Executable Models as Composition Elements in the Construction of Families of Applications

Jorge Villalobos[1], Mario Sánchez[2,3] and Daniel Romero[4]

Universidad de los Andes, Bogotá, Colombia

Abstract

In control-based applications, there is a central coordination element that leads the cooperative execution of several active entities, and provides a way to integrate them to achieve a common goal. Unfortunately, the contexts where control-based applications are used tend to evolve frequently, and they are not always flexible enough to cope with these changes in a timely and cost effective manner. In particular, they present problems with the coordination and composition of evolving and new elements.
In this paper, we propose a strategy to build families of control-based applications by lessening the problems related to evolution and adaptation. This is achieved using an executable language for modeling processes, Cumbia-XPM, and a coordination model that supports the modeling language. The latter is composed by *open objects*, which are objects that expose their internal state using a state machine that abstracts their life-cycle. These state machines provide also an event-based mechanism for coordination. Because of this underlying model, Cumbia-XPM is very extensible and can be used as an intermediate model that supports the execution of the control perspective of higher level languages. Using a similar strategy, the coordination model can be used to build models that represent other perspectives that also participate in control-based applications.

Keywords: Objects, State machines, Composition, Coordination, Workflows, BPM.

1 Introduction

Nowadays, a growing number of contexts are taking advantage of the usage of control-based applications. In these applications there is a central coordination element that leads the cooperative execution of several active entities, and provides a way to integrate them to achieve a common goal[10]. This central element is what we call the control component. It comprises and executes entities that are part of the control-flow perspective, that is, elements that describe tasks and their execution ordering[15] to form processes. Control is complemented by entities from

[1] Email: jvillalo@uniandes.edu.co
[2] Email: mar-san1@uniandes.edu.co
[3] Supported by the VLIR funded CARAMELOS project: http://ssel.vub.ac.be/caramelos/
[4] Email: da-romer@uniandes.edu.co

other perspectives, such as the data perspective, the time perspective and the resource perspective. Each context where an application is used imposes requirements about the perspectives involved. For instance, an application to support distributed software development processes should handle elements from a domain of resources (roles, people, etc.): they participate in the enactment of the processes and are responsible for its tasks.

The usage of control-based applications has surged because of the many advantages they offer. Like in other situations where coordination models are central, modularity is a key advantage here: it promotes reuse and eases the creation of new processes that achieve new goals. For instance, several learn-flows can be designed for an e-learning application, reusing the same elements, but accomplishing radically different goals. The ability to separate the order of execution of tasks from the actual tasks performed contributes also to the integration of heterogeneous components. As an example, a BPEL engine allows the composition of web-services to create more complex services. Another example are workflow applications that allow the interaction of humans and machines, and are able to route and transform the data produced in each step of the process. Unfortunately, the flexibility offered by control-based applications sometimes is not enough to accommodate certain requirements.

Because of their nature, the contexts where control-based applications are used tend to evolve frequently. For instance, in a financial company is common to have changing business rules, processes which are refined, and new systems and tools that join the application stack and have to be integrated. Control-based applications, however, are not always flexible enough to cope with these changes in a timely and cost effective manner. For instance, in some cases it is necessary to introduce changes in strange ways that affect the usability and maintainability of the applications; even worse, in some situations the only possibility is to built new applications from scratch. In an ideal situation, it should always be possible to easily build new applications as evolution of existing ones. This would lead to the creation of families of applications with common elements and used in similar, but not identical, situations.

In this paper, we propose a strategy to build families of control-based applications, by lessening the problems related to evolution and adaptation. The central point in our strategy is to tackle a critical problem of those applications: coordination of evolving elements in a process. In order to have applications that can be adapted and used to create other applications, coordination needs to have a central role: old elements should be able to interact and be coordinated with elements thrown in as the answer to new requirements. It is important that these improvements are achieved without a big impact in the original applications, and without closing the possibilities to further evolution.

Our strategy to construct control-based applications is based on two main elements: an executable language for modeling processes, and a coordination model, which supports the modeling language. This language, called Cumbia-XPM (eXtensible Process Modeling), was designed to be used in different contexts: for instance, it can be used as an intermediate model that supports the execution of higher level languages. This ability comes from extension mechanisms that can accommodate the structure and execution semantics of other control-based languages. The extensibility of Cumbia-XPM comes from the fact that it is implemented on top of *open objects*, our coordination elements. In this paper we show how these elements are structured and how we can use them in languages such as Cumbia-XPM, to build executable and extensible models. We focus specifically on control, but the same ideas can be applied to other perspectives.

Section 2 of this article presents in detail the executable model and the open objects. Section 3 presents the elements of "Cumbia-XPM" and shows how its execution is supported by open objects. In section 4 we present the implementation and testing of a Cumbia-XPM engine. The next section presents related external work, followed by a brief description of our onging research. This includes the usage of open objects to model elements form other perspectives, and the support to high-level languages offered by Cumbia-XPM. Finally, the conclusions of the paper are presented.

2 Open objects and executable models

The main goal of this section is to present open objects structure, along with the mechanisms that can be used to compose and coordinate them to create executable models. This section also presents the extension mechanisms available to adapt open objects for usage in different situations. Using the terminology of [3], a *component model* based on open objects is described with its *composition techniques* and coordination mechanisms. We have also defined the corresponding *composition languages*, but they are beyond the scope of this paper.

In order to show how open objects are composed and coordinated, this section uses a very simple scenario. The main element in this setting is a control that expects to receive a signal with some frequency; if the signal is not received when it is expected, a time-out is generated and it is necessary to execute some action to verify and correct the problem. Eventually, it could be necessary to include an external monitoring system. Using plain objects, a possible solution could be the one shown in figure 1. A class called *Control* has the methods `ack( )`, to mark the arrival of the signal, and `start( )` to encapsulate the main logic of the control: it runs an internal loop that verifies if the signal has already been received or if its available time has finished. In the event of a time-out, the method `execute( )` of the class *Action* is called to verify and correct the error.

