Achieving Recovery in Service Composition with Assurance Points and Integration Rules

Short Paper

Susan D. Urban, Le Gao, Rajiv Shrestha, and Andrew Courter
Texas Tech University, Department of Computer Science
Lubbock, TX 79409 +1-806-742-2484
{susan.urban | le.gao | rajiv.shrestha | s.courter} @ ttu.edu

Abstract. This paper defines the concept of Assurance Points (APs) together with the use of integration rules to provide a flexible way of checking constraints and responding to execution errors in service composition. An AP is a combined logical and physical checkpoint, providing an execution milestone that stores critical data and interacts with integration rules to alter program flow and to invoke different forms of recovery depending on the execution status. During normal execution, APs invoke rules that check pre-conditions, post-conditions, and other application rules. When execution errors occur, APs are also used as rollback points. Integration rules can invoke backward recovery to specific APs using compensation as well as forward recovery through rechecking of preconditions before retry attempts or through execution of contingencies and alternative execution paths. APs together with integration rules provide an increased level of consistency checking as well as backward and forward recovery actions.

Keywords: service composition; data consistency, recovery, compensation, contingency, retry, checkpoints.

1 Introduction

Web Services and service-oriented computing are becoming widely used for business-to-business integration. Prevalent techniques [7, 19, 24] have been widely adopted for process modeling, with execution engines based on standards such as the Business Process Execution Language (BPEL) [10] providing a framework for execution of processes composed of services. Service composition for business integration, however, creates challenges for traditional process modeling techniques.

In a service execution environment, a process must be flexible enough to respond to errors, exceptions, and interruptions. Backward and forward recovery mechanisms [13] can be used to respond to such events. For example, compensation is a backward recovery mechanism that performs a logical undo operation. Contingency is a forward recovery mechanism that provides an alternative execution path to keep a process running. Nevertheless, most service composition techniques do not provide flexibility

with respect to the combined use of compensation and contingency. Service composition models need to be enhanced with features that allow processes to assess their execution status to support more dynamic ways of responding to failures, while at the same time validating correctness conditions for process execution.

This paper presents our investigation of *Assurance Points* (APs) and *integration rules* to provide a more flexible way of checking constraints and responding to execution failures. An AP is a combined logical and physical checkpoint. As a physical checkpoint, an AP provides a way to store data at critical points in the execution of a process. Unlike past work with checkpointing [6, 15], our work focuses on the use of APs for user-defined consistency checking and rollback points that can be used to maximize forward recovery options when failures occur. In particular, an AP provides an execution milestone that interacts with integration rules. The data stored at an AP is passed as parameters to integration rules that are used to check pre-conditions, post-conditions, and other application conditions. Failure of a pre or post-condition or the failure of a service execution can invoke several different forms of recovery, including backward recovery of the entire process, retry attempts, or execution of contingent procedures. The unique aspect of APs is that they provide intermediate rollback points when failures occur that allow a process to be compensated to a specific AP for the purpose of rechecking pre-conditions before retry attempts or the execution of contingent procedures.

In this paper, we describe the interaction among APs, integration rules, and the different forms of recovery actions as defined in [20], illustrating the functionality of these concepts using an online shopping scenario. We then provide a comparison of our approach to the fault handling and recovery procedures in BPEL [11]. The primary contribution of our work is found in the explicit support provided for user-defined constraints, with rule-driven recovery actions for compensation, retry, and contingency procedures that support flexibility with respect to the combined use of backward and forward recovery options.

2 Related Work

From a historical point of view, our work is founded on past work with Advanced Transaction Models (ATMs). ATMs provide better support for Long Running Transactions (LRTs) that need relaxed atomicity and isolation properties [4]. In the work of [8], sagas were defined as a mechanism to structure long running processes, with each sub-transaction having a compensating procedure to reverse the affects of the saga when it fails. Other advanced transaction models have also made use of compensation for hierarchically structured transactions [18].

More recently, events and rules have been used to dynamically specify control flow and data flow in a process by using Event Condition Action (ECA) rules [17]. ECA rules have also been successfully implemented for exception handling in work such as [2, 14]. The work in [14] uses ECA rules to generate reliable and fault-tolerant BPEL processes to overcome the limited fault handling capability of BPEL. Our work with assurance points also supports the use of rules that separate fault handling from

normal business logic. Combined with assurance points, integration rules are used to integrate user-defined consistency constraints with the recovery process.

