# DSCWeaver: Synchronization-Constraint Aspect Extension to Procedural Process Specification Languages

Qinyi Wu[1], Calton Pu[1], Akhil Sahai[2], Roger Barga[3], Gueyoung Jung[1]
Jason Parekh[1], Galen Swint[1]

[1]College of Computing, Gatech
{qxw, calton, galen.swint, jason.parekh, gueyoung.jung }@cc.gatech.edu

[2]HP Labs
akhil.sahai@hp.com

[3]Microsoft Research
barga@microsoft.com

## Abstract

*BPEL is emerging as an open-standards language for Web service composition. However, its procedural style can lead to inflexible and tangled code for managing a crosscutting aspect — synchronization constraints that define permissible sequences of execution for activities in a process. In this paper, we present DSCWeaver, a tool that enables a synchronization-aspect extension to BPEL. It uses DSCL, a synchronization expression language, to specify constraints. DSCL has the desirable features of declarative syntax, fine granularity, and validation support. A designer can use DSCL to describe and validate the synchronization behavior and rely on DSCWeaver to generate BPEL code. We demonstrate the advantages of our approach in a service deployment process and evaluate its performance using two metrics: Lines of Code (LoC) and Places to Visit (PtV). Evaluation results show that our approach can effectively reduce development effort of process designers while providing performance competitive to un-woven BPEL code.*

## 1. Introduction

The recent trend in web service composition languages is to specify the structure of a process using control constructs, such as And-split and And-join [25]. However, their procedural style may lead to inflexible and tangled code in process specification for two reasons. First, the procedural style is not effective in capturing complex synchronization behavior. For example, in BPEL the `<link>` construct results in scattered code among activities nested in different concurrent subprocesses [22]. Second, the procedural style results in centralized synchronization control. It has been shown that centralized control may degrade process performance and increase network load of the orchestration server [6]. In sharp contrast, decentralized control has the advantage of balancing workload among peers.

In this paper, we address these problems by applying Aspect-Oriented Programming (AOP) techniques to the problem of synchronization constraints on process models. The advantages for modeling a process from multiple aspects have been identified by Schmidt and Assmann as simplified modeling complexity and increased robustness of the process during adaptation [20]. The utility of AOP on domain specific languages is also discussed in [9][17]. Our evaluation results further confirm their observations.

We introduce DSCWeaver, a tool that offers a synchronization-aspect extension to BPEL, a popular Web service composition language. BPEL may become verbose and complex when modeling nontrivial processes [2][8]. One contributing factor in our observation is synchronization code that crosscuts the procedural modeling code.

DSCWeaver has two unique features. First, synchronization constraints for a process are specified in the DAG Synchronization Constraint Language (DSCL) [21]. This detangles synchronization code from the base code of a process and provides flexible and expressive primitives to describe synchronization relationships. DSCL draws on synchronization research from parallel programming [5][19]. As its name indicates, DSCL can be used to specify a Directed Acyclic Graph (DAG) flow model. It has a declarative syntax, fine granularity, and validation support. DSCL defines three synchronization relations (*HappenBefore*, *HappenTogether*, and *Exclusive*) that operate on activity states (*start*, *run,* and *finish*). By specifying relationships over activity states, DSCL can describe a rich set of synchronization behaviors.

The second unique feature is that synchronization constraints written in DSCL are automatically translated into a set of messages tokens carrying the synchronization data. Activities synchronize with each other by exchanging these tokens asynchronously. The translation relies on an intermediate Petri net representation [13]. DSCWeaver uses a Petri net for two reasons. One is to simulate and validate the synchronization constraints. The other reason is to map the transition firing logic of the Petri net into a set of token messages. The advantage of this approach is that the syntax of token messages is language-independent and can be woven into any service composition language that supports a messaging mechanism. What is more, messages can be easily exchanged among distributed processes, which facilitate conversion from centralized control to decentralized control for processes orchestration.

The rest of the paper is organized as follows. In Section 2, we present an example to illustrate the synchronization constraints in a nontrivial service deployment process. In Section 3, we present an overview of the DSCWeaver implementation and explain its major modules. We then give a brief introduction to DSCL, followed by a description of the translation from synchronization constraints to token messages. In Section 4, we explain how to apply DSCWeaver to BPEL. In Section 5, we revisit the example in Section 2 and show our evaluation results. Related work and conclusion are presented last.

