Predicting Coupling of Object-Centric Business Process Implementations

Ksenia Wahler and Jochen M. Küster

IBM Zurich Research Laboratory, Säumerstr. 4
8803 Rüschlikon, Switzerland
{ryn,jku}@zurich.ibm.com

Abstract. Object-centric approaches for business process implementation distribute process logic among several interacting components, each representing a life cycle of an object. One of the challenges is to manage the component coupling, because highly-coupled components are difficult to distribute, maintain and adapt. Existing techniques that derive a component for each object that changes state in a given process do not consider component interdependencies and run the risk of producing components that are highly coupled. To make coupling explicit and manageable during component identification, we propose an approach for computing the expected coupling of an object-centric implementation for a given process model prior to actually deriving this implementation.

Keywords: Coupling, object life cycle, object-centric and data-driven processes, state machines.

1 Introduction

Most existing languages for business process modeling (e.g. BPMN [3]) and implementation (e.g. BPEL [1]) are *activity-centric*, because they represent processes as a set of activities connected by control-flow elements to indicate the order of activity execution. In recent years however, a line of alternative *object-centric* approaches for modeling and implementing business processes has been proposed, which include artifact-centric modeling [6,15], adaptive business objects [14], data-driven modeling [11] and proclets [20]. Activities in the process are distributed among several *components*, each representing an *object life cycle* that defines possible states of a particular object and transitions between these states. Interaction between such *object life cycle components* ensures that the overall process logic is correctly implemented. Object-centric implementations can be used for *distributed* process execution and can lead to a more *maintainable* and *adaptable* implementation than activity-centric approaches, as the behavior of one object can be partially changed without influencing the rest of the process [10]. However, the more dependencies and interactions there are between the object life cycle components, the costlier becomes their distribution and the more complicated it is to change their behavior.

One of the challenges in object-centric process implementation is therefore the management of component interdependencies, commonly referred to as *coupling*

M. Dumas, M. Reichert, and M.-C. Shan (Eds.): BPM 2008, LNCS 5240, pp. 148–163, 2008.

in software engineering [7]. Several object-centric approaches advocate deriving object life cycles from activity-centric process models that specify the process logic to be implemented. The existing derivation methods [6,18,19] do not explicitly address object life cycle interdependencies and hence run the risk of producing components that are highly coupled. Component refactoring, e.g. moving some behavior from one component to another or merging components, is one approach to reduce coupling. However, as a result the process model can get out of sync with its implementation, which challenges the propagation of any subsequent process model changes to the implementation. This problem can be alleviated by making the developer aware of the expected coupling before component derivation, so that the process model can be adapted until a desired level of coupling is achieved. Realization of this approach requires the computation of the expected component coupling based on a given process model.

The problem addressed in this paper is therefore the prediction of the expected coupling of an object-centric implementation based on a given process model. We first review the mapping of the most common *workflow patterns* [21] to object life cycle components in order to identify how properties of a process model influence the coupling of the derived components. We then show that given a process model, it is possible to compute the object life cycle component pairs that require interaction by analyzing the control flow between activities that change the state of objects. Finally, we use this information to compute the expected coupling of the object life cycle components.

We implement object life cycle components using Business State Machines (BSMs) [5]. BSMs are introduced in Sect. 2, along with an illustrative example and the coupling metric used. In Sect. 3, we demonstrate how workflow patterns can be implemented using BSMs and study how solutions for different patterns contribute to the overall coupling. These observations are formalized in Sect. 4, where we define how to compute the expected coupling based on a given process model. In Sect. 5, we discuss the generalization of our approach. Related work and conclusions are presented in Sect. 6 and 7, respectively.

2 Example and Background

As an illustrative example, we use a process designed for the organization of alumni events at the IBM Zurich Research Laboratory. An abridged BPMN [3] model for this process is shown in Fig. 1. After the approval of the budget, the date for the event is fixed and then two things happen in parallel: the program, invitations and web site are prepared; and catering is organized. After all these have completed, the alumni day is hosted. The process model contains three sub-processes: Fix Date, Prepare And Send Invitations and Develop Web Site.

All activities of a business process generally transform some objects by changing their state to contribute to the final goal of the process. For each atomic activity in the alumni day process, we indicate the state-changing objects[1]. For

[1] We use the notation given on p.94 in [3] for object outputs of an activity and a shorthand notation for objects that are both inputs and outputs of an activity.

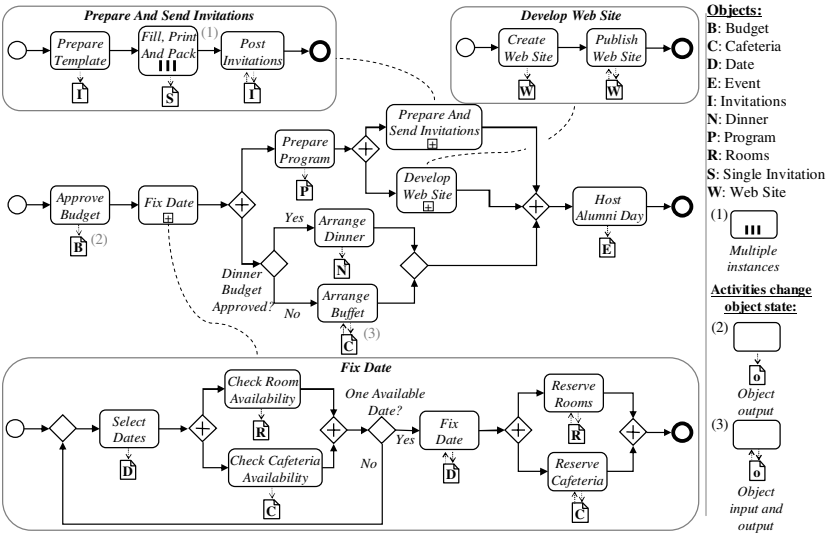

Fig. 1. Process Model for Alumni Day Organization

example, the Create Web Site activity produces a Web Site object in state Drafted and Publish Web Site changes the state of the Web Site object from Drafted to Published (states are omitted in this diagram). In the Prepare And Send Invitations subprocess, Prepare Template creates an Invitations object in state TemplatePrepared, and then multiple instances of the Fill, Print And Pack activity are performed in parallel, each creating a Single Invitation object. Once all instances of Fill, Print And Pack have completed, Post Invitations updates the state of Invitations to Posted.