Several efforts have been made to enhance the BPEL fault and exception handling capabilities. BPEL4Job [21] addresses fault-handling design for job flow management with the ability to migrate flow instances. The work in [16] proposes mechanisms like external variable setting, future alternative behavior, rollback and conditional re-execution of the Flow, timeout, and redo mechanisms for enabling recovery actions using BPEL. The Dynamo [1] framework for the dynamic monitoring of WS-BPEL processes weaves rules such as pre/post conditions and invariants into the BPEL process. Most of these projects do not fully integrate constraint checking with a variety of recovery actions as in our work to support more dynamic and flexible ways of reacting to failures. Our research demonstrates the viability of variegated recovery approaches within a BPEL-like execution environment.

In checkpointing systems, consistent execution states are saved during the process flow. During failures and exceptions, the activity can be rolled back to the closest consistent checkpoint to move the execution to an alternative platform [6, 15]. The AP concept presented in this paper also stores critical execution data, but uses the data as parameters to rules that perform constraint checking and invoke different types of recovery actions.

The work in [3] illustrates the application of aspect-oriented software development concepts to workflow languages to provide flexible and adaptable workflows. AO4BPEL [4] is an aspect-oriented extension to BPEL that uses AspectJ to provide control flow adaptations [12]. Assurance Points are similar to aspect-oriented programming but are more fully integrated into the process execution engine for support of recovery actions.

3 Service Composition and Recovery with Assurance Points

This section summarizes the service composition and recovery model from [25]. We then define and illustrate the use of assurance points and integration rules in the context of this model.

3.1 Overview of the Model

In [25], a process is defined as a top-level execution entity that is composed of other execution entities. A process is denoted as p_i, where p represents a process and the subscript i represents a unique identifier of the process. An operation represents a service invocation, denoted as $op_{i,j}$, such that op is an operation, i identifies the enclosing process p_i, and j represents the unique identifier of the operation within p_i. Compensation ($cop_{i,j}$) is an operation intended for backward recovery, while contingency ($top_{i,j}$) is an operation used for forward recovery.

Atomic groups and composite groups are logical execution units that enable the specification of processes with complex control structure, facilitating service execution failure recovery by adding scopes within the context of a process execution. An atomic group (denoted $ag_{i,j}$) contains an operation, an optional compensation, and

an optional contingency. A composite group (denoted $cg_{i,k}$) may contain multiple atomic groups, and/or multiple composite groups that execute sequentially or in parallel. A composite group can have its own compensation and contingency as optional elements. Contingency is always tried first upon the failure of an atomic or composite group. Compensation, on the other had, is a recovery activity that is only applied as a way to reverse the effects of completed atomic and composite groups.

Figure 1 shows an online shopping process that is used as an example in the remainder of the paper. The process is composed of three composite groups (cg_1, cg_2 and cg_3). The group cg_1 contains four atomic groups and a compensating procedure ($cg1.cop$) that is attached to the entire group, which is known as shallow compensation. Shallow compensation involves the execution of a compensating procedure that will reverse the effects of the entire composite group. In comparison, cg_2 is composed of two atomic groups, where each atomic group has its own compensating procedure ($ag_{21}.cop$ and $ag_{22}.cop$), which is known as deep compensation. Deep compensation involves the execution of compensating procedures for each group within a composite group. In addition, the atomic group ag_{21} also has a contingent procedure ($ag_{21}.top$) that will be executed if ag_{21} fails. The composite group cg_3 also has a contingent procedure ($cg_3.top$). The reader should refer to [25] for a more formal presentation of the recovery semantics for shallow compensation, deep compensation, and contingency in the context of the service composition model.

3.2 Assurance Point and Rule Extensions

Our work has extended the model described in the previous section with the concept of assurance points, which are depicted in Figure 1 as ovals. An AP is a process execution correctness guard as well as a potential rollback point during the recovery process. Given that concurrent processes do not execute as traditional transactions in a service-oriented environment, inserting APs at critical points in a process is important for checking consistency constraints and potentially reducing the risk of failure or inconsistent data. An AP also serves as a milestone for backward and forward recovery activities. When failures occur, APs can be used as rollback points for backward recovery, rechecking pre-conditions relevant to forward recovery. In the current version of our work, we assume that APs are placed at points in a process where they are only executed once, and not embedded in iterative control structures.