## 2. A Motivating Example

Consider a service deployment process for the PetStore e-commerce application. It is an online store where customers can browse and purchase their favorite pets. The PetStore application consists of a database tier and an application server tier. To meet performance goals, the database server and the application server are installed on different hosts. The deployment consists of a set of installation activities, each represented by $\boldsymbol{a}$, each of which interacts with its target host to perform part of the installation task. The deployment process consists of three subprocesses:

1) **Middleware installation**: It includes installing the database (*MySQL*), runtime environment (*Java*, *Ant*) and application server (*Tomcat*) denoted $\boldsymbol{a}_{\text{sql}}$, $\boldsymbol{a}_{\text{java}}$, $\boldsymbol{a}_{\text{ant}}$, and $\boldsymbol{a}_{\text{tomcat}}$ respectively.
2) **Application installation**. It includes installing the application (*PetStore*) and its dependent libraries (*Jdbc*, *Struts*, *Dao*, and *SQLMap*), propagating database with PetStore workload data (*configure MySql*) and configuring PetStore with database server information (*configure PetStore*) denoted $\boldsymbol{a}_{\text{petstore}}$, $\boldsymbol{a}_{\text{jdbc}}$, $\boldsymbol{a}_{\text{struts}}$, $\boldsymbol{a}_{\text{dao}}$, $\boldsymbol{a}_{\text{sqlmap}}$, $\boldsymbol{a}_{\text{c\_sql}}$, and $\boldsymbol{a}_{\text{c\_petstore}}$ respectively.
3) **Application ignition**. It includes starting the database and the application server denoted $\boldsymbol{a}_{\text{s\_sql}}$ and $\boldsymbol{a}_{\text{s\_tomcat}}$.

The deployment process is orchestrated by a BPEL engine on the deployment host. The target hosts for the database and the application server are preconfigured with a web service, InstallWS, which accepts installation instructions from activities and performs the corresponding tasks. Figure 1a depicts the centralized control scenario. In this architecture, all synchronization logic is managed by the deployment machine (Host A), which interacts with target machines (Host B and Host C) by sending them installation instructions in order. The disadvantage of this approach is that Host A may become overloaded by the synchronization traffic. However, it turns out that we can reduce a portion of workload from Host A by distributing deployment activities according to their target hosts. This modification leads to the decentralized deployment architecture in Figure 1b. In the decentralized approach, the process is split into two subprocesses, each of which is deployed on a target host and interacts with that host to fulfill the installation task. This means that all activities related to database installation form a subprocess to be deployed at Host B, while all activities related to application server installation form another subprocess to be deployed at Host C. These two subprocesses interact only with Host A to synchronize with each other. In Section 5, we provide run-time performance measurements for both architectures.

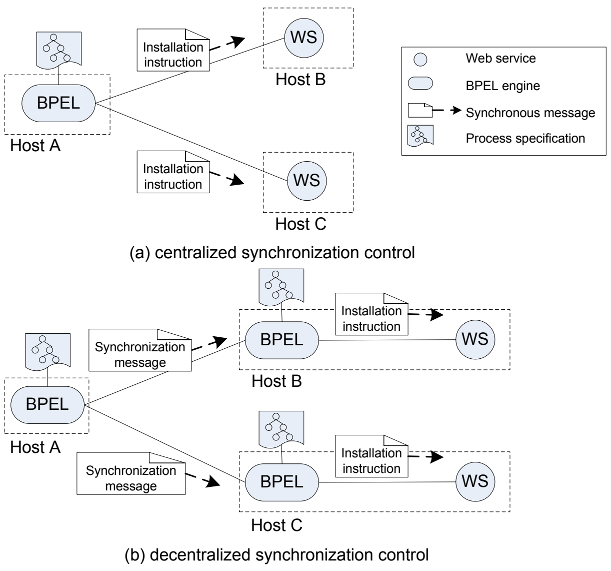


**Figure 1. PetStore deployment architecture**

Synchronization constraints are created either by installation dependencies or by user requirements. For instance, an installation dependency arises when the installation of one software package should be placed at certain location within the directory structure of another software package. An example is that PetStore code should be placed in the directory `$Tomcat/webapp`.

User requirements come from design strategy or other concerns. For example, a process designer may require that the installation subprocess should finish before the ignition subprocess starts.

