Building a modular YAWL engine with Cumbia

Mario Sánchez*, Diana Puentes and Jorge Villalobos

Department of Systems and Computing Engineering, Universidad de los Andes, Kr 1 Este # 19A-40, Bogotá, Colombia
E-mail: mar-san1@uniandes.edu.co
E-mail: dm.puentes64@uniandes.edu.co
E-mail: jvillalo@uniandes.edu.co

*Corresponding author

Abstract: Nowadays, novel strategies to develop and adapt workflow engines in efficient ways are required in order to have BPM and workflow solutions with the capacity to support frequent changes in the corporate environment. One key strategy is to build new engines by reusing as much as possible from existing components. This requires two things. Firstly, the mechanisms and technologies to build a library of reusable, extensible and adaptable workflow components. And secondly, a platform to integrate those components and form full applications. In this paper we show that Cumbia, being a platform for the development of workflow engines based on the modularisation of workflows according to concerns, suits this task. This is illustrated with YOC, a Cumbia-based implementation of YAWL.

Keywords: modularity; reuse; workflow engine; yet another workflow language; YAWL; Cumbia; model driven development.


Biographical notes:

Mario Sánchez received his PhD from the Universidad de los Andes in Bogotá, Colombia, and from the Vrije Universiteit Brussel in Belgium. He is currently an Assistant Professor at the Universidad de los Andes in the Department of Systems and Computing Engineering. His main areas of research are workflow systems, model driven engineering, and more recently enterprise architectures.

Diana Puentes received her MSc from the Universidad de los Andes in Bogotá, Colombia. Her topics of research were related to workflow languages and workflow engines’ implementation, especially in the case of BPMN and YAWL.

Jorge Villalobos received his PhD in Informatics from the University Joseph Fourier in Grenoble, France. He is currently an Associated Professor at the Universidad de los Andes in Bogotá, Colombia, and serves as the Chair of the Department of Systems and Computing Engineering. His research focuses in enterprise architectures, BPM and workflows, software architecture, requirements analysis and software design.

This paper is a revised and expanded version of a paper entitled ‘A modular approach to build workflow engines’ presented at the 2nd International Workshop on Reuse in Business Process Management (rBPM 2011), Clermont-Ferrand, France, 29 August 2011.

1 Introduction

The efforts to reuse existing assets in workflow environments are frequently studied only from the viewpoint of workflow designers. Moreover, studying the perspective of workflow languages and workflow engines developers reveals critical problems and opens up interesting opportunities. Currently, one of those problems is that building new workflow engines, or modifying existing ones to support new requirements, is expensive (Nutt, 1996). This is in part because engines’ implementations are tightly coupled to a single language, and in part because they are not developed with flexibility and extensibility in mind.

This paper shows how the Cumbia platform offers an effective alternative for designing and developing workflow engines by reusing as much as possible from existing ones. This approach uses the platform as the common base for many different workflow engines. Also, it proposes the creation of a library of reusable concern...
specific workflow languages, which are implemented as Cumbia metamodels, and can be assembled in multiple ways to support different requirements. Cumbia can thus be considered the core element in a product line of workflow engines. This paper also shows that using Cumbia results in engines that are extensible and adaptable to new business requirements.

The following are the main features of the platform that enable the aforementioned things. Cumbia is a model driven platform based on executable modelling, where each workflow specification language is represented with a metamodel (Sánchez et al., 2009b,a). Correspondingly, a workflow specification is represented in Cumbia by a model definition, and it is executed by a component called Cumbia Kernel. To enable this execution, metamodel specifications must cover structural and behavioural aspects, which is achieved by using a modelling abstraction that we have called open objects. Open objects will be discussed in detail in the following sections and they are based on plain objects and on reified state machines that are used to coordinate their execution.

Another central feature of Cumbia is the modularisation of concerns (Sánchez and Villalobos, 2008): for each concern that is relevant for a workflow application, a concern specific metamodel is built thus defining a concern specific language. This strategy results in smaller, composable languages, and brings advantages such as ease of use, development, maintenance, and flexibility (Warner and Kleppe, 2006). Concern specific metamodels are used to describe concern specific models, which are woven together at runtime. This requires the specification of relations between the models, which involves three additional elements of Cumbia: M2CL, M1CL, and CCL [see Section 4 and Rodríguez et al. (2011a,b)]. There have been other works around the topic of workflow modularisation [e.g., AMFIBIA (Kindler et al., 2006), AO4BPEL (Charfi and Mezini, 2006)], but they have limitations that have been solved in Cumbia (Sánchez, 2012).