An AP is defined as: $AP = <apId, apParameters^*, IR_{pre}?, IR_{post}?, IR_{cond}^*>$, where $apID$ is the unique identifier of the AP, $apParameters$ is a list of critical data items to be stored as part of the AP, IR_{pre} is an integration rule defining a pre-condition, IR_{post} is an integration rule defining a post-condition, and IR_{cond} is an integration rule defining additional application rules. In the above notation, * indicates 0 or more occurrences, while ? indicates zero or one optional occurrences.

IR_{pre}, IR_{post}, and IR_{cond} are expressed as Event-Condition-Action (ECA) rules based on the use of integration rules to interconnect software components [9, 22]. An IR is triggered by a process reaching an AP during execution, where an AP serves as the event of an integration rule. Upon reaching an AP, the condition of an IR is evaluated.

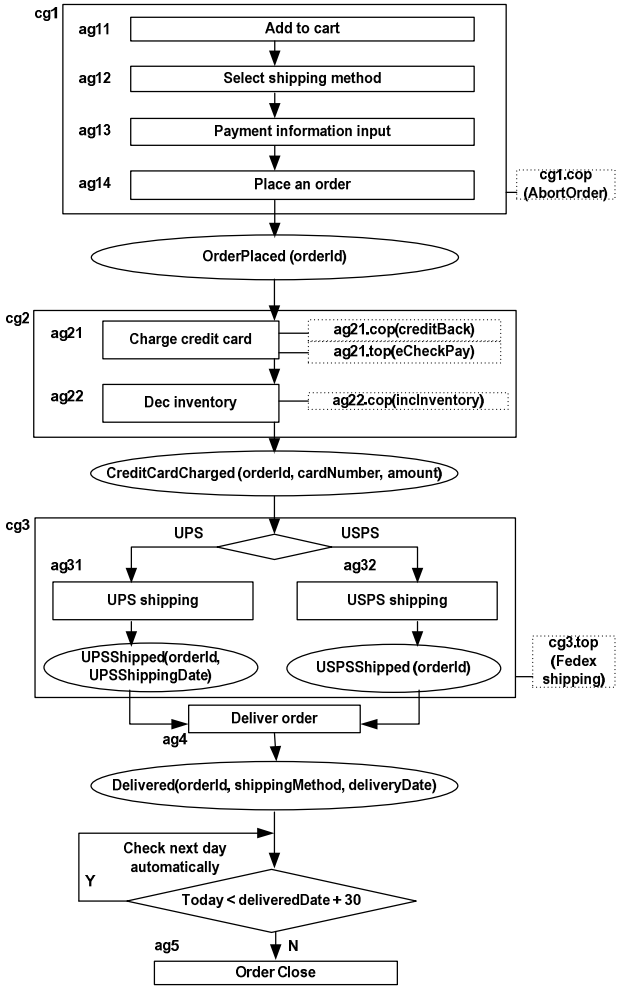

Fig. 1. Online Shopping Process with APs

The action specification is executed if the rule condition evaluates to true. For IR_{pre} and IR_{post}, a constraint C is always expressed in a negative form (not(C)). An action (action 1) is invoked if the pre or post condition is not true, invoking a recovery action or an alternative execution path. If the specified action is a retry activity, then there is a possibility for the process to execute through the same pre or post condition a second time. As a result, integration rules support the capability of specifying a second action (action 2).

In its most basic form, a recovery action simply invokes an alternative process. Recovery actions can also be one of the following actions:

- **APRollback**: APRollback is used when the entire process needs to compensate its way back to the start of the process according to the semantics of the service compensation model.

- **APRetry**: APRetry is used when the running process needs to be backward recovered using compensation to a specific AP. By default, the backward recovery process will go to the first AP reached as part of the shallow or deep compensation process within the same scope. The pre-condition defined in the AP is re-checked. If the pre-condition is satisfied, the process execution is resumed from that AP by re-trying the recovered operations. Otherwise, the action of the pre-condition rule is executed. The APRetry command can optionally specify a parameter indicating the AP that is the target of the backward recovery process.
- **APCascadedContingency (APCC)**: APCC is a backward recovery process that searches backwards through the hierarchical nesting of composite groups to find a possible contingent procedure for a failed composite group. During the APCC backward recovery process, when an AP is found before a composite group, the pre-condition defined in the AP will be re-checked before invoking any contingent procedures for forward recovery. APC is specifically used as a means of forward recovery from nested composite groups.