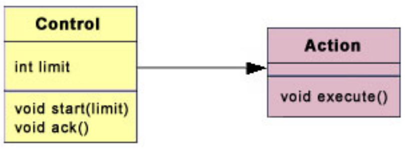

Fig. 1. Solution using objects.

The main difficulties in this scenario are the open nature of the correction action (it is not precisely defined what must be done to correct the problem), and the definition of time (it could be based on execution cycles of the machine instead of absolute time). The possible inclusion of an unknown monitoring system can also be a source of problems and incompatibilities.

2.1 Composition and coordination elements (open objects)

One of the main advantages in the object oriented paradigm is the capacity to materialize in a model the elements of a problem, their behavior, and their relations. Object-based models usually replicate the structures of the problem domain, using object attributes to recreate relations and method calls to model interaction. In order to take advantage of the capacity of building isomorphic structures to the problem domain, our composition elements are based on objects, but have some additional features that expose their internal state. That is why we call them "open objects".

In the traditional object-oriented paradigm, a state of an object is a particular combination of values of its attributes. To know the state of an object, it is necessary to call its methods. The number of states reachable by an object depends on the values that its attributes can have. However, most of the time, the elements that interact with an object are interested only in a subset of its reachable states. For these other elements, some of the possible states of the object can be grouped together to create a simpler abstraction of its life cycle. This does not mean that attribute-based states should be eliminated. Instead, these new, broader states can be materialized in an external state machine. For instance, in the case of a counter from 0 to 100, one possible abstraction can identify only three different states: "Stopped," "Counting," and "Finished." This reduction in complexity simplifies monitoring and coordination. A state machine can publish notifications when a state change occurs in order to coordinate other elements, provided that its state is always consistent with the attribute-based state of the object.

An open object is composed of an *entity* (a traditional object with attributes and methods), a *state machine* associated to the entity, and a set of *actions* (this section later explains the model to describe the state machines and their associated actions). The interface of an open object is based on the interface of the entity, improved with the methods needed to navigate the state machine. In a traditional structure based on objects or components, the ability to compose and coordinate is limited by the methods defined in the interfaces. It is not possible to modify the behavior implemented inside a method. In the case of a structure based on open objects, it is possible to compose and coordinate several elements using the state machines as handlers for the composition, thus offering a coordination mechanism more expressive than the traditional.

To keep the model in movement, a mechanism based on events and action calls is used. It allows synchronization of the state machine with its entity to keep a consistent state between the internal, attribute-based state, and the current state of the state machine. When a method of the entity that changes its internal state is called, events are generated and consumed by the state machine, which moves from one state to the next accordingly to the change in the entity. This mechanism can also be used to coordinate several open objects, since state machines can also be moved by events generated by other open objects' entities and state machines. Each time a state machine moves from its current state to the next, events are generated to show that the original state is abandoned, the processing of a transition starts, the processing of a transition finishes, and a new state is reached. Each transition in a state machine connects two states and has an associated event. To process events, a search is made for a transition that starts in the current state and has an associated event similar to the one being processed. If such a transition exists, then it is used, and the state machine changes its current state. Otherwise, the event is dropped and it is never processed again. Events received by a state machine are stored in a queue until they are processed.

Events expected by a state machine are described with an expression of the form [**ELEMENT**]**eventType**, where [**ELEMENT**] describes who is expected to generate the event, and **eventType** specifies the particular kind of event expected. To ease reusability of state machines, **ELEMENT** is not the identifier of a specific element, but a relative reference that can be used to locate it. For example, if an open object in a hierarchical structure has an event described as generated by [**PARENT**], then the generator of the event can be located ascending in the hierarchy. If an event is described as generated by [**ME**], then the generator is the same open object that owns the state machine.

The last elements in this model are actions associated to transitions, which are executed in a synchonized way when a transition is processed. Actions add semantics to state's changes like calling object methods, communicating to other applications, and generating further events to move other open objects.

The interaction of open objects based on state machines' events is a powerful mechanism to solve coordination requirements in a model. It is a very expressive way to define synchronic or asynchronic coordination rules. Since there are several handles with very fine granularity available to make the composition, it is possible to define coordination rules with a lot of precision. Furthermore, these coordination rules can be defined externally to the open objects, and can be modified even at execution time.

2.2 Composition scenario with open objects

A possible solution for the scenario is now shown using a pair of open objects. Figure 2 shows the structure of this new solution and its main two elements, *Counter100* and *Control*. The open object Counter100 is a counter that is able to count to 100. The method `increment( )` adds one to the current value and the method `reset( )` gets the value back to 0. Right now, the limit of this counter is fixed at 100. The other open object is Control, and has the methods `ack( )` to notify the arrival of the signal, and the method `start( )` to initiate the reception of events. Besides being implemented using open objects, one further difference in this solution is that an external element controls the time by calling the method `increment( )` in Counter100.

Each open object has an entity and a state machine with states and transitions that were specifically designed for the scenario. The state machine for the open object Counter100, depicted in figure 3a, has three different states: "Stopped" (the value is 0), "Counting," and "Finished" (when the value is 100). All the transitions in the state machine depend on methods generated by the entity itself (the generator is always **ME**). The method `increment( )` generates an event called incremented.

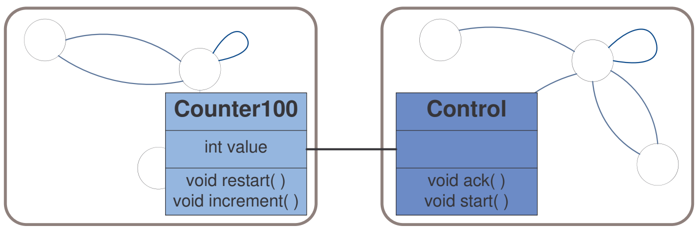

Fig. 2. Open objects in the scenario.

If the counter reaches the limit, then an event called finished is generated. When `reset( )` is called, the event reset forces the state machine to go back to the state "Stopped." For now, the counter cannot be restarted after reaching its limit.

