex workflow in great detail. For each step in the workflow, the source of input and destination of output may need to be specified. In addition, a step may need to be bound in some way to an external service, and the type and number of inputs and outputs must match the external service. Furthermore, imperative languages such as BPEL have to maintain global and local state.

In many e-commerce and information-based systems, web services are used to create XML documents. Unfortunately, describing how to compose a document in a language meant to describe workflows can be difficult. The step-by-step instructions of a workflow language require a complex level of detail to create complex XML documents. We have developed a document creation framework called Braid that can create new XML documents from other XML documents distributed throughout the web. It provides a declarative document composition language to handle the majority of common document generation tasks.

Braid's document composition language was designed to be simple, lightweight, and non-imperative. Braid has very few concepts and constructs. Because Braid's language is non-imperative, the programmers do not have to specify the steps to create a document, but, instead, they describe the document dependencies. The task of determining the order and parallelization of document construction is left to Braid itself, thereby simplifying the task for the programmer.

In this paper, we present the Braid document description language, the architecture of Braid's components that implement Braid's document composition algorithms, and evaluate Braids usefulness by comparing Braid's document composition features to common workflow patterns that are required to compose documents. Van der Aalst [11], [12] provides an extensive list of workflow control patterns (WCP) that can be used to compose services. Not all of these WCPs are required to create a service. For example BPEL supports 22 of the 43 WCP mentioned in [11]. Our limited case study of 46 example workflows found that, combined, they used only 16 out of 43 WCPs mentioned in [11]. Braid supports almost all the WCPs that are required to compose documents in our study.

The rest of the paper is organized as follows. Section 2 presents a brief discussion on related work. The architecture of Braid's document composition framework is presented in Section 3. An evaluation of Braid's document composition capability is discussed in Section 4. Our conclusions and future works are presented in Section 5.

## 2. Related Work

Various technologies exist that are related to Braid and our work on Braid's document description language. These technologies include Web Services and Service Oriented Computing, Enterprise Service Buses, the BPEL workflow language, and workflow patterns.

## 2.1 Web Services and Service-Oriented Computing

Service-oriented computing and service-oriented architectures are the basic building blocks of web services. A web service allows multiple machines to interact across a wide-area network. Clients communicate with Web services using a text-based messaging protocol such as SOAP. These messages are usually delivered using HTTP in conjunction with other web standards [13]. Services should be able to run on any type of machine, such as Windows or Unix-based machines. The actual hardware that hosts the service should not affect clients, as long as the input and output of the service are well-defined and agreed upon.

Service Oriented Architectures describe high-level structures used in building internet applications and how these internet applications are built from services. Service Oriented Computing encompasses a set of paradigms in addition to the tools used to implement those paradigms. These paradigms and tools can then be used to construct service-oriented internet applications. Communication between consumers of a service and the service itself is typically accomplished through providers, registries, and contracts [14]. Providers register their services with the registry that a consumer can then use to locate a service. A service contract specifies how the interaction will take place and what conditions need to be met in order for the service to be executed.

The Braid document generator has much in common with traditional service oriented architectures. In Braid, a document can be composed from multiple distributed documents, just as a service can be composed from other services. Documents are transferred using an agreed-upon protocol, similar to how a message is transferred between services using a protocol like SOAP. However, Braid is not intended to be a replacement for Web services. Braid is intended, instead, to compliment existing Web service technologies to provide easily configured and efficient document generation.

## 2.2 Enterprise Service Bus

Enterprise Service Buses [6], [7], [8], [9], [10] combine a number of software packages, standards, and protocols in order to implement a service oriented architecture. A number of Enterprise Service Buses are available to the public. Each performs the same basic functions. An ESB typically includes a messaging framework. ESBs also include an application server that is used to host the web services. Additionally, an ESB supports writing Web-service code in a general purpose programming language, such as Java or Python. Finally, an ESB also implements a workflow language, such as BPEL, so that the programmer can specify the ordering and composition of services. In general, workflow composition languages allow the programmer or designer to specify the flow of both data and execution, though each language has a different approach.