The most basic use of an AP together with integration rules is shown in Figure 2, which shows a process with three composite groups and an AP between each composite group. The shaded box shows the functionality of an AP using AP2 as an example. Each AP serves as a checkpoint facility, storing execution status data in a checkpoint database (AP data in Figure 2). When the execution reaches AP2, IRs associated with the AP are invoked. The condition of an IR_{post} is evaluated first to validate the execution of cg_2. If the post-condition is violated, the action invoked can be one of the pre-defined recovery actions as described above. If the post-condition is not violated, then an IR_{pre} rule is evaluated to check the pre-condition for the next service execution. If the pre-condition is violated, one of the pre-defined recovery actions will be invoked. If the pre-condition is satisfied, the AP will check for any additional, conditional rules (IR_{cond}) that may have been expressed. IR_{cond} rules do not affect the normal flow of execution but provide a way to invoke additional parallel activity based on application requirements. Note that the expression of a pre-condition, post-condition or any additional condition is optional.

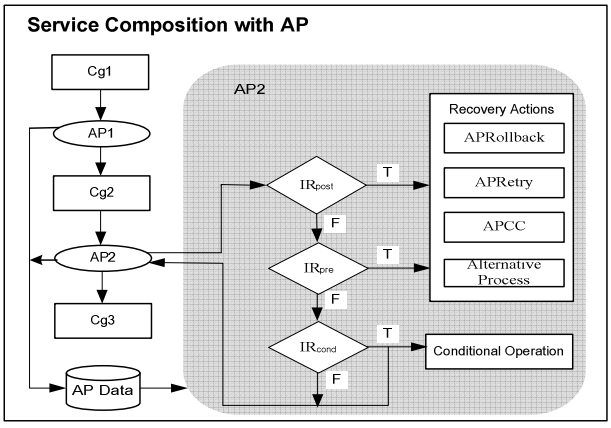

Fig. 2. Basic Use of AP and Integration Rules

3.3 Case Study: Assurance Points and Rules

This section provides an example of assurance points, integration rules, and conditional rules using the online shopping application in Figure 1. APs are shown as ovals between composite and/or atomic groups, where each AP has a name and a list of parameters that are stored at the AP.

Table 1 shows rules associated with the APs in Figure 1. The orderPlaced AP follows the execution of cg_1, which is a composite process that supports adding items to a cart, selecting a shipping method, entering payment information, and submitting an order. The orderPlaced AP marks the transition from placing the order to actually processing the order. The AP has a pre-condition called the QuantityChecked1 rule that serves as verification that the store has enough goods in stock to proceed with the order. The condition validates the status of the inventory and, if the availability has changed since the customer began adding items to the cart, the rule invokes the backOrderPurchase process. In this case, the rule does not invoke one of the predefined recovery actions but, instead, invokes an alternate execution path.

If the process passes the pre-condition verification, the process then executes cg_2, which charges the customer's credit card and decrements the inventory. The composite group is followed by the CreditCardCharged AP, which has a post-condition that further guarantees the in-stock quantity is greater than zero after the inventory has been decremented. If the post-condition is violated, the APRetry action is invoked. APRetry will perform a logical rollback of cg_2 by performing deep compensation (i.e., executing ag_{22}.cop followed by ag_{21}.cop). According to the retry semantics, when the process reverses itself to the orderPlaced AP, the pre-condition of the orderPlaced AP will be rechecked. The retry of cg_2 will only be allowed if the pre-condition is satisfied. Otherwise, the backOrderPurchase process will be invoked. In the case where the precondition of the orderPlaced AP is satisfied, the cg_2 process will be re-executed. If the post-condition of the orderPlaced AP fails a second time, the entire process will go through a rollback process by performing deep compensation on cg_2 and shallow compensation on cg_1. Note that in Table 1, the CreditCardCharged AP also has a conditional rule that sends a message notification for large charges.

The APCC recovery action can be specified as a rule action, but it is also the default action to take when the execution of an atomic group and the contingency of an atomic group fails. As an example, suppose the process is executing inside cg_3 and fails during the execution of UPSshipping. Since there is no contingency attached to UPSshipping, the process will enter APCC mode, which attempts to reverse the process to the beginning of the most enclosing composite group, which in this case is cg_3. The APCC recovery process will then execute the contingency of the composite group (cg_3.top). The APCC recovery process will alway check for an AP with a precondition and test the precondition before executing a contingent procedure. The APCC recovery process provides a well-defined procedure for backing out of nested composite groups and checking for contingent, forward recovery procedures.