The state machine of the open object Control (figure 3b) has four states: "Inactive" (the control has not been started), "Waiting" (waiting for the signal), "Restarting" (the counter is restarted), and "Timeout" (available time has run out). This state machine shows several of the composition features offered by the model. Some transitions have associated events generated by the Control itself (the ones with [**ME**]) and others depend on events generated by the Counter100 (those marked with [**COUNTER**]). This state machine has an action associated with the transition that goes from the state "Waiting" to the state "Restarting," which calls `restart( )` in the counter whenever the transition is taken. The same mechanism of actions is the one used to define what has to be done when a timeout is generated. We have not included this action in the diagram to highlight the fact that it can be configured very late in the life-cycle of the open object.

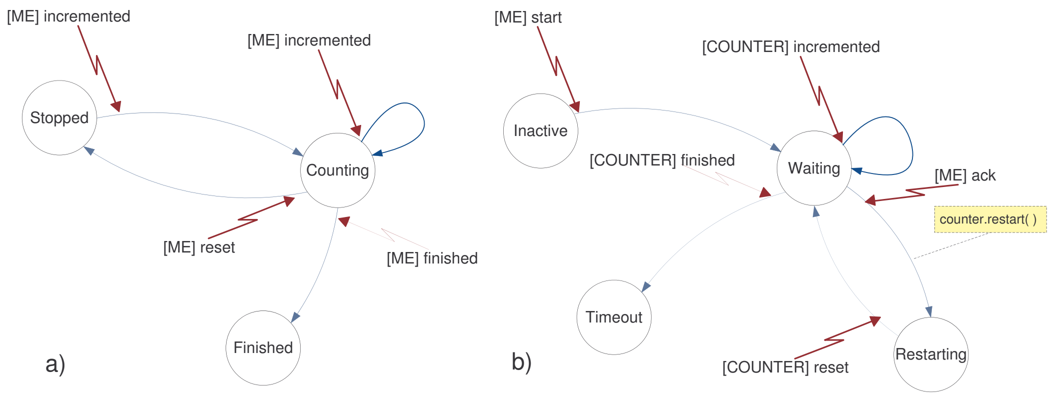

Fig. 3. State machines of the open objects a) Counter100 and b) Control.

Besides showing how events produce coordinated movements of the state machines, this scenario also exemplifies other details of the model: in the first place, state machines are the only elements in our model that consume events, but they can be generated by state machines and by entities; finally, methods of entities can be invoked by elements external to the model (other elements in the application), and by other open objects.

This scenario has shown a small system based on open objects that solves a problem with composition and coordination issues. However, it is not always possible to know beforehand all the interaction cases where an element will be used. This creates the need for extension mechanisms that allow the definition of new elements and the adaptation of old ones.

2.3 Extension mechanisms

Additionally to the definition of composition and coordination strategies, open objects also include three different extension mechanisms that give flexibility to the model. Although traditional mechanisms have points of flexibility like interfaces in components, there are usually limitations in what can be changed of the interaction semantics between elements. Most of the time it is necessary to change the implementation; furthermore, if it is necessary to connect something totally unexpected, like a monitoring system, it is very likely that it will be necessary to change interfaces. The extension mechanisms that the open objects model provides were designed to address different cases and have specific expression capacities that determine how much of the behavior of the elements can be modified. Two of the mechanisms allow the extension of behavior, without modifying the implementation of the entities.

The least expressive mechanism, but the easiest to use, modifies the actions associated with state machines. This method, called "simple extension," can be applied to the definition of an element and affect all its instances, or it can be applied at runtime to affect a single instance. In the scenario, this is the adequate mechanism to specify the action that is executed when a timeout occurs.

Another way of extending the model is by modifiying the structure of a state machine, either by adding or removing states and transitions. This extension mechanism is much more expressive. In the scenario, it allows the Counter100 to be restarted after reaching its limit. This mechanism allows deeper changes to the model, but the programmer is responsible for guaranteeing the proper behavior of the element in relation with the other elements.

The last extension mechanism allows the creation of new open objects by specializing existing ones. This mechanism requires modifications to the implementation of the entities to change the part of behavior of open objects that is not expressed with a state machine. In the scenario, this mechanism is needed to change the limit for the counter that is defined right now inside the implementation of the method `increment( )`.

2.4 Using open objects to build families of applications

Our strategy to construct families of applications is based on the usage of executable models for each perspective (control, resources, data, etc.) that appears in an application. These models are constructed using the elements defined in a specialized metamodel: each instance of a model represents one of the perspectives in a solution. This facilitates the construction of families of applications, because it allows the independent development, adaptation and evolution, of each perspective and its later composition.

To start a family of applications it is necessary to have a basic set of metamodels, which support the fundamental requirements of each perspective. The space of applications that can be constructed by extending those metamodels, or adding new compatible ones, depends on the capacity of adaptation of each metamodel:

with more flexible metamodels, the family of potential applications gets bigger. An important limitation to the evolution of metamodels is posed by the need to keep coordinated and synchonized several perspectives. Because of this, each metamodel should provide powerful mechanisms of extension and the ability to flexibly describe coordination.

The usage of open objects to materialize the elements of the metamodels offers the features required. When open objects are used, it is possible to specify explicitly the structure and behavior of metamodel elements, including their interaction and coordination. Since these relationships are externalized, they are easier to manipulate and modify for evolution. Additionally, the extension mechanisms offered by open objects confer to the metamodels the extensibility capabilities that are required to build families of applications.

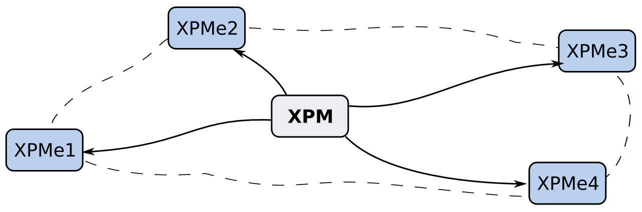

Fig. 4. Extensions to Cumbia-XPM.