This paper uses yet another workflow language (YAWL) to illustrate the advantages that Cumbia brings to the construction of workflow engines. YAWL is a workflow language well known in the academic community, whose design was mostly guided by the structure of the control flow. Since other aspects where introduced as complements, their constructs are not as powerful as those for describing the control flow. However, if we were to modify YAWL to improve one of those aspects, we would encounter serious problems because neither the language nor its engine were designed with language flexibility in mind. Hence, introducing changes would require a lot of effort, or even a complete reimplementation of the engine.

The rest of the paper is structured as follows. Section 2 describes the implementation of YOC, an engine for YAWL built on top of Cumbia. The focus of this description is the implementation and extension of the control concern. Section 3 then shows how Cumbia manages extensions to the metamodels and how this was used to make improvements to YOC. Section 4 describes the addition and replacement of concerns. Finally, Section 5 describes how YOC’s implementation was verified and Section 6 presents the main conclusions to the paper.

2 A Cumbia-based engine for YAWL

YAWL (van der Aalst and ter Hofstede, 2006) is a well known workflow language which originated in the academic community but has also been used in commercial applications. Its notation and semantics are common place, and it subsumes the core elements found in most workflow languages. From the point of view of the control-flow, YAWL is very expressive. It supports most workflow patterns (Russell et al., 2006) and it has been frequently used as a case study in workflow research.

Unlike languages where the semantics is informally defined, such as BPMN, YAWL’s semantics is formally specified using extended workflow nets (EWF-Nets), which are an extension to Petri nets, and, at the same time, are also defined in terms of Petri nets. Because of this formality, implementing the language does not require a subjective interpretation of its specification.

Figure 1 The five typical dimensions in YAWL

Typically, five dimensions are involved in a YAWL process:

- **Control.** It is the core of YAWL and defines the tasks in a process and their order of execution. The main objective of the YAWL language is to specify control elements, and all the other dimensions are structured around this one.

- **Application.** This dimension defines the actual tasks to realise in a process. For example, specifying whether an activity must be automatic or must be performed by a human belongs in this dimension, as well as specifying bindings and bridges to concrete external applications (e.g., services). In spite of its importance, this dimension is only represented in YAWL by one element, Decomposition, which determines the specific behaviour of Tasks.

- **Data.** This dimension describes the data used in a process. This includes the inputs and results of the process, and the inputs and results of each task. YAWL’s engine uses XSD schemas to define the structure of data, and uses XPath and XQuery to define what data each task consumes, and how the data produced by each task is transformed and stored.
• **Time.** This dimension describes timeouts and expiration dates for tasks. In YAWL’s engine, these are also defined using low level XML expressions.

• **Resources.** This dimension describes who participates in a process and the policies to assign participants to tasks. In YAWL, this involves a complex procedure that offers the tasks to qualified participants which may accept or refuse the offer, and then assigns the task to one of the willing. The mechanisms to define which participants are qualified to perform the tasks range from very simple (e.g., ‘anyone can do the task’, or ‘user X must do the task’), to very complex (e.g., ‘users with characteristics Y and Z, and which have not participated yet in this case’). At run time, the assignment procedure can be modified or bypassed by the administrator of the process.

2.1 **YOC: YAWL on Cumbia**

The first step to implement YOC was to identify the concerns to support: we started from the five dimensions; then, we assimilated the control and application dimensions because it is not likely that they will evolve independently, and we do not expect to have control models reused independently from application models, and vice-versa. The next step was to design and develop a metamodel for each concern. Figure 2 shows the relations between these metamodels: the one for control can be considered the central one, since the metamodels for time and resources have dependencies towards it. The initial version of YOC did not include a metamodel for the data concern.

![Concern specific metamodels in YOC](image)

Most of the effort went into the development of the control metamodel because the control concern is the most complex and the best documented aspect of YAWL. Figure 3 shows the structure of the control metamodel in YOC, which includes the assimilated application dimension, i.e., Decomposition is included in this metamodel.

The semantics of the elements that YAWL provides to model the control dimension are defined in terms of EWF-Nets, and thus they depend on the transfer and consumption of tokens: Flows define how tokens can be transferred between elements in a process, and elements are only executed when tokens are available for them to consume (as in Petri nets). There are four main categories of elements in YAWL.