In an object-centric implementation of the alumni day process, the process logic is split into ten object life cycle components, assuming an approach in which one component is derived for each state-changing object. We implement each object life cycle component as a Business State Machine (BSM) [5]. A simple example of a BSM is shown in Fig. 2.

A BSM is a finite state automaton, tailored for execution in a service-oriented environment. Each BSM can have several of the following: *interfaces*, *references* and *variables*. The Simple BSM in Fig. 2 has two interfaces: basic comprising *operations* start and stop, and stateQuery with the getState operation. These are the three operations that can be invoked on this BSM. Simple also has one reference r, referencing an interface of another BSM, with one operation getState. Operations in addition have parameters, which we omit here. Simple has one variable rState, initialized to the literal "Unknown".

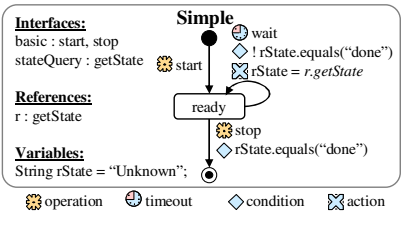

Fig. 2. Example BSM

State transitions in BSMs follow the *event-condition-action* paradigm. A transition can be triggered either by an expiration of a *timeout* or by an invocation of an *operation* defined in one of the BSM's interfaces. Once a transition has been triggered, its associated *condition*, if any, is evaluated. If the condition evaluates to true or there is no condition, the *action* associated with the transition, if any, is performed and the target state of the transition is entered. An action either invokes an operation on one of the BSM's references or performs some other processing specified in a custom language, such as Java. For example, once the Simple BSM is in state ready, a self-transition is triggered repeatedly after expiration of the timeout wait. Each time the transition is triggered and rState is not equal to "done", the getState operation is invoked on r (invocation is indicated using italics in the diagrams). The operation getState is implicitly handled by every BSM and returns the BSM's current state. Invocation of the stop operation on Simple results in a transition to the final state only if rState is equal to "done".

At runtime, each BSM instance is associated with a *correlation ID*. The runtime engine creates a new BSM instance if it receives a call to an operation associated with an initial transition of some BSM and this operation call specifies a correlation ID that does not correspond to an existing BSM instance.

For the implementation of the alumni day process, we distribute the process activities among ten BSMs (Budget, Cafeteria, Date, etc). We make a simplifying assumption that one activity changes the state of exactly one object, as in the example process model. Each activity is placed into the BSM that represents the state-changing object for this activity. In Sect. 6, we explain how our approach can be extended to handle activities that change the state of several objects.

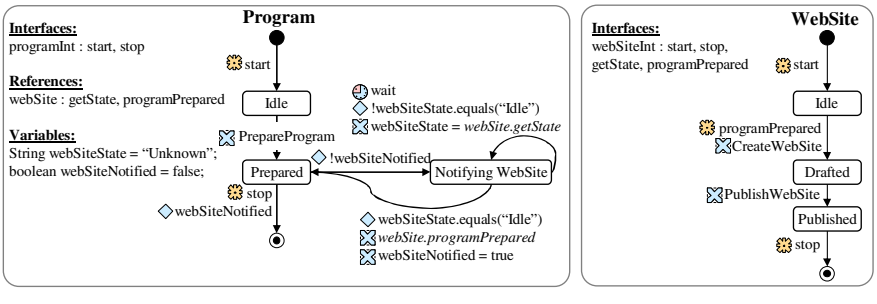

Fig. 3. Example BSMs

Partial implementations of the Program and WebSite BSMs are shown in Fig. 3. Activities that change the state of these two objects are mapped to actions associated with state transitions in the BSMs, e.g. the PrepareProgram and CreateWebSite actions. These actions can be implemented to invoke service operations, human tasks, etc. In the process model, the Prepare Program activity must complete before the Create Web Site activity can execute. *Synchronization* of the Program and WebSite BSMs is implemented to preserve this dependency: After the PrepareProgram action has been performed in the Program BSM and the Prepared state

has been reached, the Program BSM transits to state Notifying WebSite. In this state, the Program BSM queries the state of the WebSite BSM repeatedly. Once the WebSite BSM has reached the Idle state, the Program BSM notifies the WebSite BSM that it has reached the state Prepared by invoking the programPrepared operation. After this, the Program BSM transits back to state Prepared, and the WebSite BSM can perform the CreateWebSite action. In a complete BSM implementation of the alumni day process, many such synchronizations need to be implemented, e.g. Program also needs to synchronize with the Invitations BSM.