Braid, like Enterprise Service Buses, utilizes a messaging protocol to transport requests and replies for documents. Furthermore, Braid, like ESB's that support service composition, provides a language that is used to describe the composition of documents. However, Braid's document description language is designed particularly for composing distributed documents, not for general purpose workflows.

## 2.3 BPEL

BPEL is a language created to describe business processes [5]. BPEL is used as the workflow language in a number of ESBs. For example, BPEL is used as a component in Oracle ESB [8] and Apache ServiceMix [10]. Other enterprise service buses, such as Apache Synapse [6] and Microsoft Biztalk ESB [7], make use of other workflow languages or create their own. Even so, BPEL is a popular workflow language.

BPEL allows the use of control structures such as those found in most programming languages. These include sequence and switch statements as well as while loops. Furthermore, services that run in parallel are specified in BPEL using a special control structure. BPEL also specifies a way to wait a certain length of time for results, make choices based on external events, or even do nothing [5]. BPEL uses these control structures to specify the exact sequence of steps in a workflow. However, this approach can make service composition complex as the programmer must specify all the details of the workflow.

BPEL and Braid's document description language share similar roles within their respective frameworks. BPEL, and other workflow languages, allow the programmer to compose individual services into compound services, which is similar to Braid's document description language, which allows programmers to compose documents from other distributed documents. However, unlike BPEL, Braid is designed to compose documents, not services, and Braid uses a non-imperative language to do so.

## 2.4 Workflow Patterns

After constructing a number of workflows, a programmer may notice certain patterns that invariably show up in most workflows. W. M. P. van der Aalst et al. [12] identified a total of 43 such patterns and proceeded to map these workflow patterns to functionality in various workflow languages, such as BPEL [11]. No single language has been shown to implement all 43 patterns, though there are several patterns that are implemented in all languages. In fact, there are some patterns that are much more prevalent than others.

These patterns describe a wide range of actions that may be accomplished in a typical workflow. They can describe ordering of actions, canceling or stalling actions, the merging and splitting of a path, looping over a set of actions, and even spawning multiple instances of the same action.

Workflow patterns have been used in the past as a measure of comparison between workflow languages. Though Braid's document description language is not meant to describe workflows, we believe it is similar enough to workflow languages that these patterns can be used to determine functional gaps within the language. Part of our work described in this paper is determining which patterns can aid us in our gap analysis.

## 3. Braid Document Composition and Architecture

Braid is a document composition framework that can create new documents from multiple distributed documents. The document description language in Braid is a dependency based declarative language, as opposed to an imperative workflow language such as BPEL. Braid is document oriented not service oriented. In other words, the focus is not on creating, calling, and retrieving results from services, but rather on creation of new document by transforming and combining distributed documents.

In Braid, a document is modeled as a set of dependent activities, instead of a set of state transitions [15]. In this model, the programmer does not determine the flow of activities. Instead, Braid determines the actions to perform at run-time. While this model does take away the complexity of determining control flow and concurrency from programmers, it introduces the activity of determining dependencies among documents. In practice, a programmer would create a file that describes these dependencies. Braid reads these dependency files and uses them to generate documents. The dependency file is similar conceptually to a UNIX makefile.

This section first describes how Braid composes documents given the document dependencies, and then describes Braid's architectural components that implement Braid's document composition algorithm.

### 3.1 Braid Document Composition

Consider the example given in Figure 1. In this example, a loan broker service provides customers with personalized quotes for loans from banks. The customer makes the request by filling out a form in a standard web browser. Once the loan broker service receives the customer's request, the loan broker retrieves a credit report from the credit bureau for the customer. Then the credit bureau simultaneously requests quotes from the two participating banks, merges the results from the two banks, formats those results into a final XHTML report document, and sends the final report back to the customer.

Implementing such a loan broker service requires creating a Braid document description that describes the needed components and actions required by Braid to construct the result document. Figure 2 shows a Braid document description for the example in loan broker service.