Instead of describing the constraints procedurally, we use two relations for declaring constraints as relationship statements: *HappenBefore(*$\rightarrow$*)* and *HappenTogether(*$\leftrightarrow$*)*. Unlike other synchronization expression languages [5][19], these relations operate on activity state. An activity progresses through three states: *start(S)*, *run(R)* and *finish(F)*. The activities interact with each other as they transit from one state to the next subject to the relationship statements. We detail the relation and activity

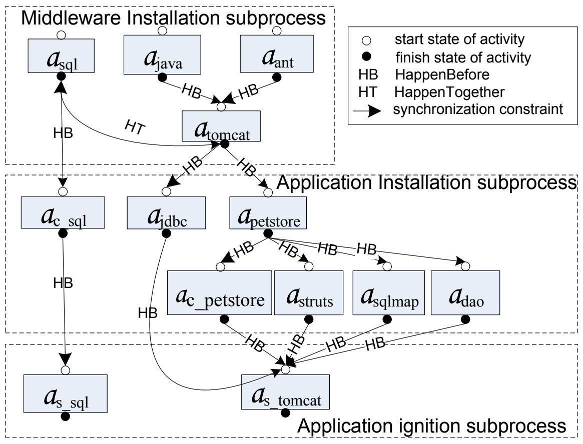


**Figure 2. Synchronization constraints in PetStore deployment process**

state in Section 3.2. Figure 2 illustrates the synchronization constraints of PetStore deployment.

For example, due to an installation dependency the *finish* state of $\boldsymbol{a}_{\text{sql}}$ should happen before the *start* state of $\boldsymbol{a}_{\text{c\_sql}}$, abbreviated as $\boldsymbol{F}_{\text{sql}}\rightarrow\boldsymbol{S}_{\text{c\_sql}}$. Furthermore, the designer requires that the runtime environment should be properly set up before installing application server, which adds $\boldsymbol{F}_{\text{java}}\rightarrow\boldsymbol{S}_{\text{tomcat}}$ and $\boldsymbol{F}_{\text{ant}}\rightarrow\boldsymbol{S}_{\text{tomcat}}$. He also requires that the middleware should be properly installed before going to application-dependent installation, which introduces another relationship $\boldsymbol{F}_{\text{sql}}\leftrightarrow\boldsymbol{F}_{\text{tomcat}}$.

## 3. DSCWeaver

### 3.1. Implementation Overview

DSCWeaver is an integrated tool written in Java for providing synchronization aspect extension to flow specification language. It contains several submodules to automate the translation from high-level specification to low-level implementation code. During the multi-stage translation process, intermediate outputs are formatted in XML for further processing. Figure 3 illustrates its architecture. The input to DSCWeaver is a process specification containing activities and their associated synchronization constraints in the form of state relationships. The process specification can be written in any host language appended by an extra section containing the synchronization information. DSCWeaver selects the corresponding *code weaver* for the host language. Below we briefly explain each submodule.

**State Relation to Petri Net (SR2PN).** This module takes the state relationships as input and translates them into Petri net. The Petri net is not only an input for the CPN/Tools for validation [18], but also an input for the next submodule PN2TM.

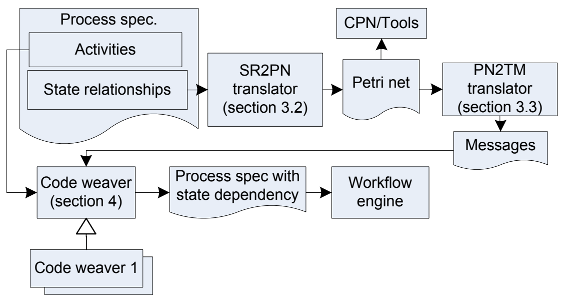


**Figure 3. DSCWeaver architecture overview**

**Petri Net to Token Message (PN2TM).** This maps the transition firing logic of a Petri net into token messages. These messages consist of token information like which transition is going to consume this token or to which transition this token should be delivered.

**Code Weaver.** The weaver collects information from the token messages about which messages must be sent or received for the state transitions of an activity. To weave the messages correctly, the activities are tagged in the original process specification. These tags are AOP joinpoints that introduce hooks in the base code [11]. We demonstrate this using BPEL in Section 4.