• **Nets,** which enclose structured sets of elements. However, unlike processes in BPMN, nets cannot be directly nested.

• **Tasks,** the units of work in a net. Tasks can be **atomic,** and represent something that has to be performed once. They can also be **multiple,** and represent the same action that has to be performed multiple times, in parallel. Tasks are **composite** when they represent sub-nets.

Each Task has a **split behaviour** and a **join behaviour** that determine the interaction of a Task with other elements in the Net. If a task’s split behaviour is that of an **AND-Split,** after the task is completed tokens flow through **all** of its outgoing flows. In the case of an **XOR-Split** behaviour, only one token is produced, and the outgoing flow that carries it is selected by evaluating predicates associated to each one; when one predicate evaluates to true, the corresponding outgoing flow is selected and the process continues. In the case of an **OR-Split,** all the predicates are evaluated and tokens are produced for each outgoing flow whose corresponding predicate evaluated to true. The join behaviour of a task specifies when to enable said task. In the case of an **XOR-Join,** it is only necessary to receive a token through one of the tasks’ incoming flows. In the case of an **AND-Join,** one token must be received through each incoming flow. The case of the **OR-Join** is discussed below. The split and join behaviours fulfil a role similar to that of Gateways in BPMN.

• **Conditions** are elements that can contain one or more tokens, without consuming them. They are analogous to places in Petri nets, and they create some of the most complex synchronisation problems from the viewpoint of implementation. Conditions are so named because they contain a token when a certain condition holds in the net.

• **Cancellation regions** group tasks and conditions which should be **cancelled** when a certain task (located outside the cancellation region) is executed. The cancellation operation removes all the tokens in the region, and aborts the execution of every task in it. Cancellation regions are not easily modelled with Petri nets, and they are one of the main reasons to define YAWL’s semantics using EWF-nets.
2.1.1 Implementation considerations

To implement this metamodel we made four important decisions. Firstly, we left out OR-Joins. The reason for this is that supporting the semantics associated to this construct requires a strong algorithmic effort (as shown in YAWL’s own implementation (Wynn et al., 2005), which is not valuable to illustrate the expressive power of the Cumbia platform. Solutions to the OR-Join problem (van der Aalst et al., 2002) are largely independent of the underlying implementation artefacts.

Secondly, we assumed AND-Join and AND-Split behaviours for every task where these behaviours are not explicitly defined. This does not change the semantics of processes because in YAWL it is mandatory to specify a join behaviour for all tasks with more than one incoming flow, and to specify a split behaviour for all tasks with more than one outgoing flow. The official YAWL editor assumes the same behaviours that we do, and our implementation only makes this explicit.

Thirdly, we introduced a condition in each flow connecting two tasks. This does not change the semantics of the language either, as we are only reversing YAWL’s designers’ decision of hiding these conditions to simplify the layout of diagrams. In terms of Petri nets, our decision is equivalent to forcing flows to connect places and transitions, and disallowing flows connecting two transitions.

Finally, we had to select a strategy to implement YAWL’s coordination model, which calls for atomic updates. Since Cumbia is mostly based on concurrent execution and asynchronous interactions, it was not straightforward to support the synchronisation requirements. This is analogous to the known problem of implementing Petri nets-based workflows. To solve this, we evaluated the three solution strategies proposed in Barril (2002), and then implemented a centralised control system, without locksets. The centralised queue (EventQueue) is local to each process instance and registers which tasks are enabled and can be executed at any given moment. Tasks are removed from the queue when they are no longer enabled, and this guarantees that no deadlocks are caused by tokens assigned to tasks that never execute,

2 guarantees that no inconsistencies happen because of tasks that execute without having the required tokens.
From the point of view of EWF-nets and Petri nets’ semantics, this strategy results in a correct implementation: at most one transition can be activated at any given time; and the consumption and production of tokens happen in different times, but no other actions can occur before the whole procedure is completed.

### 2.2 Tasks and Cancelation Regions

For reasons of space we are going to describe in detail only two elements of the YOC control metamodel, namely Task and Cancelation Region. Every element in a Cumbia metamodel is represented with an *open object*, which means that every element has a class (which in Cumbia is called the *entity*) and an associated state machine. Figure 4 depicts the open object that models Tasks in YOC. On the upper part there is the entity, which holds the internal state of a task and implements its behaviour. The entity also generates events, and these are listed on the lower part of the box.