In the case of control-based applications, the metamodel that is used to represent the perspective of control has a central role in guaranteeing the capacity of adaptation of the family. We defined a metamodel called Cumbia-XPM, which is used to model the control perspective in our applications. Figure 4 summarizes the idea of extending a metamodel, using Cumbia-XPM as an example. In this case, we have four sample extensions represented (XTMe1 to XTMe4), and each provides different capabilities to the applications that use them. The effort associated to creating each extension varies, like the distance between the base metamodel and each extension varies in the figure. The next section explores this metamodel and shows its elements, its features, and its applicability to the control domain.

3 Cumbia-XPM metamodel

Cumbia-XPM is a metamodel expressed in terms of open objects, designed to describe the control component of applications. The following explanation of the elements that are part of this metamodel, and of the process we used to find them, serves also to show some features of open objects that facilitate both the construction of the control component, and its adaptation to new contexts.

To find the elements for the metamodel, the first step was to study several existing languages (XPDL[5], BPEL[6], BPMN[7], IMS-LD[8], SPEM[9], and others) that are used in contexts like business process modeling, workflows, e-learning,

[5] XPDL Specification http://www.wfmc.org/standards/xpdl.htm
[6] BPEL Specification http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
[7] BPMN Information http://www.bpmn.org/
[8] Learning Design Specification http://www.imsglobal.org/learningdesign/
[9] Software Process Engineering Metamodel http://www.omg.org/technology/documents/formal/spem.htm

and application orchestration. In each language, a set of elements that is used to express control was identified and included in an initial set of common control elements. Our goal was to find a small set of elements that could be used in a lot of different contexts. We were not interested in having a huge set that posed a high risk of becoming difficult to understand, use, and implement. Because of this, we progressively eliminated specific elements and introduced more general ones until we had a reasonable sized set. Since we knew that these elements were going to be represented using open objects, this reduction took advantage of the extension mechanisms: every specific element eliminated had to be obtainable by extending one of the remaining elements. The result of this was a small set of elements (see figure 5), which can be composed to form complex structures and can be extended to express the same things (structures, actions, dependencies, etc.) as the original languages.

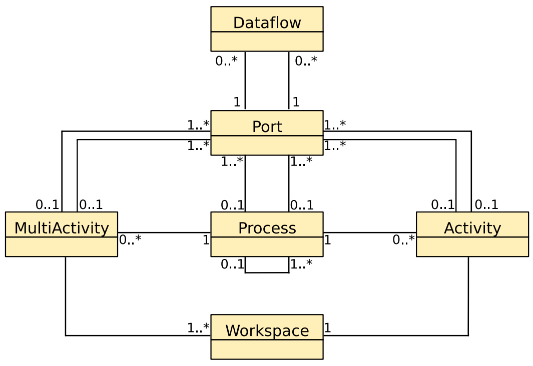

Fig. 5. Cumbia-XPM Metamodel.

The control perspective of an application specifies a set of tasks that have to be executed in a precise order to achieve a specific goal [5]. This set has a hierarchical structure and, in the case of Cumbia-XPM, it has what we call *Process* at its root. To be executed, a process needs some entry products that are consumed when tasks are executed. These products are called initial data and are received through one of the entry ports of the process. When that process finishes its execution, it generates a set of data that can be recovered through one of its exit ports. To model this, we have defined an element called *Port* that can play the role of entry or exit port. In both cases a port is given a set of data that has to be delivered to someone who needs it: in the case of an exit port of the process, the data is picked up by someone or somebody external to the process; in the case of an entry port, the data is picked up by one of the elements that is part of the internal structure of the process.

Executing a process means performing in order a set of either simple or complex (sub-processes) tasks. Each task of a process requires also initial data and produces results, thus tasks also need entry and exit ports. The internal structure of a process defines the execution order of the tasks and depends on *Dataflows*, which connect ports. A dataflow connected to an exit port of a task receives the results that it produces and makes them flow to an entry port of the next task, carrying along both data and control. Figure 6 depicts a sample process with a structure of ports and dataflows that clearly defines the control-flow for the process. In this image it is possible to see how splits, joins and loops can be achieved. For example, after Coding the activities WriteTests and ReviewCode are executed in paralel. ExecuteTests can be executed after the completion of CorrectCode and WriteTests. CorrectCode2 has two possible outcomes: one enables a new execution of ExecuteTest, while the other terminates the process.

The simple tasks that are executed within a process can be modeled with *Activities* and *Workspaces*, two of the elements available in Cumbia-XPM. An activity controls the beginning of the execution of a task and is responsible for managing the data required and the data produced. Workspaces are encapsulated inside activities and are responsible for executing specific tasks. A workspace interacts only with the enclosing activity, which provides the data needed for its execution: when the data reaches one of the entry ports of the activity, it is given to the workspace to start its execution; when the workspace finishes its work, it generates an event, which the activity uses to pick up the data, reset the workspace (so that it can be executed again), and finally put the data to one of the exit ports. When one of those exit ports becomes full, the flow of control and data continues through the dataflows.

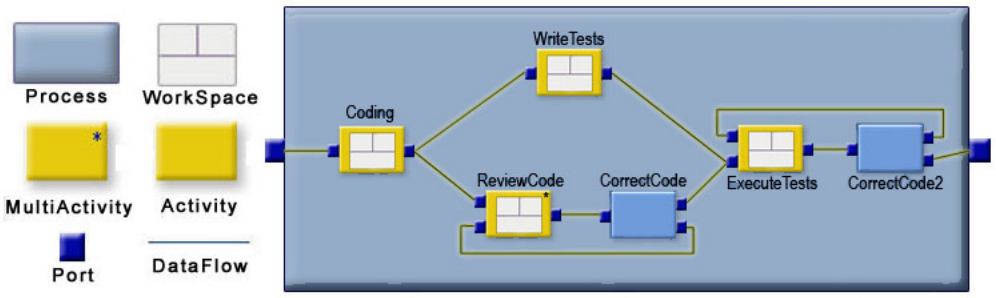

Fig. 6. Elements of Cumbia-XPM and sample process.