Since no pre or post condition is specified for the Delivered AP, only the conditional rule shippingRefund is evaluated. Assume the delivery method was overnight through UPS with an extra shipping fee. If UPS has delivered the item on time, then the Delivered AP is complete and execution continues. Otherwise,

refundUPSShippingCharge is invoked to refund the extra fee while the main process execution continues.

Table 1. AP Rules in the Online Shopping Process

Integration Rule	Conditional Rule
create rule QuantityCheck1::pre **event:** ***OrderPlaced*** (orderId) **condition:** exists(select L.itemId from Inventory I, LineItem L where L.orderId=orderId and L.itemId=I.itemId and L.quantity>I.quantity) **action:** backOrderPurchase(orderId)	**create rule** Notice::cond **event:** ***CreditCardCharged*** (orderId, cardNumber , amount) **condition:** amount > $1000 **action:** highExpenseNotice(cardNumber)
create rule QuantityCheck2::post **event:** ***CreditCardCharged*** (orderId, cardNumber, amount) **condition:** exists(select L.itemId from Inventory I, LineItem L where L.orderId=orderId and L.itemId=I.itemId and I.quantity<0) **action1**: APRetry **action2:** APRollback	**create rule** ShippingRefund::cond **event**: ***Delivered*** (orderId, shippingMethod, deliveryDate) **condition:** shippingMethod = UPS && deliveryDate != UPSShipped.UPSShippingDate+1 **action:** refundUPSShippingCharge(orderId)

4 Comparison to Recovery in BPEL

We have developed a prototype implementation of the AP concept and also conducted a comparison of our work with BPEL. The work in [11] highlights the two main problems with the BPEL fault and compensation mechanism: 1) compensation order can violate control link dependencies if control links cross the scope boundaries, and 2) high complexity of default compensation order due to default handler behavior. Unlike BPEL, the order of the AP compensation approach is clear since APs support a hierarchical process structure and do not support control links between non-peer scopes, making the semantics of compensation in the AP approach unambiguous.

In general, the notion of compensation should also be capable of handling constraint violations [5]. Since BPEL's compensation handling mechanism through the <compensate> activity can only be called inside a fault handler, this limits the ability to call compensation outside a fault handling. In the case of the AP model, compensation can be invoked during normal execution (no error has yet occurred) when integration rules are not satisfied. This allows a flexible way to recover the process through compensation in response to constraint violations.

BPEL does not explicitly support a contingency feature other than fault, exception, and termination handlers. The designer is responsible for complex fault handling logic, which, as pointed out in [5, 11] has the potential to increase complexity and create unexpected errors. The AP model provides explicit contingency activities so that forward recovery is possible.

5 Conclusions and Future Directions

This research has defined the use of assurance points, integration rules, and recovery actions to 1) provide a way of expressing user-defined constraints for process execution and 2) provide greater flexibility for use of forward and backward recovery options when constraints are not satisfied or execution fails. Assurance points enhance traditional work with checkpointing, providing logical points for backward recovery with semantics that increase the potential for forward recovery by rechecking pre-conditions, retrying services, and looking for contingencies. Planning for failure and recovery should be an important part of every process specification.

There are several directions for future work. We are currently extending the recovery algorithms to support parallel execution as in the flow activity of BPEL. Another direction involves formalization of the assurance point concept using Petri-nets and model-checking. Methodological issues for the specification of APs, integration rules, and recovery procedures will also be addressed, together with refinement of recovery actions for concurrent and iterative activity. We are also investigating the integration of assurance points with our work on decentralized data dependency analysis [23] in Process Execution Agents (PEXAs), where PEXAs communicate about data dependencies so that when one process fails and recovers, other data dependent processes can be notified of potential data inconsistencies. The AP concept can be used to enhance decentralized PEXAs with greater flexibility for process recovery options.

Acknowledgments. This research has been supported by NSF Grant CCF-0820152. Opinions, findings, conclusions or recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of NSF.