**Figure 4** The open object that models a Task in YOC (see online version for colours)

On the bottom part of the figure, there is the state machine, which reifies the life cycle of a task. This state machine is composed by four states connected by transitions triggered by events. In this case, all the events are generated by the task itself (they have the mark [ME]) but they can also be generated by other elements in the metamodel (see the state machine of Cancelation Region). Finally, some of the transitions have actions associated: when those transitions are triggered, the corresponding actions are executed. Cancelation Regions are also modelled with an open object in the YOC control metamodel. Figure 5 depicts such open object. We can see, from the structure of its state machine, that a Cancelation Region interacts with other elements in a net. On the one hand, each Cancelation Region has a trigger task which activates it by generating the event *activateCancelation*. When that event is received, the Cancelation Region can go from the state *Init* to the state *Aborting Elements*.

**Figure 5** The open object that models a Cancelation Region in YOC (see online version for colours)

On the other hand, the Cancelation Region also interacts with the elements that it groups: when the transition from *Init* to *Aborting Elements* is triggered, the action *initiateAbortingElementsCR* is executed. That action invokes the method *abortElements()*, which locates all the elements in the region, removes the tokens that they currently hold. If one of the elements is a Task that is currently being executed, its execution is aborted.

These two examples illustrate important aspects of the implementation of metamodels in Cumbia: the reification of state machines, the interaction between elements based on actions and method invocations, the interaction based on events, the relevance of the state machines for the coordination of the execution, and the implementation of elements’ behaviour in the methods of the classes. In http://cumbia.unicolombia.edu.co, there are more details about this metamodel and its state machines.

### 3 Extensibility and flexibility in the control dimension

According to the specification of the language, the documentation of the official engine, and several papers, YAWL has some inherent flexibility, which is represented by decompositions and worklets. The mechanism of decompositions serves to differentiate the concrete tasks in a process: for each task added to a net, a decomposition is selected and it determines the concrete actions that must happen during the execution of the task. However, decompositions are static: at design time, process designers...
M. Sánchez et al.

have to choose the concrete actions to be executed in each task. To counter this limitation, YAWL’s designers introduced worklets (Adams et al., 2006), which are dynamically selected YAWL processes that act as subnets. Using worklets, a dynamic element is introduced into YAWL specifications.

However, these flexibility mechanisms are not enough to support every new requirement that may appear as they are limited to tasks. It is possible to encounter requirements in YAWL-based applications that are difficult to model with the standard language, and would be better supported with modifications to the language. In this respect, the main advantage of Cumbia is offering a platform where modifications to the languages can be introduced with relative ease.

Figure 6 shows the taxonomy of extension mechanisms offered by Cumbia, which can be used to extend or change any engine’s metamodel thus changing the workflow language. These mechanisms vary both in their expressiveness and in their impact to the unmodified elements of the metamodel. This means that some kinds of changes (e.g., replacing a state machine) can radically alter the behaviour of an element, but they usually also require changes in many other elements. Instead, other kinds of changes (e.g., adding an action in a state machine) introduce smaller modifications to some elements’ behaviour and have minimal impact on the other elements of the metamodel.

Figure 6 Taxonomy of extensions in Cumbia

The extension mechanisms offered by Cumbia can be separated into three broad groups depending on what they target. First of all, new open objects can be added to metamodels at any point. Secondly, it is possible to introduce changes that only affect the entities of the open objects. These can modify the interface (including the events it generates), the relationships towards other open objects, and the implementation of the entity itself. Finally, it is possible to extend or replace in their entirely the state machines. All of these mechanisms can be combined to obtain the necessary changes to the metamodels’ structure and behaviour. In the following section we illustrate the application of one of the described mechanisms to YOC.

3.1 Ad hoc subprocesses in YOC

To illustrate how a language and its engine can be extended, we modified the control metamodel of YOC and introduced new constructs to represent ad hoc subprocess. In workflow languages such as BPMN, an Ad Hoc Subprocess is a kind of un-structured process, which groups activities but does not specify their order of execution. Typically, this order is defined at run time and it is necessary to execute once, and only once, every activity in the ad hoc subprocess. To support them, we extended the control flow metamodel and added two new open objects, namely Ad Hoc Region and Ad Hoc Task. The former was based on the old element Cancelation Region, while the latter was based on Task.