In the sample process there are three activities: Coding and WriteTest are activities expected to be made by a user; ExecuteTest is expected to be executed automatically by a workspace specialized in testing. Since Cumbia-XPM is based on the open objects' model, its extension mechanisms (simple extension, adaptation, and specialization) can be used. For instance, to build a system that orchestrates applications, workspaces can be specialized to give them the ability to invoke web-services. On the other hand, two of the tasks (CorrectCode and CorrectCode2) are sub-processes but their internal structure is not shown.

There are cases where it is necessary to execute several instances of the same activity in parallel. Cumbia-XPM has an element called MultiActivity to achieve that. Every time a normal Activity gets data from its entry ports, the corresponding workspace is executed once. In a MultiActivity several instances of the workspace are created and executed in parallel when the initial data required is received. The number of workspaces executed can be defined during the design of the process, or during runtime, and new workspaces of a MultiActivity can be created in a dynamic way. In the sample process, several people execute the activity ReviewCode at the same time, but the exact number is decided during the execution of the process.

To clarify how the interaction between elements of Cumbia-XPM proceeds, we present a simplified scenario that includes only three elements: an Activity that is going to be executed; an entry-port of that activity, which will receive the data that is needed to start the execution (entry-data); and a Workspace that is inside the activity and will be responsible of executing a specific task using the entry-data. The action in this scenario begins when the entry-port receives the data it was expecting; the activity then picks the entry-data, feeds and activates the workspace, and waits until it finishes its execution. An activity can have several entry-ports which can receive data concurrently; however, when one of those ports gets full, the activity waits until the workspace finishes its execution before verifying if there is any other full entry-port.

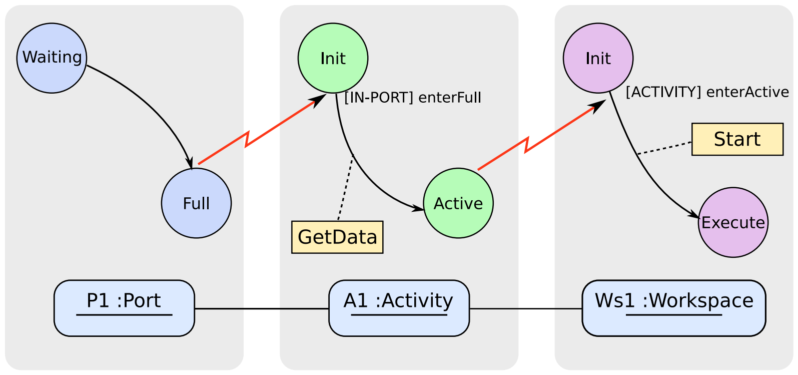

Fig. 7. Partial interaction between a port, an activity and its workspace.

The three state machines depicted in figure 7 correspond to fragments of the state machines of a Port, an Activity and its Workspace. These three state machines were designed to be composable: some of the events that they generate are expected by another state machine to continue its execution. We now describe the interaction between those state machines, that starts when the port receives data.

(i) The port P1 receives entry-data and takes the transition from state Waiting to state Full.

(ii) The state machine of activity A1 receives the event *enterFull* and takes the transition to state Active, executing the **getData** action, that retrieves the data from the port and gives it to the Workspace.

(iii) The state machine of workspace Ws1 receives the event *enterActive* generated by the Activity's state machine. The transition to state Execute is taken, executing the action **start**, that initiates the execution of the workspace.

This interaction sequence is repeated every time an activity instance starts its execution. Although the interactions between pairs of elements are as simple as the one presented, the complete set of state machines that interact in a single process instance offer a very fine granularity level: the execution of a relatively simple process can be supported by the interaction of tens of state machines. In a process as simple as the one depicted in figure 6, there are 40 state machines (roughly 180 states) synchronized during the entire execution of the process: there is a state machine per instance of an element, and each one of those has between three and twelve states.

An important feature of Cumbia-XPM is its ability to model *dataflow* machines, where the main responsibility is to control the flow of data consumed and generated, as well as *workflow* machines, where the main responsibility is to control the order of execution of the tasks. When implementing a *dataflow*, the three implementation strategies described in [13] can be used: it is possible to combine the data and control flows in a single dataflow, to use different channels for tokens and for data, or to use a central repository for data. When implementing a *workflow*, simple pieces of data called tokens can be used. These tokens, which flow as normal data, are consumed and produced by activities; however, just the activities that hold tokens can be active in a given moment.

Another important feature of the metamodel is to be navigable and reflexive. The former means it is possible to reach any point in a process by traversing its structure. This increases the extensibility of the model, because a program can reach any element that needs to be modified. Since the metamodel is reflexive, it is possible for an element in the process to alter its structure at runtime. These features are possible because the open objects' model allows the composition of state machines at runtime.

In order to be applicable to different contexts, a model like Cumbia-XPM needs to be very expressive regarding the definition of tasks and in structures that can be constructed. We studied the expressiveness of Cumbia-XPM using two different strategies. In the first place, compatibility with some existing languages was verified (XPDL, BPEL, BPMN, IMS-LD) by making an analysis of the control structures from each one and defining translation schemas to convert structures from the original models to corresponding structures expressed with Cumbia-XPM. In some cases, the translation schema was implemented in an application called "importer" to allow the execution in our engine of processes defined with other languages. The emphasis in this comparison of the languages was put on structural elements related to control flow. Thus, specific tasks and details related to other domains were ignored. We are currently working on a comparison with YAWL, as is further explained in sections 5 and 6.

The other strategy to assess the expressiveness of the model involved control flow patterns. The twenty basic control flow patterns [16] stand for control structures, which appear frequently in workflow related applications. These patterns have become a standard for language comparison and the majority of commercial languages have available documents explaining how each pattern is supported. For someone who evaluates a language, these documents are useful, because they give a standardized way of knowing if the language can be used for a specific context. In the case of Cumbia-XPM, table 1 shows how patterns can be constructed using extension mechanisms. Values used in the column labeled "Support" have the following meaning: *D*, pattern is directly supported by the model; *Sp*, pattern is supported by specializing some elements; *Ada*, pattern is supported by adaptation of state machines.