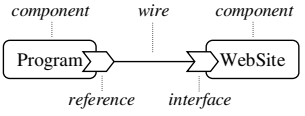

Fig. 4. Assembly Model

Aside from additional states and transitions within BSMs, synchronization also leads to interface bindings between the BSMs. We use the *Service Component Architecture (SCA)* [2], which is a service-oriented component framework, to represent these bindings. Each BSM is an implementation of an *SCA component* (used interchangeably with component from now on). An *assembly model* in SCA is a representation of directed communication channels, called *wires*, between components. The assembly model for the BSMs from Fig. 3 is shown in Fig. 4. Synchronization of the Web Site and Program BSMs requires that the components are connected by a wire in the assembly model.

Definition 1 (Assembly model). *An* assembly model *is a tuple* $M = (C, \phi)$*, where* C *is the set of components in* M *and* $\phi \subseteq C \times C$ *is the wire relation between components.*

In the context of SCA, we use the term coupling to refer to interdependencies of components in an assembly model. We quantify the coupling of an assembly model by defining the *interface coupling* metric, adapted from existing work on quality metrics in the business process domain [17].

Definition 2 (Interface coupling). *Given an assembly model* $M = (C, \phi)$*, its interface coupling is defined as follows:*

$$p(M) = \begin{cases} 0 & \text{if } |C| = 0 \text{ or } 1 \\ \frac{|\phi|}{|C| \times (|C|-1)} & \text{otherwise} \end{cases} \quad (1)$$

Interface coupling represents the ratio between the actual number of wires and the maximum possible number of wires between the components in the assembly model. A coupling value of 0 means that there is no interaction at all between the components. This implies that the distribution of these components does not incur any communication costs, and the implementation of each component can be maintained and its behavior adapted at run time with no side-effects on the other components. On the contrary, a coupling value of 1 means that every component interacts with every other component. The distribution of such components will incur high communication costs, and maintenance or adaptation of one component affects the behavior of all other components. The interface coupling of the assembly model shown in Fig. 4 is $\frac{1}{2 \times 1} = 0.5$. More refined coupling metrics could also be used here, e.g. to take into account the number of operations in the component interfaces connected to wires or the number of operation calls inside the BSMs.

In the following section, we explore how the implementation of different workflow patterns using BSMs introduces wires between the BSM components and thus contributes to the coupling of the resulting assembly model.

3 Implementing Workflow Patterns Using BSMs

Workflow patterns [21] are a well-established benchmark for exploring how common process behaviors can be represented in different business process modeling and implementation languages. In this section, we show how the basic control-flow patterns, WP1-WP5, can be modeled using BSMs. In addition, we provide a solution for WP14, as it can be used to represent the processing of object collections, commonly occurring in process models. We provide BSM solutions on an exemplary basis, similar to existing evaluations of other languages (e.g. [22]). We discuss the requirements that each pattern has with respect to the synchronization of BSMs and its contribution to the coupling of the overall implementation.

WP1 Sequence. Several activities are executed one after another in this pattern, as illustrated with two examples[2] in Fig. 5. In E1, ActivityA and ActivityB change the state of the same object o1, whereas in E2 ActivityA and ActivityB change the state of different objects, o1 and o2.

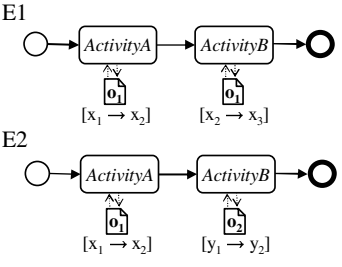

Fig. 5. WP1 Examples

The solution for E1 is straightforward, see Fig. 6(a) (interfaces and references are omitted). It comprises one component, shown in the assembly model at the bottom of the diagram. A solution for E2 is shown in Fig. 6(b), where BSMs o1 and o2 represent the life cycles of objects o_1 and o_2, respectively.

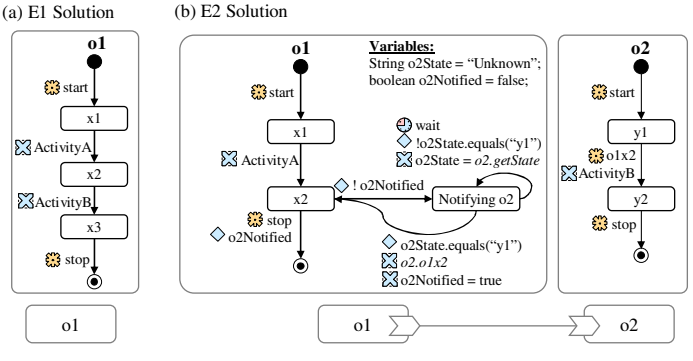

Fig. 6. WP1 Solutions

[2] We use a shorthand of the form $[state_{src} \rightarrow state_{tgt}]$, based on the notation given on p.94 in [3], to show how an activity changes the state of an object.

Once ActivityA has been performed by o1, o1 notifies o2 that it has reached state x2 by first ensuring that o2 is in state y1 and then invoking the o1x2 operation on o2. Once o1x2 has been invoked on o2, ActivityB is performed by o2. The resulting assembly model has an interface coupling of $\frac{1}{2\times1} = 0.5$.

WP1 Synchronization Requirements: A generic instance of WP1 comprises activities $a_1, ..., a_n$, which change the states of objects $o_1, ..., o_n$, respectively. A pair of activities a_i, a_{i+1}, with $1 \leq i < n$, requires a synchronization of BSM o_i and BSM o_{i+1} if and only if $o_i \neq o_{i+1}$. We introduce the *control handover* synchronization category for such synchronizations, since they represent the handover of control between BSMs. Each such control handover requires a wire from BSM o_i to BSM o_{i+1} to be present in the assembly model. The introduction of these wires contributes to the overall coupling of the resulting assembly model.