An Ad Hoc Region has a similar structure to a Cancelation Region, but different behaviour (see Figure 7). When an Ad Hoc Region is activated, the tasks contained in it become ready to be executed. However, these cannot be normal task, as they are not related to other elements and do not share flows. Therefore, they are special Ad Hoc Tasks. The differences between these and the basic Tasks, are mostly relegated to the actions of the state machine. In particular the code in the action Start processing is different from the original Task, because the order of execution of Ad Hoc Tasks is defined at run time.

Figure 7 The state machine of Ad Hoc Region (see online version for colours)

We cannot present one example for every kind of metamodel extension that is possible in Cumbia. However, this experiment evidences important aspects of the support for extensions offered in the platform: we added new elements into an existing metamodel, and we related these elements with old ones. These relations were not only structural, but they also implied interactions between new and old elements. Finally, the new elements are seamlessly blended with the old ones, and can be used without distinction.

4 Beyond a basic Cumbia-based engine

This section shows two reuse scenarios where Cumbia-based engines are built or adapted with existing
Building a modular YAWL engine with Cumbia

components. The final result is shown in Figure 8: YAWL Time was replaced with a metamodel called XTM; and a metamodel for the auditing concern, LOG, was introduced to work with the control and the resources concerns.

**Figure 8** Metamodels in the extended YOC

The critical features of Cumbia that make these scenarios possible are the decomposition of concerns, the management of multiple metamodels, and the run-time weaving process. The following are the four concrete elements of the approach that support this:

- **The coordination mechanisms** offered by the open objects. They not only serve between elements contained in the same model, but they can also be used between elements in two different model instances. As a result, coordinating multiple models is very similar to coordinating a single one.

- **M2CL**, the language to describe the types of relations that can be established between types included in two or more metamodels (Rodríguez et al., 2011a, 2011b). Since M2CL specifications are external to the metamodels, these maintain their independence.

- **M1CL**, the language to describe the relations between specific elements of two or more models (Rodríguez et al., 2011a, 2011b)). These relations must be instances of the relation types described in an M2CL specification, and they are kept in a specification that is external to the model descriptions.

- **CCL**, the low level language to describe how to alter open objects in order to coordinate their execution (Rodríguez et al., 2011a, 2011b)). While M2CL specifications are written by experts in the metamodels involved, and M1CL specifications are written by the same domain experts that write the models, CCL specifications are automatically generated from those other specifications.

**4.1 Introduce a new concern – auditing workflows**

Not every workflow engine involves the same concerns. Control, time, resources, and data are the three concerns that are most commonly found, and most languages and engines support them to some degree. Moreover, in particular contexts there are other concerns that are equally important (e.g., auditing, billing, or security – authorisation). Because of this, workflow engine developers should be able to select and integrate as many concerns as they require in each case. Similarly, if a concern is not relevant for their cases, they should not be forced to include it.

This section illustrates this flexibility requirement by introducing into YOC some functionalities that do not fit in the concerns that were originally developed. The new functionalities are to support the logging of selected information about workflows’ executions, such as the duration of each task, or their intermediate results, or the names of the people assigned to perform them. For this, a new concern has to be introduced, the auditing concern, and a metamodel has to be designed and implemented for it.

The metamodel developed to support this concern, LOG, is very simple (see Figure 9), but it is powerful enough to support the requirements described. It is even desirable for metamodels to be small: smaller languages are easier to adapt, extend, and maintain (Warmer and Kleppe, 2006), and thus are more likely to be reused. If it later becomes necessary, LOG can be enlarged and improved using the extension mechanisms described in the previous sections.

**Figure 9** Metamodel for the logging concern

There are two open objects in the LOG metamodel, namely Logged Event Receiver and Logger. The former receives and processes the events produced in the other concerns. The latter registers those events in a log file. Models built with this metamodel are then woven to models describing the control, resources, and time concern. This means that elements in LOG models react to the execution of elements in the other models by either capturing events or by receiving method invocations from actions. Since the LOG metamodel is not tied to any other particular metamodel, we can use it to register what occurs in any of the other concerns. For example, we can create log files with the time of execution of each task and the sizes of input data used in each task execution (control concern), and a detailed registry of tasks distribution among employees (resources concern).