4 Implementation and validation

The Cumbia group has developed several tools to validate its proposals. The most important one is an execution platform for Cumbia-XPM called *JCumbia*. JCumbia is formed by a set of components developed in Java, and was designed to be run in JBoss (see figure 8). In this platform the main element is an *engine* designed

Table 1
Support of control-flow patterns in Cumbia-XPM.

Pattern	Support	Pattern	Support	Pattern	Support
1 (seq)	D	2 (par-sp)	D	3 (synch)	D
4 (ex-ch)	Sp	5 (simple-m)	D	6 (m-choice)	Sp
7 (sync-m)	Sp	8 (multi-m)	D	9 (disc)	Ada-Sp
10 (arb-c)	D	11 (impl-t)	Sp	12 (mi-no-s)	Sp
13 (mi-dt)	D	14 (mi-rt)	D	15 (mi-no)	D
16 (def-c)	Sp	17 (int-par)	Sp	18 (milest)	Sp
19 (can-a)	D	20 (can-c)	D		

to execute Cumbia-XPM processes by implementing the open objects model. In order to run processes in this engine, it is first necessary to load a definition using an XML. From this representation, instances can be created and put into execution using a set with the initial data. Running instances of the same process are grouped together to simplify managment and to allow the sharing of data between several instances of the same process. The responsibilities of this engine are limited to executing processes by means of allowing the interaction of open objects, but it relies on additional components of the platform to offer other requirements.

Besides the engine, JCumbia also has a deposit that is used to store both definitions of processes and the state of suspended processes. JCumbia has a *Web interface* that is used to deploy processes, and to create and control instances. This interface provides textual and graphical views of the running processes, which are kept updated while instances are executed. Another way to monitor the execution of the different components of JCumbia is to use *consoles*, that are specialized desktop applications that have a direct connection to the server. The design of JCumbia offers the capacity to connect new components to enrich the execution of the process. These include engines for other domains that should run synchronized to the control component: section 6 presents briefly some projects related to this idea.

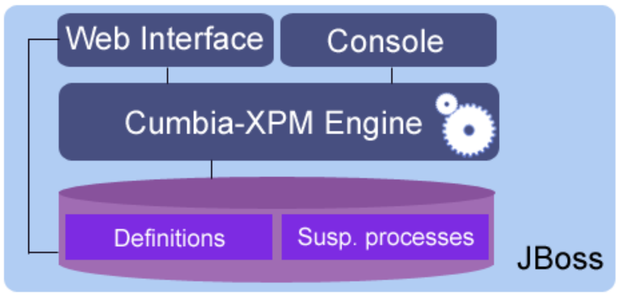

Fig. 8. Simplified architecture of JCumbia.

Besides doing manual tests of JCumbia, we have also addressed the problem of building automatic tests for it. However, to build a test suite of functional tests for a workflow engine is not an easy task, because techniques based on unitary testing present several problems. Since Cumbia-XPM is highly concurrent, a lot of synchronization issues appear when things are verified during execution; furthermore, since elements in the Cumbia-XPM metamodel are strongly interrelated, they are not susceptible to be tested independently. To solve this problems we developed a testing framework that allows the definition, execution and verification of testing scenarios. Scenarios are described by a static structure (a Cumbia-XPM process), instructions to control its execution and some assertions that have to be verified after its execution. The information to validate the assertions is gathered from execution traces in a fashion similar to what is done in [7,8]. This strategy based on static elements and off-line verification eliminates problems related to synchronization and concurrency. This framework was used to verify the engine of Cumbia-XPM, using scenarios selected to cover the most important and prone to error interaction cases that can be constructed with the metamodel.

Using the same framework, scalability tests of JCumbia were also made. These tests used also a complementing profiling tool[10] that led us to points in the implementation that were critical in the use of memory and processing time.

In this moment we have a complete workflow engine that has been thoroughly tested. This engine was built on a platform that can escalate to be used in real scenarios. As is reported on section 6, this engine is currently been used to execute high level languages such as BPEL and BPMN.

5 Related work

This section presents some works related to our research. In [10], Papadopoulos and Arbab present a survey of coordination models and classify them in two groups: data-driven and control-driven. In the former group, the main focus is put on Linda and its descendants but, as is presented in [11], Linda-like models have some deficiencies that make them inadequate to model workflows. However, some of their underlying ideas are useful in modern coordination models, such as the independence between the computation and coordination or the usage of a shared space for data (as is required in one of the implementation strategies described in [13]). One of the control-driven models presented in [10] is Manifold[11]. This model has several similarities to Cumbia-XPM and the open objects, but its design totally separates data from control. Whereas in Cumbia-XPM we can use the flow of data to control the state of a process, Manifold is completely event-driven. Also, there is no direct interaction between elements, other than the flow of data through streams.

A more recent and powerful coordination model is Reo[2]. It is an exogenous language based on channels which can be used to create very complex connectors. The usage of Reo as glue-code between heterogeneous components, promotes loose coupling because it makes possible to separate them and externalize their interaction and the flow of data between them. For instance, Reo has been applied to the problem of composing web services, as is presented in [6] and [9]. In these works, several features of Reo lead to advantages over similar solutions like BPEL: using Reo it is possible to dynamically reconfigure the channels to restructure the

[10] JMP Java Memory Profiler, URL: http://www.khelekore.org/jmp/

processes; distribution and mobility is also supported by the coordination model; finally, the formal description of Reo makes it possible to apply model checking techniques to the web services compositions.

In the work we have presented, state machines are used both to represent the state of the elements and to coordinate their execution. In other works we consulted, state machines have been used to analyze components and verify their compatibility. These techniques have been applied in cases where a static analysis of the interfaces is not enough, and further information is required about the behavior of the components[4].