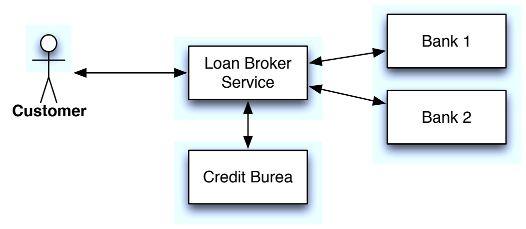


Fig. 1: Example Document Composition

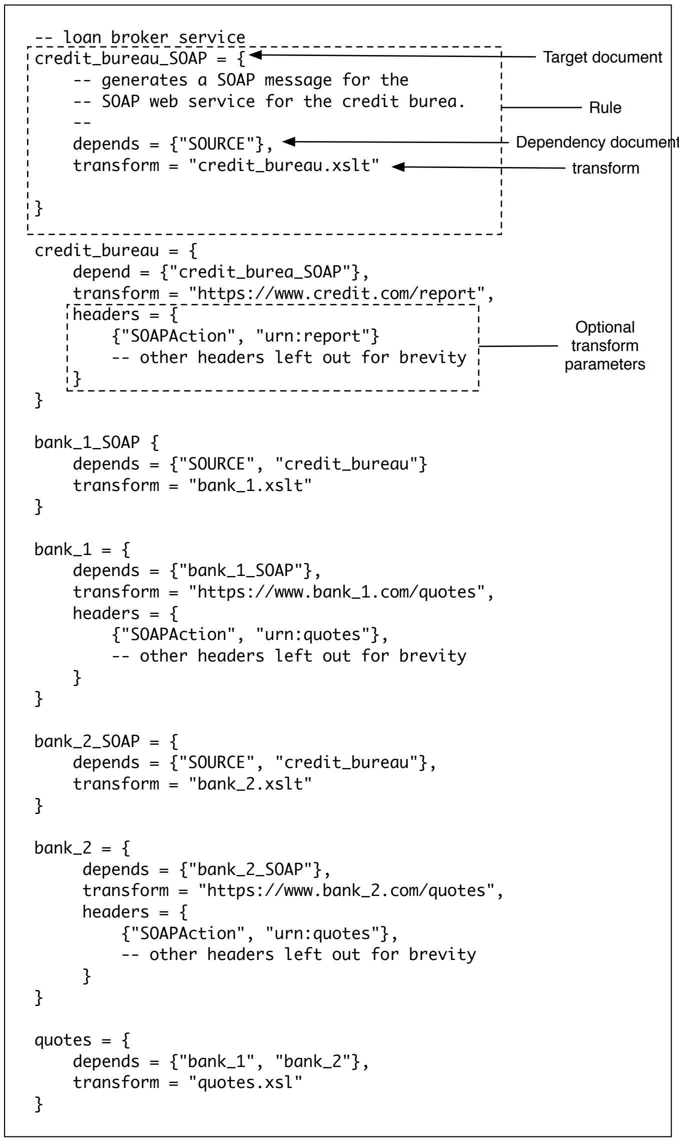
Each rule is made up of a target document, a list of dependencies, a transform, and optional parameters for the transform. Braid will construct a directed, acyclic graph (DAG) from the document description, and then traverse this graph, applying each transform in order to construct the given target from the given dependencies. A DAG for the loan broker service is depicted in Figure 3. Braid creates a target document named SOURCE from the request sent by the browser. This target has no transform, so the source document is passed immediately to the rule for the dependent document credit_bureau_SOAP. Braid applies the transform for credit_bureau_SOAP to the source document to create a SOAP message. Braid then applies the rule for credit_bureau to the resulting soap message, which results in Braid invoking the SOAP web service, via the HTTP protocol, at the URL given for the transform.

Next, Braid, in parallel, applies the rule for the bank_1_SOAP and bank_2_SOAP document targets by merging each targets dependencies, which are the source document and the results of credit_bureau. Then, in parallel, Braid applies the rules form bank_1 and bank_2 by calling the corresponding SOAP web service. Braid applies the final rule for the quotes document target by merging the results from its dependencies and transforming the dependency results into a final XHTML document via the XSLT transform. Lastly, Braid send these result back to the customers browser.

### 3.2 Braid Components

In order to implement Braid's document composition algorithm, Braid consists of a distributed architecture. Figure 4 shows the high-level, distributed architecture of Braid. Braid consists of collaborating, Internet connected Braid servers. Each outer-most rounded box in the figure represents a Braid server instance. Each Braid instance in the proposed architecture consists three major parts: a web service interface exposed by the Braid instance, a Braid transform module, and a local/remote transform module.