**Figure 10** The open object that models a LogEventReceiver in LOG (see online version for colours)
Figure 10 shows the structure of the open object that represents a Logged Event Receiver. Its state machine has a deceivingly simple structure: it only has one state and one transition, but they play a very important role in the execution of LOG models and the synchronisation with other concerns. The transition, which is called registeringEvent, is invoked when the method registerEvent of the entity is invoked, which generates the event eventReceived. The only action initially associated to the transition, writeToLog, registers the occurrence of the event on the Loggers associated to the Logged Event Receiver.

Listing 1 presents a fragment of an M2CL specification that describes how to relate a LoggedEventReceiver from LOG and a Task from YOC. This specification describes a new type of relation called LoggedTask (line 4) by specifying how to instrument the models with the purpose of coordinating their execution. First (lines 8 to 10), a listener is added to a Task to enforce the invocation of the method registerEvent when the task finishes its execution and generates the event exitProcessing. Then, the state machine of the LoggedEventReceiver is instrumented with an action that is added at the beginning of the transition registeringEvent. This action has the sole responsibility of gathering all the data that should be stored in the log. Note that the action should be developed by someone that is interested in composing these specific metamodels (e.g., the person that wrote the M2CL specification). Therefore, this developer should know both metamodels and their implementations beforehand.

Listing 1  Fragment of an M2CL specification

```
1 namespace logExample
2 import YOC , LOG
3
4 composite LoggedTask
5   use YOC . Task t
6   use LOG . LoggedEventReceiver ler
7   when t->exitProcessing
8   call ler:.registerEvent()
9 end
10
11 extend ler
12   new action gatherData:
13     in registeringEvent 0
14     yoc.log.actions.GatherTaskData
15 end
16
17 // any number of relation types can be described
18 // in the same specification file
19 // ...
```

Listing 2 presents a fragment of the M1CL specification that describes how to relate and coordinate a YOC model with a LOG model. It specifies that the task called task1 in the control model should be composed to the LoggedEventReceiver called receiverA in the log model, and specifies the type of relation (LoggedTask). Since this M1CL specification makes use of the relations defined with M2CL, it only has to provide pointers to elements in the models.

Listing 2  Fragment of an M1CL specification

```
1 namespace logExample
2 load YOC . ControlModel1 cm1 ,
3 LOG . LogModel1 lm1
4
5 LoggedTask loggedTask_1
6 t: cm1 . task1
7 ler: lm1 . receiverA
8 end
```

Considering that every concern in Cumbia is developed using open objects, and given that the weaving mechanisms are independent of the metamodels, we can add new concerns in any moment, as they become necessary. To do this, we only need to develop the corresponding metamodel, and create the necessary M2CL and M1CL specifications.

Another characteristic to highlight of this approach, is that the applications created by adding new metamodels maintain the properties of the base applications. Therefore, we can keep on adding, modifying or replacing concerns. Also, if a certain concern is no longer necessary in an application, we can remove it in the same way as it was added.

4.2 Replace a metamodel – handle complex time restrictions

The modularisation of concerns and metamodels makes it possible to have concern-level flexibility. Therefore, any metamodel can be replaced with another one if the former does not support all the requirements. This section illustrates this in a concrete scenario: starting from the basic implementation of YOC, we introduced changes into the time concern without affecting the other ones.

YAWL provides a few elements to describe time restrictions, but they are not very expressive and they are basically limited to timeouts. However, in many contexts the time restrictions associated to workflows can be quite more complex than timeouts. Therefore, it is reasonable to improve YOC by giving it more powerful capabilities to express time restrictions. Luckily, time is such an important concern for workflows that we already had XTM, a very expressive metamodel to describe advanced time restrictions in workflows. XTM is independent of the control flow metamodel, and we have used it before with BPMN and BPEL, among others.

With respect to describing time restrictions, XTM is much more expressive than YAWL. This means that many things that can be said with XTM cannot be expressed in YAWL, or can only be expressed in very complicated ways. This is evident from the number of types of time restrictions supported in XTM (17) which go from very simple (e.g., restrict the duration of a task) to very complex (e.g., make the duration of a task depend on the time elapsed between a series of events). In http://cumbia.uniandes.edu.co, there is a complete description of the metamodel of XTM and of the time restriction patterns that it supports.
Since XTM was not developed to be used with YAWL or to be used in YOC, there are no elements in that metamodel that tie it to the YOC control metamodel. Furthermore, YOC’s implementation of YAWL’s control concern is oblivious of the time concern and of the metamodel implementing it. Therefore, replacing the original YAWL Time metamodel with XTM was done without an impact on the other concerns. Two actions were required to integrate XTM into YOC. Firstly, the component that knows how to execute XTM was introduced into YOC’s architecture. This was a very simple operation thanks to its modular design and only required the deployment of a library and the re-configuration of a connector. The second action was writing a new M2CL specification to describe the relations between the original control metamodel and XTM. Since this M2CL specification is external to the metamodels involved, then there are no hard dependencies between them, and they can be replaced with relative ease.