WP2 Parallel Split & WP3 Synchronization. In WP2, several activities are executed simultaneously or in any possible order, and in WP3, several parallel threads are joined together into a single control thread. An example containing an instance of both of these workflow patterns is shown in Fig. 7. In E3, each activity changes the state of a different object. Note that we do not only consider block-structured process models, but examine these two patterns together for the sake of conciseness.

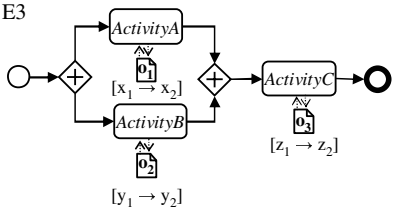

Fig. 7. WP2 & WP3 Example

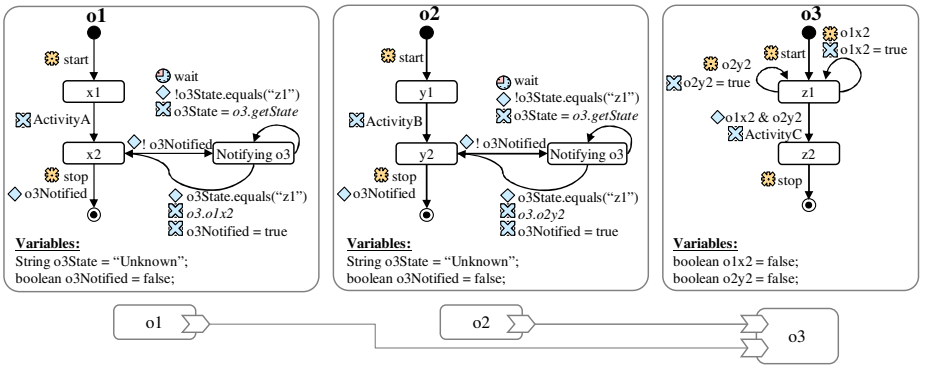

Fig. 8. WP2 & WP3 E3 Solution

A solution for E3 is shown in Fig. 8. As by default all BSMs are executed concurrently, no explicit parallel split is required. Synchronization of the threads is performed using notifications, similar as in the E2 solution in Fig. 6(b). BSM o3 waits to receive notifications from both o1 and o2 (operation calls o1x2 and o2y2) before performing ActivityC. The interface coupling of the assembly model for this solution is $\frac{2}{3\times2} \approx 0.33$.

WP2 & WP3 Synchronization Requirements: As instances of WP2 do not require any interaction between BSMs, they do not contribute wires to the assembly model and have no effect on the coupling. A generic instance of WP3 comprises activities $a_1, ..., a_n$ that all need to complete before activity a_{n+1} can begin execution. Assuming that $a_1, ..., a_n, a_{n+1}$ change the states of objects $o_1, ..., o_n, o_{n+1}$, respectively, a pair of activities a_i, a_{n+1}, with $1 \leq i \leq n$, requires a synchronization of BSMs if and only if $o_i \neq o_{n+1}$. These synchronizations also fall into the control handover category, introduced for WP1.

WP4 Exclusive Choice & WP5 Simple Merge. In WP4, one out of several activities is executed based on the outcome of a decision, and in WP5, several alternative threads are joined into one control thread without synchronization. An example containing instances of these patterns is shown in Fig. 9.

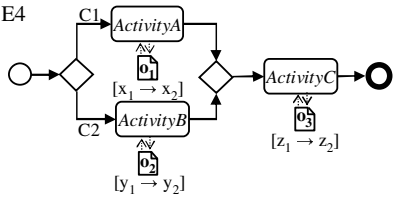

Fig. 9. WP4 & WP5 Example

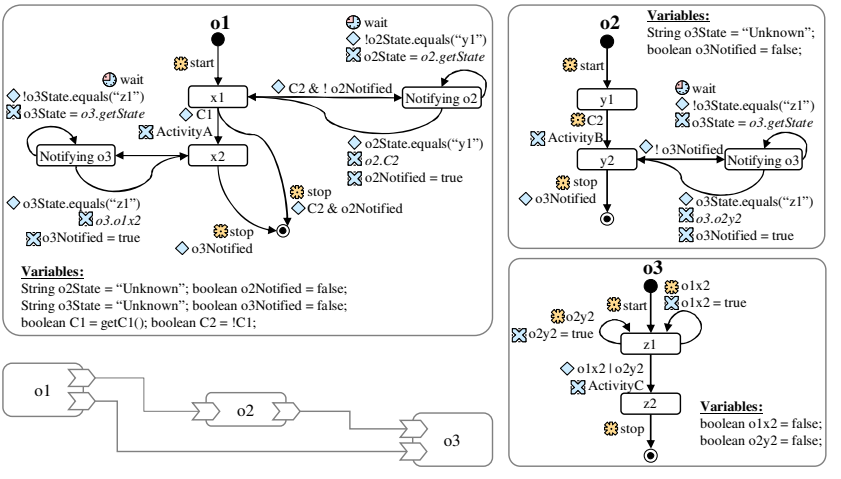

Fig. 10. WP4 & WP5 E4 Solution

In a solution for this pattern, the decision needs to be placed into one of the BSMs, as shown in Fig. 10 where it is placed in BSM o1 (two transitions going out of state x1 with conditions C1 and C2). Once the decision has been evaluated in o1[3], either ActivityA is performed (C1 is true) or o2 is notified and ActivityB is performed in o2 (C2 is true). The merging of alternative control threads is implemented similarly to the synchronization solution in Fig. 8, except that BSM o3 performs ActivityC as soon as it receives one of the operation calls,

[3] For simplicity, we initialize C1 and C2 in the variable definitions here. In a real implementation, they would be evaluated in state x1.

o1x2 or o2y2. The interface coupling of the assembly model is $\frac{3}{3\times 2} = 0.5$. These three components have a higher coupling value than those in Fig. 8, because of an additional wire between o1 and o2 required for communicating the decision outcome.