A Client, or another Braid instance, requests a document from Braid via Braid's REST web services interface. The request is in the format of an XML input document. The web service interface then passes the input document to the Braid engine which consists of the Braid transform module and a local/remote transform module. The Braid transform



```
-- loan broker service
credit_bureau_SOAP = {
    -- generates a SOAP message for the
    -- SOAP web service for the credit burea.
    --
    depends = {"SOURCE"},
    transform = "credit_bureau.xslt"

}
credit_bureau = {
    depend = {"credit_burea_SOAP"},
    transform = "https://www.credit.com/report",
    headers = {
        {"SOAPAction", "urn:report"}
        -- other headers left out for brevity
    }
}

bank_1_SOAP {
    depends = {"SOURCE", "credit_bureau"}
    transform = "bank_1.xslt"
}

bank_1 = {
    depends = {"bank_1_SOAP"},
    transform = "https://www.bank_1.com/quotes",
    headers = {
        {"SOAPAction", "urn:quotes"},
        -- other headers left out for brevity
    }
}

bank_2_SOAP = {
    depends = {"SOURCE", "credit_bureau"},
    transform = "bank_2.xslt"
}

bank_2 = {
     depends = {"bank_2_SOAP"},
     transform = "https://www.bank_2.com/quotes",
     headers = {
         {"SOAPAction", "urn:quotes"},
         -- other headers left out for brevity
     }
}

quotes = {
    depends = {"bank_1", "bank_2"},
    transform = "quotes.xsl"
}
```

Fig. 2: Braid Document Description

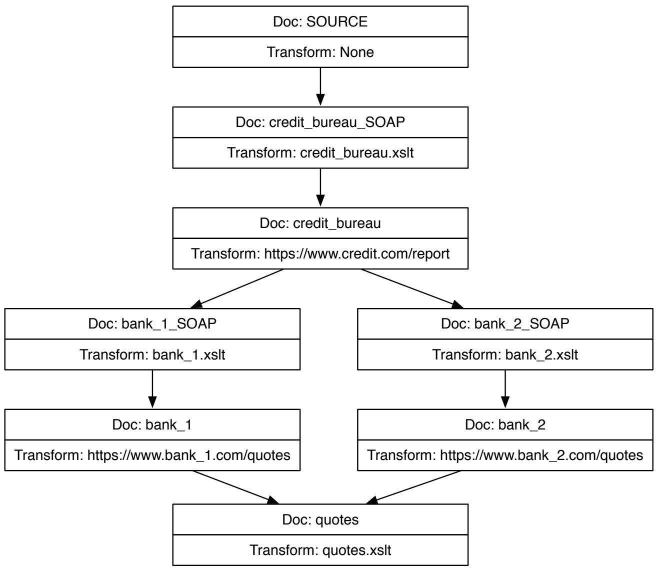


Fig. 3: The DAG for the loan broker example.

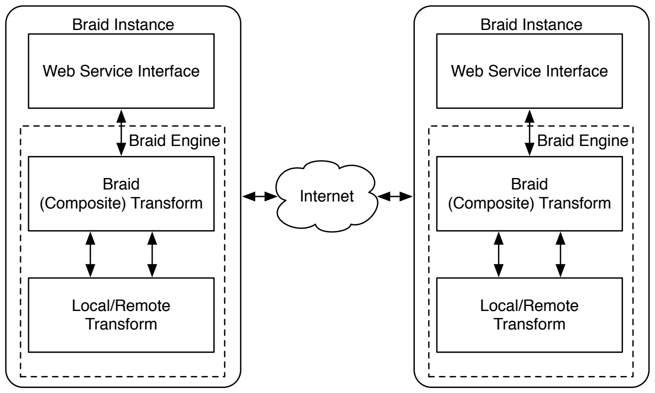


Fig. 4: High Level Architecture

module is the kernel of the Braid server and coordinates the overall functioning of the server. The Braid transform module implements a dependency based transform on the input document and other documents retrieved or constructed by the local/remote transform module.