### 3.2. DAG Synchronization Constraint Language

In Section 2, we informally introduced DSCL's *HappenBefore* and *HappenTogether* relations. The design of DSCL adopts three features from parallel programming research [5][19]:

- Fine granularity. The life cycle of an activity is a sequence of states and can be synchronized with other activities based on its current state.
- Declarative syntax. A process designer only needs to specify *what* to be synchronized instead of *how* to implement it. This simplifies and accelerates service composition task [4].
- Validation support. A designer should have tools to assist the validation of synchronization behavior of processses, especially those that are complex or evolving.

*Fine granularity* is accomplished by synchronizing an activity at different states of its life cycle. An activity goes through three states: *start* (***S***), *run* (***R***) and *finish* (***F***). This brings more expressive power for synchronization behavior. For example, there are cases that a designer wants to express "*Activity B can not finish until Activity A starts*." The necessity of modeling activity at the granularity of state is discussed in [1].

DSCL *declarative syntax* defines three state relations. These three relations reflect the basic synchronization constraint, if any, between any pair of states.

- HappenBefore (→): the state at the beginning of the arrow should happen before the state at the end.
- HappenTogether (↔): the two states at both ends should be reached together.
- Exclusive (O): states at both ends must not be concurrent. Note that this only applies to *run* states because they are the only states where activities can actually interfere with each other.

By specifying synchronization relationships on activity states, we can express a rich set of synchronization behavior. For instance, the sequence is expressed as $\boldsymbol{F}_i \rightarrow \boldsymbol{S}_j$. The *And-split* and *And-join* are expressed as $\boldsymbol{S}_i \leftrightarrow \boldsymbol{S}_j$ and $\boldsymbol{F}_i \leftrightarrow \boldsymbol{F}_j$. DSCL is also able to describe the synchronization constraints such as $\boldsymbol{S}_i \rightarrow \boldsymbol{F}_j$, which are difficult to express in constructs available in existing workflow specification languages due to the atomicity of an activity. For example, it is not easy to enforce the constraint of "*Before finishing the activity of closing a purchasing order, the activity of customer satisfactory survey should have been started.*" As a concrete example, the synchronization constraints in the PetStore deployment in Section 2 are:

- Middleware installation
  $\boldsymbol{F}_{\text{java}} \rightarrow \boldsymbol{S}_{\text{tomcat}}$, $\boldsymbol{F}_{\text{ant}} \rightarrow \boldsymbol{S}_{\text{tomcat}}$, $\boldsymbol{F}_{\text{mysql}} \leftrightarrow \boldsymbol{F}_{\text{tomcat}}$
- Application installation
  $\boldsymbol{F}_{\text{sql}} \rightarrow \boldsymbol{S}_{\text{c\_sql}}$, $\boldsymbol{F}_{\text{tomcat}} \rightarrow \boldsymbol{S}_{\text{jdbc}}$, $\boldsymbol{F}_{\text{tomcat}} \rightarrow \boldsymbol{S}_{\text{petstore}}$, $\boldsymbol{F}_{\text{petstore}} \rightarrow \boldsymbol{S}_{\text{c\_petstore}}$, $\boldsymbol{F}_{\text{petstore}} \rightarrow \boldsymbol{S}_{\text{struts}}$, $\boldsymbol{F}_{\text{petstore}} \rightarrow \boldsymbol{S}_{\text{sqlmap}}$, $\boldsymbol{F}_{\text{petstore}} \rightarrow \boldsymbol{S}_{\text{dao}}$
- Application ignition
  $\boldsymbol{F}_{\text{c\_mysql}} \rightarrow \boldsymbol{S}_{\text{s\_mysql}}$, $\boldsymbol{F}_{\text{jdbc}} \rightarrow \boldsymbol{S}_{\text{s\_tomcat}}$, $\boldsymbol{F}_{\text{c\_petstore}} \rightarrow \boldsymbol{S}_{\text{s\_tomcat}}$, $\boldsymbol{F}_{\text{struts}} \rightarrow \boldsymbol{S}_{\text{s\_tomcat}}$, $\boldsymbol{F}_{\text{sqlmap}} \rightarrow \boldsymbol{S}_{\text{s\_tomcat}}$, $\boldsymbol{F}_{\text{dao}} \rightarrow \boldsymbol{S}_{\text{s\_tomcat}}$