This scenario shows the value in Cumbia of a library of composable languages or metamodels, which can be reused in different applications. In the example presented, XTM was an existing metamodel in this library, and it was possible to integrate it with YOC because it did not have any explicit dependency towards another metamodel. In a similar fashion, other metamodels can be easily selected and integrated based on the particular requirements for a domain.

5 Validation and testing

After a new workflow engine is built, the most important question to ask is whether the engine truly implements the semantics of the language. Considering that YAWL has been formally specified, it would be great to have a formal proof about the correctness of the engine’s implementation. Unfortunately, it is very cumbersome to build such a proof, and it has not even been done for the reference YAWL engine. In consequence, language implementations are usually validated by means of test suites.

Since YOC was built on top of Cumbia, to test the implementation we used the Cumbia test framework (CTF) (Sánchez et al., 2011). The CTF is a collection of tools to create testing environments for engines built on top of Cumbia. The CTF solves several issues. The first one is that of concurrency management, because the CTF performs offline analyses of execution traces. The second one is that of non-determinism, which is solved by running the test-suites multiple times while introducing random delays that try to bring light to errors in the metamodels. A further issue is that of comparing the behaviour implemented in the engine with the semantics of the languages. In typical cases, this is addressed using assertions over the structure of the execution traces, but it is also possible to plug-in ad hoc interpreters. Finally, there is the issue of describing the test-cases using languages that are expressive enough to stress all the corner cases of the implementation. In the CTF, this is achieved by separating the definition of the structure of the testing scenarios and the definition of their behaviour. While the former is usually represented by model definitions, the latter is described with animation programmes that specify when the framework must generate certain stimuli to keep the models running.

Using the CTF we created a testing environment that was tailored for YAWL. Unfortunately, it is not possible to use the same environment to test other engines not based on Cumbia because their elements cannot be instrumented as easily. After having a working testing environment, we created a test-suite that was structured using the description of YAWL: for each element in the language, we created many test scenarios. In those, we attempted to include the relationships and interactions that the selected element can have with any of the other elements in the language, and we documented the expected behaviour for each scenario. Table 1 presents the specification of one case study out of more than 50 that were created to test the control dimension. It should be noted that the test-suite contains both sound and non-sound processes, and that each test case may involve any set of dimensions besides that of control.

Table 1 Sample specification of a test case (see online version for colours)

<table>
<thead>
<tr>
<th>Test Case 5b, Deferred choice continued with XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification ID: test5b_yawl, Net ID: Test5b</td>
</tr>
</tbody>
</table>

![Diagram of a test case](image)

**Execution description**

This process starts with the execution of Initial Task. Afterwards, the process reaches the condition ‘Deferred Choice’ and both Upper Task and Lower Task are enabled. However, it is not possible to execute both tasks: someone has to choose which one of them to execute. As soon as one of them is started, the other one is removed from the queue of pending tasks. After either Upper Task or Lower Tasks have finish, Final Task can be executed and then the process can finish.

**Notes**

This process is sound.
The rest is identical to Test Case 5a.

**Data**

There is a net variable called ‘intermediateValue’. Upper Task sets ‘intermediateValue’ to ‘output upper task’, while Lower Task sets ‘intermediateValue’ to ‘output lower task’. Final Task does not modify any variable.

**Resources**

All activities are set to be assigned randomly to someone with the role ‘testUser’.

**Time**

There are no time considerations in this test case.
The described test-suite was very useful during the design of
the metamodel, because some coordination problems
between state machines are very difficult to discover by
manually running the models a few times, or, even worse,
by doing a manual review of their design. Therefore, the
design and implementation of the metamodel (including the
design of each state machine and the implementation of the
entities and actions) was not considered completed until all
of the tests were run successfully.