WP4 & WP5 Synchronization Requirements: A generic instance of WP4 comprises a decision d and activities $a_1, ..., a_n$, which change the states of objects $o_1, ..., o_n$, where one of these activities is executed depending on the evaluation of the conditions of d. We assume that the evaluation of d can be assigned to an object o_i, where $1 \leq i \leq n$. BSM o_i requires synchronization with each BSM o_j, where $1 \leq j \leq n$ and $o_i \neq o_j$. Since such synchronizations do not represent control handovers, we introduce a new synchronization category called *decision notification* for such synchronizations. Instances of WP5 require control handover synchronizations, similar to instances of WP1 and WP3.

WP14 Multiple Instances with a priori Run-Time Knowledge. In WP14, multiple instances of the same activity are created, all of which need to complete before a subsequent activity can be executed. The number of instances is not known at design time, but is determined at run time before activity instances are created. This pattern can be used to represent the processing of object collections, as shown in the example in Fig. 11. In E5, a collection of o2 objects is processed by multiple instances of ActivityB. In this example, each activity instance creates a new object o2 in state y1. Once all instances of ActivityB have completed, ActivityC is executed. We show this particular example here, because it corresponds to the Prepare And Send Invitations sub-process in Fig. 1, where Invitations and Single Invitation objects take the role of o1 and o2, respectively.

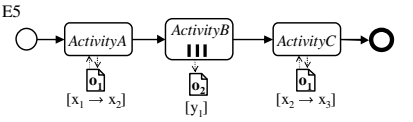

Fig. 11. WP14 Example

A solution for E5 is shown in Fig. 12. After performing ActivityA, o1 transits to state x2 and then to Creating o2s, where it creates n instances of the o2 BSM by repeatedly invoking the start operation with a fresh correlation ID. Each o2

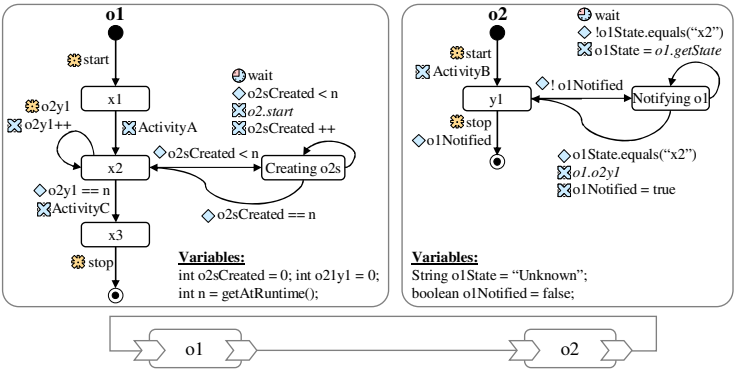

Fig. 12. WP14 E5 Solution

instance performs ActivityB and then notifies o1 that it has reached state y1. Once o1 has received notifications from all o2 instances, it performs ActivityC and transits to state x3. The interface coupling for the assembly model is $\frac{2}{2\times1} = 1$.

WP14 Synchronization Requirements: A generic instance of WP14 comprises activities a_1, a_2, a_3, which change the states of objects o_1, o_2, o_3, where activity a_2 is to be instantiated multiple times. Provided that a_1 and a_3 are not themselves multiple instance activities, the following control handovers are always required: from BSM o_1 to instances of BSM o_2, and from instances of BSM o_2 to BSM o_3. Although the number of synchronizations at run time will vary, the contribution to the coupling is constant, as two wires, (o_1, o_2) and (o_2, o_3), are introduced into the assembly model to enable the synchronizations (this also holds if $o_1 = o_3$). The case where $o_1 = o_2$ and $o_2 = o_3$ is an exception, as in this case only one wire (o_2, o_2) would be introduced into the assembly model. For simplification, we do not consider this case in the remainder of the paper.

In this section, we have demonstrated how workflow patterns can be implemented using BSMs and discussed the requirements of each pattern for the synchronization of BSMs. In the next section, we show how the number of control handovers and decision notifications can be computed for a given process model, and then used to compute the expected coupling of a BSM implementation.

4 Predicting Coupling of BSM Implementations

We assume that the process model provided as a specification for a BSM implementation comprises instances of WP1-WP5 and WP14 only and has each activity associated with one state-changing object, as in the alumni day process model in Fig. 1. As a sub-process hierarchy in a given process model can be flattened for processing, we use the following definition for a process model.

Definition 3 (Process model). *A* process model *is a tuple* $P = (G, O, \sigma)$*:*

- $G = (N, E)$ *is a directed graph, in which each node* $n \in N$ *is either a start node, stop node, activity, fork, join, decision, or merge. As a shorthand, we use* N_A *and* N_D *to denote activities and decisions in* N*, respectively.*
- O *is the set of objects whose states are changed by activities* $a \in N_A$*.*
- $\sigma \subseteq N_A \times O$ *is the state-changing relation between activities and objects. We use* o_a *to denote the object whose state is changed by activity* $a \in N_A$*, i.e.* $(a, o_a) \in \sigma$*.*

Given a process model P, the number of components in the assembly model of its BSM implementation is equal to the number of objects whose states are changed in P, assuming a simple mapping. In Sect. 3, we showed that the number of wires between the components depends on the control handover and decision notification synchronizations between the BSMs. As all the synchronizations required by different patterns fall into these two categories, we directly compute all object pairs that require such synchronizations, instead of first identifying workflow pattern instances in a given process model.