One of the fundamental points of our proposal is the separation of processes in different perspectives. In Cumbia-XPM we have focused on the control or structural perspective, but within the Cumbia project other perspectives have also been studied and have resulted in complementing domains like the ones shown in section 6. In [13], the main topic discussed is the data perspective, but several works related to other perspectives are also referenced (time, resources, transactions, and functionality). However, the essential difference between their strategy and ours is that they propose the need to have a clear separation of perspectives when modeling a process, even if its implementation and execution integrates the perspectives into a single solution. With our strategy, the perspectives are separated when modeling, and stay separated, but coordinated, during execution.

YAWL is a workflow language specifically designed to support the original control-flow patterns[15]. Its design started with an analysis of the suitability of Petri nets to accomodate the patterns. After finding some limitations, they proposed a formal coordination model based on a Labeled Transition System (LTS). In order to compare YAWL to the approach proposed in this paper, we identify two complementing conceptual elements: the control model, that defines the elements that are used to build processes, and the coordination model that supports the execution of those elements. Like Cumbia-XPM, YAWL is also proposed as an intermediate language that should be used to make some high level languages executable.

One of the purposes of YAWL was to provide a model with a clear execution semantics. Because of this, the LTS they use has a formal description and a set of rules with conditions for state transition. Since the project started with Petri Nets, a similar textual and graphical syntax is used to describe the LTS, which also keeps concepts such as places, conditions and tokens. In the specific case of describing the execution semantics of a task, the life cycle is modeled with a Petri net that resembles the state machines used in open objects; however, these nets are identical for every element and are used only for documentation purposes and not for coordination or composition. Furthermore, in YAWL the execution of a running process depends on the state of the complete instance, and not just on the state of the single elements. One of the clear advantages of the formal semantics of YAWL is the possibility of analyze and verify processes. This has been explored using a tool called Woflan[11]. Currently, neither Cumbia-XPM nor open objects have a formal definition and semantics; however, that is part of our ongoing research.

[11] Woflan, URL: http://is.tm.tue.nl/research/woflan/

The other level of YAWL, which holds the elements used to build processes, has some deep differences with Cumbia-XPM. While our meta model was designed to have a few base elements and being able to accommodate extensions, YAWL has more base elements and proposes very limited extension mechanisms. Because of their inspiration in control-flow patterns, they include extra elements that are useful to solve patterns with simplicity, even though the same results can be achieved with other elements. Nevertheless, some YAWL elements can express behaviors that in Cumbia-XPM require extensions based on extra code, like the regions for token removal; however, used as intermediate languages, the complexity of the resulting executable models should not be taken into account. The extension mechanisms in YAWL are very limited: its main flexibility points are Tasks, which are used to execute specific code and consume services through worklets[1].

Finally, a big difference between YAWL and Cumbia is related to the support for other perspectives. In the case of YAWL, the support to the data, resource and operational perspectives is provided by the implementation and it is not part of the model [14,12]. In *newYAWL*[12] there is comprehensive support to data and resource patterns, but there is no uniformity between control and these perspectives. On the contrary, in Cumbia all the perspectives share the open objects and are executed and coordinated using homogeneous mechanisms.

6 State of the project

Since the Cumbia project [12] started three years ago, team efforts have been divided between the development of the coordination model and Cumbia-XPM, and some complementing subprojects that pursue specific goals: 1) to specify new domains that can be composed with the control domain; 2) to design and build complete working applications which use Cumbia-XPM as an intermediate language; 3) to formalize the coordination model. These projects have also been used to validate our proposals from different points of view.

Using open objects, we have built metamodels to handle other perspectives that are tightly related to control. *Cumbia-XRM* (eXtensible Resources Model) was designed to model the resources used in the execution of a process (working personnel or machines, for example). This metamodel is also capable of modeling complex rules to assign these resources to tasks. *Cumbia-XTM* (eXtensible Time Model) can be used to model time restrictions over the execution of a process. For instance, one restriction can specify that if a set of activities lasts more than three days an email has to be sent to the supervisor. XTM and XRM models are composed with Cumbia-XPM processes by synchronization of state machines: the same coordination methods used between open objects inside XPM, are used to synchronize elements from several metamodels. The weaving between elements of different models is specified externally, thus it is easy to add new models and to modify their relationships.

The construction of full working applications validates our proposal of using Cumbia-XPM as an intermediate language. One of the applications that we have

[12] Cumbia Home, URL: http://cumbia.uniandes.edu.co

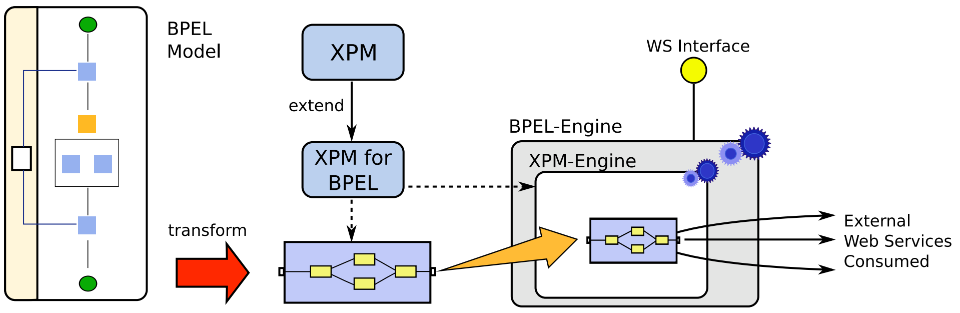

Fig. 9. Building a BPEL engine with Cumbia-XPM.

constructed is an engine for BPEL 2.0, which is currently in a beta stage. Figure 9 shows an overview of this. In the first place, there is an extension of XPM that adds the necessary elements to execute BPEL processes, like several types of activities to interact with web services. A BPEL model, built with one of the BPEL editors available, is automatically transformed into an equivalent XPM process that uses the extended elements. This process is then executed in a normal XPM Engine, as the extension itself provides all the extra logic required by BPEL. Finally, a wrapper is used around the XPM Engine to make it appear like a BPEL engine. In particular, this layer provides a management interface that is published as a web service. Another application that is being constructed is called *PaperXpress*, a tool to support collaborative writing of research papers. It allows the definition of ad-hoc processes, and offers the support needed to coordinate tasks and to handle the results of the writing process. The cases of BPEL and PaperXpress have a very important difference. Whereas BPEL is a language already defined, the high level language used in PaperXpress was specifically designed for it. However, the costs associated to the implementation of this language where low because of the usage of Cumbia-XPM as its basis.