Other valid questions about a specific workflow engine
involve comparisons with other ones that support the same
language. For example, someone that is planning to adopt
a certain language would probably want to know which of
the available engines is ‘faster’. In the case of YOC, it
would be natural to compare it to the official YAWL engine.
Unfortunately, it is difficult to compare them in a complete
and fair way because each one offers a different set of
features. For instance, while an engine that runs inside a
JEE container will normally be a lot slower than one that
runs directly on top of the JVM, the former provides a
richer set of features. In consequence, we have avoided
most comparisons between engines built on top of Cumbia
(such as YOC) and engines built using more traditional
methods. The only thing that we fully analysed, as it was
discussed previously, is whether language semantics are
correctly implemented.

6 Conclusions

In this paper, we have addressed the issue of building
workflow engines by reusing previously developed modular
elements. The solution proposed is based on Cumbia, a
platform for the development of engines which supports
the modularisation of languages and workflows based on
concerns. This, together with powerful composition
mechanisms that focus both on structure and behaviour, has
made possible the construction of a library of Cumbia-based
concern specific workflow engines.

The ultimate goal of this work is to establish a product
line where new workflow engines are built mostly by
assembling components developed for previous ones. This
paper has illustrated the means proposed to achieve this
with an example based on YAWL. It should be noted that
the level of reuse achieved is made possible not only by the
composition mechanisms offered by Cumbia, but also by its
mechanisms for adaptation and extension.

Building various engines on top of the same platform
has further advantages. Complementary tools, such as
monitoring applications and the testing framework, can be
language agnostic and be reused with several workflow
languages. Also, engines are built on top of an existing
and tested platform. This reduces the implementation effort
and allows more focus on the language itself. Finally,
improvements to the platform are made once but they
benefit a large number of applications.

We are currently advancing this research in two
directions. We are working on more case studies to improve
our metamodel library which already comprises engines for
BPMN, BPEL, IMS-LD (a language for the description of
workflows in the e-learning domain), PaperXpress
(a collaborative workflow-based tool to support writing
efforts), and other domain specific workflow languages. On
the other hand, we are working on the design of composable
editors that should complement at the graphical/design
level, what is already done at the behaviour/run-time level.

References

Adams, M., ter Hofstede, A., Edmond, D. and van der
implementation of dynamic flexibility in workflows’,
in On the Move to Meaningful Internet Systems 2006:
CoopIS, DOA, GADA, and ODBASE, Lecture Notes in
Computer Science, Vol. 4275, pp.291–308, Springer-Verlag,
Berlin/Heidelberg, Germany.

(Eds.): Petri Nets for Systems Engineering, pp.417–431,
Springer-Verlag, Berlin/Heidelberg, Germany.

languages’, in On the Move to Meaningful Internet Systems 2006:
CoopIS, DOA, GADA, and ODBASE, Lecture Notes in
Computer Science, Vol. 4275, pp.183–200, Springer-Verlag,
Berlin/Heidelberg, Germany.

a meta-model for the integration of business process
modelling aspects’, in The Role of Business Processes in
Service Oriented Architectures, Dagstuhl Seminar
Proceedings, Vol. 06291, Internationales Begegnungs- und
Forschungszentrum für Informatik (IBFI) Schloss Dagstuhl,
Germany.

systems’, Distributed System Engineering, Vol. 3, No. 4,

model composition – a multi-level approach’, in ACM
Symposium on Applied Computing (SAC ’11), pp.877–884,
ACM, NY, USA.

Rodríguez, C., Sánchez, M. and Villalobos, J. (2011b)
‘Metamodel dependencies for executable models’, in TOOLS
Europe 2011, Lecture Notes in Computer Science, Vol. 6705,

Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P.
A Revised View, BPM-06-22, BPMcenter.org.

build workflows using aspect-oriented concepts’, in AOSD
Workshop on Aspect-oriented Modeling (AOM’08), pp.25–30,
ACM, NY, USA.

‘A state machine based coordination model applied to
workflow applications’, Avances en Sistemas e Informática
Vol. 6, No. 1, pp.35–44, Universidad Nacional de
Colombia – Medellín.

Sánchez, M., Jiménez, C., Villalobos, J. and Deridder, D.
(2009b) ‘Extensibility in model-based business process
engines’, in TOOLS Europe 2009, Lecture Notes in Business
Information Processing, Vol. 33, pp.157–174, Springer,
Berlin/Heidelberg, Germany.