More detail of the Braid transform module and the local/remote transform module is shown in Figure 5. The Braid transform module consists of the transform engine, and a dependency processor. The transform engine maintains a map of document requests via each document's REST URL to its corresponding dependencies. When the transform engine receives an input document it finds the corresponding dependencies from the map and loads them from the configuration file. It then passes the dependencies and input document to the dependency processor.

The dependency processor behaves similarly to the Unix Make utility as it processes Makefiles. The dependencies form a directed, acyclic graph that the dependency processor negotiates and, at each node in the graph, executes an associated action. To execute the specified action, it invokes the corresponding sub-module in the local/remote transform module. For each target, the dependency processor merges the results of its dependencies and then invokes one of the following: the script module that executes a processing script written in a general purpose scripting language (Lua is currently supported), the pull protocol module that fetches a remote document via a supported protocol (libcurl is the current implementation, which supports many protocols, including HTTP, HTTPS, FTP, and SCP), or invokes the XSLT transform module to apply an XSLT transform. These modules support a wide range of functionality for generating documents. For example, if XSLT does not provide

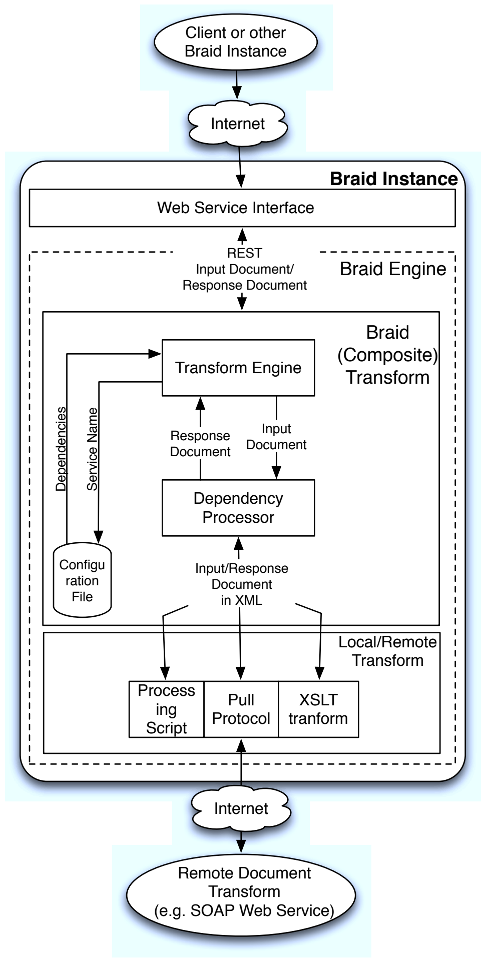


Fig. 5: Detailed architecture of a Braid server instance.

the needed functionality to transform the document, or a particular protocol is not supported that is needed to fetch a document, then a script can be written to provide arbitrary processing.

As it visits each node in the dependency graph, the dependency processor updates the state of its targets with the results from the local/remote transform sub-modules. Each target that has a result is considered to be satisfied.

Once a target that has no other dependents, i.e. the root of the dependency tree, is satisfied, the Braid transform is considered complete, and the transform engine returns the results of the satisfied dependency to the caller via the web service interface.

## 4. Evaluation of Braid

Van der Aalst [12], [11] in his seminal work identified and described a comprehensive list of workflow control patterns (WCP) that can be used as a basis for evaluating and comparing workflow management systems. Not all of these forty three work flow patterns identified in [11] are supported in most of the commercial workflow systems. For example BPEL supports 22 of the 43 WCPs. Many of the WCPs mentioned in [11] are not required to implement most common workflows.

To evaluate Braid's document composition capability, we first needed to determine which WCPs are essential in composing most documents. Unfortunately no statistics or studies have been published on the frequency of WCPs used to compose documents. Therefore, we performed our own case study of 46 examples.

Many of these examples contain more than one workflow and multiple WCPs. These examples are gathered from tutorials, white papers, books, and journal articles. One of the major difficulties in such a study is that most service composition workflows are proprietary because they are developed and used by commercial organizations. They do not publish or disclose these workflows. So obtaining a comprehensive list of representative workflows is difficult. Note that our study is not comprehensive or exhaustive. However, it gives us a reasonable indication of what workflow patterns are most commonly used when composing documents.