A control handover is required whenever an activity in the process model that changes the state of one object has a direct successor activity [4] that changes

[4] Only edges and gateways connect an activity and its direct successor activity.

the state of another object. A decision notification is required between the object assigned to evaluate a decision and all the objects whose state is changed by the direct successor activities of that decision. To compute the objects that require control handovers and decision notifications, we propagate the information about state-changing objects *downstream* from each activity to its direct successor activities and *upstream* from direct successor activities of decisions.

Definition 4 (Downstream and upstream control objects). *Given a process model $P = (G, O, \sigma)$ where $G = (N, E)$, each edge $e \in E$ is associated with* downstream *and* upstream control objects*, $dco(e), uco(e) \subseteq O$ respectively, defined as follows:*

$$dco(e) = \begin{cases} \emptyset & \text{if } e \text{ is the outgoing edge of the start node} \\ \{o_a\} & \text{if } e \text{ is the outgoing edge of activity } a \in N_A \\ \bigcup_{i=1}^{m} dco(e_i) & \text{otherwise, where } e_1, ..., e_m \text{ are the incoming edges} \\ & \text{of node } n, \text{ which has } e \text{ as its outgoing edge} \end{cases} \quad (2)$$

$$uco(e) = \begin{cases} \emptyset & \text{if } e \text{ is the incoming edge of the stop node} \\ \{o_a\} & \text{if } e \text{ is the incoming edge of activity } a \in N_A \\ \bigcup_{i=1}^{m} uco(e_i) & \text{otherwise, where } e_1, ..., e_m \text{ are the outgoing edges} \\ & \text{of node } n, \text{ which has } e \text{ as its incoming edge} \end{cases} \quad (3)$$

Downstream and upstream control objects can be computed for a given process model using data flow analysis techniques [9]. For example, to compute the downstream control objects, $dco(e)$ is initialized to an empty set for each edge e and then the nodes in the process model are traversed, evaluating the *dco* equations (Equation 2) for each outgoing edge of the traversed node. Reverse postorder traversal ensures that in the absence of cycles each node is visited once. In the presence of cycles, the nodes are traversed repeatedly until a fixpoint is reached, i.e. an iteration when no *dco* values are updated. Fig. 13 shows the alumni day process model with the downstream and upstream control objects indicated above and below each edge, respectively[5]. The set of object pairs that need to perform control handover is then defined as follows.

Definition 5 (Control handover object pairs). *Given a process model $P = (G, O, \sigma)$ where $G = (N, E)$ and each of the edges $e_1, ..., e_n$ is an incoming edge of some activity $a \in N_A$, the set of directed* object pairs *that require BSMs to perform* control handover *is defined as follows:*

$$O^{ch}(P) = \bigcup_{i=1}^{n} (dco(e_i) \times uco(e_i)) \setminus \{(o, o) \mid o \in O\} \quad (4)$$

For example, the incoming edge of the AD activity gives rise to two control handover object pairs: (R,N) and (C,N); and the incoming edge of the AB2 activity gives rise to only one control handover object pair: (R,C).

Next we define object pairs that require decision notification between BSMs. Given a decision d, we denote its outgoing edges by E_d^{out} and assume that the evaluation of d can be assigned to the object $co(d)$, which is one of the upstream control objects of some edge in E_d^{out}.

[5] Activity names are abbreviated in Fig. 13.

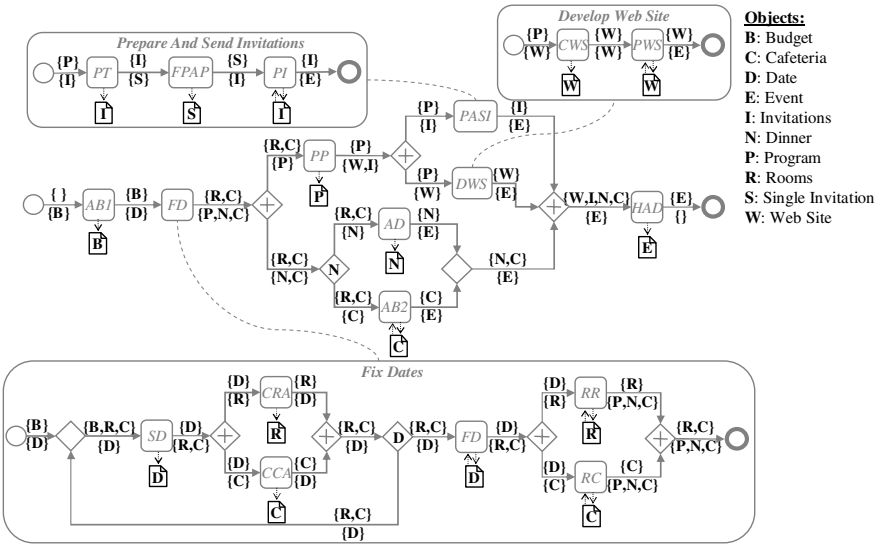

Fig. 13. Downstream and Upstream Control Objects in a Process Model

Definition 6 (Decision notification object pairs). *Given a process model* $P = (G, O, \sigma)$ *where* $G = (N, E)$*, the set of directed* object pairs *that require* decision notification *between BSMs is defined as follows:*

$$O^{dn}(P) = \bigcup_{d \in N_D} \left(co(d) \times \bigcup_{e \in E_d^{out}} uco(e) \right) \setminus \{(o, o) \mid o \in O\} \tag{5}$$

The decision in the parent alumni day process model is assigned to object N and gives rise to one decision notification object pair: (N,C). The decision in the Fix Date sub-process is assigned to object D. It does not introduce any decision notification object pairs, as the sets of upstream control objects for both edges going out of the decision are the same: {D}.