References

1. Baresi, L., Guinea, S., Pasquale, L.: Self-Healing BPEL Processes with Dynamo and the JBoss Rule Engine. ACM Int. Workshop on Eng. of Software Services for Pervasive Environments, pp. 11-20. ACM New York, (2007)
2. Brambilla, M., Ceri, S., Comai, S., Tziviskou, C.: Exception Handling in Workflow-Driven Web Applications. Proc. of the14th Int. Conf. on World Wide Web, pp. 170-179. ACM New York, (2005)
3. Charfi, A., Mezini, M.: Aspect-Oriented Workflow Languages. Lecture Notes in Computer Science 4275, 183 (2006)
4. Cichocki, A.: Workflow and Process Automation: Concepts and Technology. Kluwer Academic Pub (1998)
5. Coleman, J.: Examining BPEL's Compensation Construct. Workshop on Rigorous Eng. of Fault-Tolerant Systems (2005)
6. Dialani, V., Miles, S., Moreau, L., De Roure, D., Luck, M.: Transparent Fault Tolerance for Web Services Based Architectures. Lecture Notes in Computer Science 889-898 (2002)
7. Engels, G., Förster, A., Heckel, R., Thöne, S.: Process Modeling using UML.

Process-Aware Information Systems: Bridging People and Software through Process Technology. Hoboken, New Jersey: Wiley 85-117 (2005)
8. Garcia-Molina, H., Salem, K.: Sagas. Morgan Kaufmann Publishers Inc., (1994)
9. Jin, Y.: An Architecture and Execution Environment for Component Integration Rules. Ph.D. Diss., Arizona State University (2004)
10. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y.: Web Services Business Process Execution Language Version 2.0. OASIS Standard 11, (2007)
11. Khalaf, R., Roller, D., Leymann, F.: Revisiting the Behavior of Fault and Compensation Handlers in WS-BPEL. On the Move to Meaningful Internet Systems: OTM 2009, 286-303 (2009)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. Lecture Notes in Computer Science 327-353 (2001)
13. Lee, P.A., Anderson, T., Laprie, J.C., Avizienis, A., Kopetz, H.: Fault Tolerance: Principles and Practice. Springer-Verlag New York (1990)
14. Liu, A., Li, Q., Huang, L., Xiao, M.: A Declarative Approach to Enhancing the Reliability of BPEL Processes, Proc. Of the Int. Conf. on Web Services, pp. 272-279. (2007)
15. Luo, Z.W.: Checkpointing for Workflow Recovery. Proc. of the 38th Annual Southeast Regional Conf., pp. 79-80. ACM New York, (2000)
16. Modafferi, S., Conforti, E.: Methods for Enabling Recovery Actions in WS-BPEL. Lecture Notes in Computer Science 4275, 219 (2006)
17. Paton, N.W., Díaz, O.: Active Database Systems. ACM Computing Surveys 31, (1999)
18. Rolf, A., Klas, W., Veijalainen, J.: Transaction Management Support for Cooperative Applications. Kluwer Academic Pub (1997)
19. Scheer, A.W., Thomas, O., Adam, O.: Process Modeling Using Event-driven Process Chains. Process-aware Information Systems: Bridging People and Software through Process Technology. New Jersey: Wiley 119-145 (2005)
20. Shrestha, R.: Using Assurance Points, Events, and Rules for Recovery in Service Composition. M.S. Thesis, Texas Tech University (2010)
21. Tan, W., Fong, L., Bobroff, N.: Bpel4job: A Fault-Handling Design for Job Flow Management. Lecture Notes in Computer Science 4749, 27 (2007)
22. Urban, S.D., Dietrich, S.W., Na, Y., Jin, Y., Sundermier, A., Saxena, A.: The IRules Project: Using Active Rules for the Integration of Distributed Software Components. Proc. of the 9th IFIP Working Conf. on Database Semantics: Semantic Issues in E-Commerce Systems, pp. 265-286. (2001)
23. Urban, S.D., Liu, Z.A., Gao, L.: Decentralized Data Dependency Analysis for Concurrent Process Execution. Proc. of the 13th Enterprise Dist. Object Computing Conf. Workshops (Edocw 2009) 74-83 (2009)
24. White, S.A.: Business Process Modeling Notation (BPMN). URL http://www. bpmi. org/bpmi-downloads/BPMN-V1. 0. pdf (2004)
25. Xiao, Y., Urban, S.D.: The DeltaGrid Service Composition and Recovery Model. Int. Journal of Web Services Research (2009)