Figure 6 shows a scatter plot WCPs that are found in our case study. The y axis shows the number of examples where a particular pattern is used. It can be seen from the plot that only 16 out 43 patterns are found to be used in our case studies. Table 1 lists the WCPs that are used along with whether Braid supports them or not.

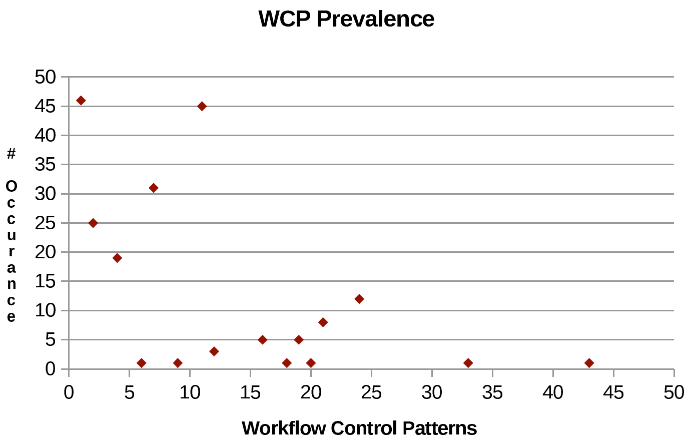


Fig. 6: WCP occurrences in case study.

As can be seen from the table that Braid supports 11 out of the 16 WCPs that are found in our study. In addition Braid can support three more WCPs (exclusive choice, Multi-choice, Cancel Activity) with minor enhancement. In its present form Braid doesn't support Structured Discriminator and Persistent Trigger patterns.

The structured discriminator pattern is not possible in Braid. In this pattern, only the first incoming branch to be activated is allowed to continue to the subsequent item, all other incoming branches are ignored. Ignoring branches in

| WCP | Description | Occurrence | Braid |
|---|---|---|---|
| 1 | Sequence | 46 | yes |
| 2 | Exclusive | 25 | yes |
| 4 | Exclusive Choice | 19 | can |
| 6 | Multi-Choice Structure Synchronizing | 1 | can |
| 7 | Merge | 31 | yes |
| 9 | Structured Discriminator | 1 | no |
| 11 | Implicit Termination Multiple Instances w/o | 45 | yes |
| 12 | Synchronization | 3 | yes |
| 16 | Deferred Choice | 5 | yes |
| 18 | Milestone | 1 | no |
| 19 | Cancel Activity | 5 | can |
| 20 | Cancel Case | 1 | yes |
| 21 | Structured Loop | 8 | yes |
| 24 | Persistent Trigger | 12 | no |
| 33 | Generalized AND-join | 1 | yes |
| 43 | Explicit Termination | 1 | yes |

Table 1: Workflow pattern occurrences

```
warehouse = {
    depends = {},
    transform = function (xml)
        local socket = require("socket")
        local server = socket.bind("*", 0)
        local ip, port = server:getsockname()
        local client = server:accept()
        client:settimeout(1000)
        local all, err = client:receive("*a")
        client:close()
        server::close()
        if not err then
                return all
        else
                return "<EMPTY />"
        end
    end
}
```

Fig. 7: A Braid rule approximating Persistent Trigger

Braid is not possible because each document has a list of dependencies that must all be satisfied before execution can continue on that branch. Braid has no concept of waiting for a subset of its dependencies to be fulfilled. However, Structured Discriminator is rarely required in document composition. In our case study we found only one instance of Structured Discriminator.

Persistent Trigger is not supported due to Braids communication model. This pattern requires that the service be able to handle external events. However, Braid uses a pull model where Braid fetches remote dependency documents via its protocol module; It is not able to handle asynchronous events from the operating environment. Unfortunately for Braid, occurrence of persistent trigger in workflows to produce documents is significant, and does indicate a possible functional gap in Braid.