The predicted assembly model for a BSM implementation of a given process model can now be constructed by introducing a component for each object and a wire for each of the control handover and decision notification object pairs.

Definition 7 (Predicted assembly model for a BSM implementation). *Given a process model* $P = (G, O, \sigma)$*, the* predicted assembly model for a BSM implementation *of* P *is defined as follows:*

$$M_P = (C_P, \phi_P) \tag{6}$$

- *where* $C_P = \{c_{o_1}, ..., c_{o_n}\}$ *is the set of components, with one component* c_{o_i} *for each object* $o_i \in O$ *where* $1 \leq i \leq n$*,*
- *and* $\phi_P = \{(c_{o_1}, c_{o_2}) \in C_P \times C_P \mid (o_1, o_2) \in O^{ch}(P) \cup O^{dn}(P)\}$ *is the wire relation between components.*

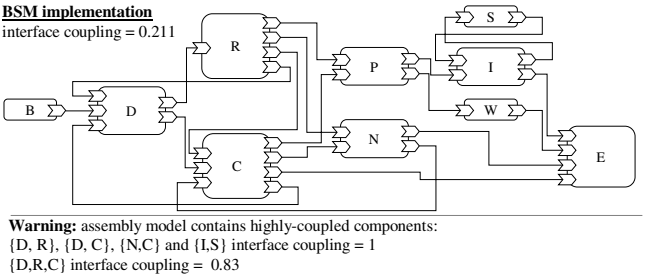

Fig. 14. Predicted Assembly Model for the Alumni Day Process

The assembly model for the alumni day process model is shown in Fig. 14. It can be seen that each distinct control handover and decision notification object pair, such as (R,N) or (C,N), introduces a wire in the predicted assembly model.

The interface coupling is computed for the entire assembly model and for all component subsets according to Definition 2. A configurable upper bound is used to assess the predicted coupling values. This upper bound can be evolved as a best practice by developers, i.e. first initialized to some value and then refined in further iterations or projects based on the experience gained in deploying and maintaining object-centric implementations. Empirical evaluations can also help in determining a generic guideline for this upper bound. In Fig. 14, the overall interface coupling is $\frac{19}{10\times 9} \approx 0.211$, which would not give a reason for concern, assuming for example an upper bound of 0.8. However, component sets {D,R}, {D,C}, {N,C}, {I,S} and {D,R,C} have a coupling value higher than 0.8 and would thus be brought to the attention of the developer, as shown in Fig. 14.

Once the expected coupling is predicted using the proposed approach, the developer should decide how to deal with each set of highly-coupled components. High coupling may be tolerated for components that have a stable design and do not require distributed deployment. Otherwise, the process model should be revised in such a way that the expected coupling between components is reduced. Possible revisions include identification of objects that can be represented by a merged life cycle and refactoring control flow in the process model. Object life cycle merger should be applicable only for those objects that have a strong semantic relationship. For example, the Dinner (N) and Cafeteria (C) life cycles, which give rise to the highly-coupled component set {N,C}, can be merged to produce a Catering life cycle. In order to alleviate component coupling by process model refactoring, the number of control handovers and decision notifications should be reduced. In the alumni day process, the decision Dinner Budget Approved? and the activities connected to its outgoing edges could take place directly after the Reserve Cafeteria activity, without waiting for the Reserve Event Rooms activity to complete. This refactoring would reduce the coupling of the {R,C} and {R,N} component sets. After each life cycle merger and process model refactoring, the coupling computations need to be repeated and shown to the developer.

5 Discussion

In this paper, we have shown how the coupling of an object-centric implementation using BSMs can be predicted using a given process model. We assumed that each activity in the process model changes the state of one object. An activity that changes the state of several objects would be placed into several BSMs, which would need to synchronize, thus contributing to component coupling. Our current approach can be extended to handle such activities by adding a new synchronization category, *activity synchronization*, and providing a definition for computing the object pairs requiring such a synchronization (similar to Definitions 5 and 6). The approach can be further extended to handle workflow patterns other than WP1-WP5 and WP14 by investigating BSM solutions for these patterns, identifying pattern requirements for synchronization of BSMs, and extending Definitions 5, 6 and 7.

Although our approach was demonstrated using SCA and BSMs, it can be generalized to other component frameworks (not necessarily based on services) and other object-centric approaches. For example, adaptive business objects (ABO) [14] are based on communicating automata, and our approach is applicable once every ABO has been encapsulated in a component and communication channels between the components have been made explicit. In data-driven modeling [11], object life cycles are synchronized by so-called external state transitions. To compute the coupling, each life cycle can be seen as a component, and communication channels need to be introduced between components whose life cycles are connected by external state transitions. Proclets [20] use WF-nets to represent object life cycles and make use of explicit communication channels. Although more advanced communication options, such as multicast and broadcast, are supported in proclets, our approach is still applicable.

6 Related Work

In component-based development, the coupling has been used for component identification [8] and refactoring [4]. For example, a statistics technique called clustering analysis to form components that have high cohesion and low coupling is used in [8]. Such approaches are complementary to what we propose in this paper, as they can help to identify how the highly-coupled components in the predicted assembly model of a BSM implementation can be alleviated.

Many different categories or types of coupling have been identified in software engineering [7]. Given source code or a component model, it is usually straightforward to calculate the different coupling values, as the metrics are defined directly in terms of source code or component model elements. In our approach, we determine how the control flow in a given process model influences the coupling of the resulting BSM implementation before actually deriving the BSMs. So far we have focused on the so-called interface coupling of SCA components; however other types of coupling, such as data coupling, could also be considered.