Finally, the goal of other of our ongoing projects is to provide a formalization to the coordination model. This will provide tools to verify processes and further validate the approach by verifying its properties and making comparisons possible. As we already said, the formal aspect of YAWL permits static analysis of processes, and we will like to replicate that. On the other side, in order to do in depth comparisons with other coordination models, or other process models, we believe that a more formal description of the approach is required. We are currently working on this direction and this objective is guiding our formalization efforts.

7 Conclusions

In this paper we presented a strategy to build, at low cost, families of control-based applications. Around a central control component, these applications coordinate the execution of elements from other perspectives like resources, data or time. The flexibility required to build with ease families of these applications, is obtained with a coordination model based on open objects, which is used to materialize an independent metamodel for each concern. This coordination model offers extensibility mechanisms and provides a flexible way to describe composition and coordination between the models that represent perspectives appearing in an application.

Our proposal is based on a metamodel for control, Cumbia-XPM, and on a coordination model based on open objects, which is used to represent the metamodel. Cumbia-XPM is used to build processes (the control component of an application) and can be applied to several contexts: it was designed to support the elements that are part of a general concern of control, and it can be extended and adapted when it has to solve specific problems.

Currently, the models proposed are being formalized to do more formal comparisons with existing approaches. Meanwhile, we have evaluated the expressiveness of our models using the workflow patterns. The concepts presented in this paper have also been validated with the construction of engines and applications. The basic Cumbia-XPM engine was implemented and tested intensively using a framework based on scenarios. Similar engines were constructed to execute models that represent different perspectives of the applications.

Applications that run BPEL and BPMN processes are almost complete: they make use of an extended version of Cumbia-XPM as an executable intermediate layer; they also depend on the synchronized execution of other models, built with open objects, that represent the other perspectives. We expect to continue this line of research in the future, applying it to other control-based applications, specially those that use their own domain specific languages (like PaperXpress). This line of research also requires the development of new metamodels for different perspectives, and the evolution of our testing framework to support also these new concerns.

References

[1] Adams, M., A. H. M. ter Hofstede, D. Edmond and W. M. P. van der Aalst, *Worklets: A service-oriented implementation of dynamic flexibility in workflows*, in: *CoopIS 06: Proceedings of the 14th International Conference on Cooperative Information Systems, Montpellier, France*, Lecture Notes in Computer Science **4275** (2006), pp. 291–308.

[2] Arbab, F. and F. Mavaddat, *Coordination through channel composition*, in: *COORDINATION '02: Proceedings of the 5th International Conference on Coordination Models and Languages* (2002), pp. 22–39.

[3] Aßmann, U., "Invasive Software Composition," Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[4] Brogi, A., C. Canal and E. Pimentel, *Behavioural types for service integration: Achievements and challenges*, Electronic Notes in Theoretical Computer Science **180** (2007), pp. 41–54.

[5] Craven, N. and D. Mahling, *Goals and processes: a task basis for projects and workflows*, in: *COCS '95: Proceedings of conference on Organizational computing systems* (1995), pp. 237–248.

[6] Diakov, N. K. and F. Arbab, *Compositional construction of web services using reo*, in: *Web Services: Modeling, Architecture and Infrastructure - Proceedings of the 2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure, WSMAI 2004* (2004), pp. 49–58.

[7] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Patterns in property specifications for finite-state verification*, in: *ICSE '99: Proceedings of the 21st international conference on Software engineering* (1999), pp. 411–420.

[8] Hu, Z. and S. M. Shatz, *Explicit modeling of semantics associated with composite states in uml statecharts*, Automated Software Engg. **13** (2006), pp. 423–467.

[9] Limniotes, T. A., G. A. Papadopoulos and F. Arbab, *Web services: separation of concerns: computation coordination communication*, in: *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing* (2004), pp. 492–497.

[10] Papadopoulos, G. A. and F. Arbab, *Coordination models and languages*, in: *761*, Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 1998 p. 55.

[11] Papadopoulos, G. A. and F. Arbab, *Modelling activities in information systems using the coordination language manifold*, in: *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing* (1998), pp. 185–193.

[12] Russel, N., A. ter Hofstede, W. van der Aalst and D. Edmond, *newYAWL: Achieving comprenhensive patterns support in workflow for the control-flow, data and resource perspectives*, Technical report, BPMcenter.org (2007), BPM Center Report BPM-07-05.

[13] Sadiq, S., M. Orlowska, W. Sadiq and C. Foulger, *Data flow and validation in workflow modelling*, in: *ADC '04: Proceedings of the 15th Australasian database conference* (2004), pp. 207–214.

[14] van der Aalst, W., L. Aldred, M. Dumas and A. ter Hofstede, *Design and implementation of the yawl system*, in: *Advanced Information Systems Engineering. 16th International Conference, CAiSE 2004, Riga, Latvia, June 2004*, Lecture Notes in Computer Science **3084** (2004), pp. 291–308.

[15] van der Aalst, W. and A. ter Hofstede, *Yawl: Yet another workflow language (revised version)*, Technical report, Queensland University of Technology, Brisbane (2003), QUT Technical report, FIT-TR-2003-04.

[16] van der Aalst, W., A. ter Hofstede, B. Kiepuszewski and A. Barros, *Workflow patterns*, Technical report, BPMcenter.org (2003), BPM Center Report BPM-03-06.