However, the functional gap is mitigated because Braid can use its scripting module to listen for incoming messages and approximate the behavior of asynchronous communication. For example, consider the simplified Braid rule in Figure 7. This rule has no dependencies, so it will immediately fire when the corresponding DAG is traversed. This rule creates a TCP socket and listens for an incoming message. Once it receives the message, it returns the message as the result of the transform. In our evaluation, although improvements to Braid can be made, Braid provides the functionality needed to compose documents from distributed resources.

## 5. Summary and Conclusions

We have presented a distributed document composition frame work called Braid. It provides a simple lightweight declarative language to compose new XML document from multiple documents distributed throughout the web. Unlike imperative language based workflow engines, in Braid a programmer does not have implement, step by step, how to compose a document. A programmer declares the document dependencies which are internally represent as a directed acyclic graph, and Braid automatically transforms and combines the distributed documents to compose the new document.

To evaluate the Braid's document composition capability we performed case study of 46 examples to determine which workflow control patterns are used in document composition. Braid supports 11 out of the 16 workflows found in the study. It can support additional workflow control patterns with minor modification. In it's current form Braid does not directly support two workflow patterns: Structured Discriminator and Persistent trigger.

Our immediate planned future work includes extension of Braid to support Exclusive Choice and Multi-Choice patterns. The exclusive choice pattern describes a situation where, based upon a certain condition, a single path out of many is taken through a workflow. The exclusive choice pattern is not directly implemented in Braid, but can be done by making use of a script as a transform. The script performs the check to determine if the document's external transform should occur. If this is not the case, the script returns a document telling the external transform to skip execution. This process can be considered a workaround, since the pattern is not directly supported within the language itself.

The multi-choice pattern is a more general form of the exclusive choice pattern, in which more than one branch can be taken. We intend to make minor modifications to Braid's dependency processor to support both exclusive and multi-choice patterns.

## References

[1] "Simple Object Access Protocol (SOAP)." http://www.w3.org/TR/soap12. [Accessed on March 26, 2012].

[2] "Universal Description, Discovery, and Integration (UDDI)." http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3. [Accessed on March 26, 2012].

[3] "Web Service Description Language (WSDL)." `http://www.w3.org/TR/wsdl`. [Accessed on March 26, 2012].

[4] "Web Services Security: SOAP Message Security 1.1." `http://www.oasis-open.org/committees/download.php/16790/wssv1.1-spec-os-SOAPMessageSecurity.pdf`. [Accessed on March 26, 2012].

[5] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Lui, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana, "Business Process Execution Language for Web Services." `http://www.ibm.com/developerworks/library/specification/ws-bpel/`. [Accessed on March 26, 2012].

[6] "Apache Synapse, the Lightweight ESB." `http://synapse.apache.org/index.html`. [Accessed on March 26, 2012].

[7] "Microsoft Biztalk Server ESB Guidance." `http://www.microsoft.com/biztalk/en/us/esb-guidance.aspx`. [Accessed on March 26, 2012].

[8] "Oracle Enterprise Service Bus." `http://www.oracle.com/technology/products/integration/esb/index.html`. [Accessed on March 26, 2012].

[9] Jackie Wheeler, "What is Mule ESB?." `http://www.mulesoft.org/what-mule-esb`. [Accessed on March 26, 2012].

[10] James Strachan and Lars Heinemann, "Apache Servicemix, the Agile Open Source ESB." `http://servicemix.apache.org/home.html`. [Accessed on March 26, 2012].

[11] N. Russell, A. H. M. ter Hofstede, W. M. P. van der Aalst, and N. Mulyar, "Workflow Control-flow Patterns: A Revised View," Tech. Rep. Report BPM-06-22, BPM Center, BPMcenter.org, 2006.

[12] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, pp. 5–51, July 2003.

[13] "Web Services Glossary." `http://www.w3.org/TR/ws-gloss/`. [Accessed on March 26, 2012].

[14] Michael N. Huhns and Munidar P. Singh, "Service Oriented Computing: Key Concepts and Principles," *IEEE Internet Computing*, pp. 2–8, Jan, Feb 2005.

[15] M. Rogers, S. Ghafoor, and J. Graves, "Braid: Distributed Patient Information for Health Care Providers," in *The International Conference on Internet Computing*, 2009.