A tight correlation between the semantic relationships of objects and synchronization of their life cycles has been identified in manufacturing processes [11,16].

In manufacturing, objects are naturally coupled by the "part-of" relationship. Our approach is valuable also in this context, as it can identify whether the implementation components have dependencies other than those resulting from the semantic relationships between objects.

In workflow management, several approaches have been proposed for decentralizing workflows with the goal of optimizing their execution [12,13]. For example, the approach in [12] involves minimizing the loads and number of synchronization messages exchanged between the distributed workflow components. Although in our approach we also strive to reduce the number of dependencies between components, execution optimization is not our primary focus. The object life cycle components dealt with in this paper need to be refined and maintained by developers, whereas workflow decentralization happens once a workflow is deployed and its results are not exposed to the developers.

7 Conclusions and Future Work

We have presented an approach for predicting the coupling of an object-centric implementation for a given process model. Our example showed that deriving one component for each state-changing object can produce highly-coupled components, which are difficult to distribute, maintain and adapt. The predicted coupling information allows the developer to take preventive actions to arrive at a better decomposition of the final implementation. Although our approach has been demonstrated using BSMs, it is possible to generalize it to other languages suitable for object-centric process implementation.

We are currently extending the approach with the prediction of cohesion and complexity metrics. We expect that the incorporation of these two metrics with coupling will not only offer deeper insights into object-centric implementations, but will also facilitate a comparison of activity-centric and object-centric implementation approaches.

References

1. Business Process Execution Language for Web Services, Version 1.1. Joint specification by BEA, IBM, Microsoft, SAP and Siebel Systems (2003)
2. SCA Service Component Architecture, Assembly Model Specification, SCA Version 1.00, Open SOA Collaboration Specification (March 2007)
3. Business Process Modeling Notation Specification, V1.1., formal/2008-01-17. OMG Document (January 2008)
4. Abreu, F.B., Pereira, G., Sousa, P.: A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems. In: Proc. of the Conf. on Software Maintenance and Reengineering, pp. 13–22. IEEE Computer Society, Los Alamitos (2000)
5. Beers, G., Carey, J.: WebSphere Process Server Business State Machines concepts and capabilities, Part 1: Exploring basic concepts. IBM developerWorks (October 2006)

6. Bhattacharya, K., Guttman, R., Lyman, K., et al.: A Model-Driven Approach to Industrializing Discovery Processes in Pharmaceutical Research. IBM Systems Journal 44(1), 145–162 (2005)
7. Briand, L.C., Daly, J.W., Wüst, J.: A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering 25(1), 91–121 (1999)
8. Lee, J.K., Seung, S.J., Kim, S.D., Hyun, W., Han, D.H.: Component Identification Method with Coupling and Cohesion. In: Proc. of the 8th Asia-Pacific Conf. on Software Engineering, pp. 79–86. IEEE Computer Society, Los Alamitos (2001)
9. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)
10. Müller, D., Reichert, M., Herbst, J.: Flexibility of Data-Driven Process Structures. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 181–192. Springer, Heidelberg (2006)
11. Müller, D., Reichert, M., Herbst, J.: Data-Driven Modeling and Coordination of Large Process Structures. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 131–149. Springer, Heidelberg (2007)
12. Muth, P., Wodtke, D., Weißenfels, J., Kotz Dittrich, A., Weikum, G.: From Centralized Workflow Specification to Distributed Workflow Execution. Journal of Intelligent Information Systems 10(2), 159–184 (1998)
13. Nanda, M.G., Chandra, S., Sarkar, V.: Decentralizing Execution of Composite Web Services. In: Proc. of the 19th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp. 170–187. ACM, New York (2004)
14. Nandi, P., Kumaran, S.: Adaptive Business Object - A New Component Model for Business Integration. In: Proc. of the 8th Int. Conf. on Enterprise Information Systems, pp. 179–188 (2005)
15. Nigam, A., Caswell, N.S.: Business Artifacts: An Approach to Operational Specification. IBM Systems Journal 42(3), 428–445 (2003)
16. Reijers, H.A., Limam, S., van der Aalst, W.M.P.: Product-Based Workflow Design. Journal of Management Information Systems 20(1), 229–262
17. Reijers, H.A., Vanderfeesten, I.T.P.: Cohesion and Coupling Metrics for Workflow Process Design. In: Desel, J., Pernici, B., Weske, M. (eds.) BPM 2004. LNCS, vol. 3080, pp. 290–305. Springer, Heidelberg (2004)
18. Ryndina, K., Küster, J.M., Gall, H.: Consistency of Business Process Models and Object Life Cycles. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 80–90. Springer, Heidelberg (2007)
19. Ryndina, K., Küster, J.M., Gall, H.: A Tool for Integrating Object Life Cycle and Business Process Modeling. In: Proc. of the BPM Demonstration Program at the 5th Int. Conf. on Business Process Management. CEUR-WS (2007)
20. van der Aalst, W.M.P., Barthelmess, P., Ellis, C.A., Wainer, J.: Proclets: A Framework for Lightweight Interacting Workflow Processes. International Journal of Cooperative Information Systems 10(4), 443–481 (2001)
21. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
22. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Analysis of Web Services Composition Languages: The Case of BPEL4WS. In: Song, I.-Y., Liddle, S.W., Ling, T.-W., Scheuermann, P. (eds.) ER 2003. LNCS, vol. 2813, pp. 200–215. Springer, Heidelberg (2003)