DSCWeaver offers validation support by translating state relationships into a Petri net. There are three types of synchronization constraints: the *intrastate relation construction*, which manages the state relations within an activity, and the *interstate relation construction*, which establishes the state relations between activities, and the *exclusive relation construction*, which handles the exclusive state relation. During the mapping, a place in the Petri net represents a state of an activity. A transition represents the conditions that need to be satisfied before the activity can reach that state. The firing of a transition means that the states it depends on corresponding to the in-bound places have been reached. The activity can transmit to its next state and put a token in its out-bound places. The idea is illustrated in Figure 4. It gives an example of state relationships between two activities $a_i$ and $a_j$. The translated Petri net is formatted to the input of CPN/Tools. CPN/Tools is a graph editor and simulator of the Colored Petri Net (CPN) [10]. It provides toolkits to identify dead transitions, infinite occurrence sequences, etc. For more details, please refer to our earlier work [21].

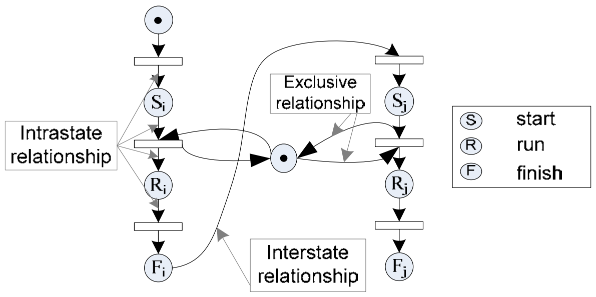
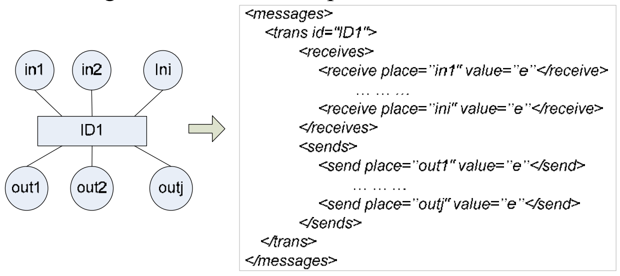


**Figure 4. Translation of state relationships to Petri net**

### 3.3. Translation of Petri Net to Token Messages

The PN2TM submodule translates the Petri net to a set of token messages that carry the Petri net firing information. There are two types of token messages: the *receive token message* (`<receive place=p value=e />`), which tells from which place *p* a transition receives a message with value *e*, and the *send token message* (`<send place=p value=e />`), which tells to which place *p* the transition send a message with value *e*. A transition should receive all inbound messages before sending outbound tokens. Figure 5 illustrates this procedure.



```
<messages>
  <trans id="ID1">
      <receives>
          <receive place="in1" value="e"</receive>
                ... ... ...
          <receive place="ini" value="e"</receive>
      </receives>
      <sends>
          <send place="out1" value="e"</send>
                ... ... ...
          <send place="outj" value="e"</send>
      </sends>
  </trans>
</messages>
```

**Figure 5. Translation from Petri net to token messages**

Each activity contains three transitions corresponding to its *start*, *run* and *finish* states. Each transition collects message information that is formatted in Figure 5. Translating the synchronization constraints to token messages is a distinctive feature of our approach. Unlike other work using formal technique for process verification [14][15], our approach augments formal techniques used to analyze the existing process with formal messages derivation that need to be exchanged to enforce the synchronization constraints.

There may be race conditions between activities. If the receiver is not active by the time the sender begins sending the message, it gets lost. We need a persistent queue for these messages until delivery to the receiver. Instead of direct communication, a sender sends its token to a queuing web service. When a receiver is ready to receive a particular message, it sends its request to the queue. If the message is present, the queue will forward it to the receiver. Otherwise the receiver blocks until the arrival of the message. We implemented a *PersistentQueue* web service that has two ports: *register*, which queues a message, and *query*, which accepts a query for the existence of a particular message and notifies the receiver when it becomes available. Notice that the *PersistentQueue* is slightly different from the standard message brokers in that the message delivery among queues is dependent. Only when all the messages corresponding to the inbound places of a transition have arrived will they be delivered together to the subscribers. In standard persistent message broker, the messages are delivered to the requester as soon as they arrive, independent of other messages.

## 4. BPEL Extension

In this section, we demonstrate how DSCWeaver operates on BPEL to implement the synchronization-aspect extension. We start with the explanation of extending BPEL with the DSCL syntax and then explain how the extended code can be woven into BPEL to form a complete process specification.

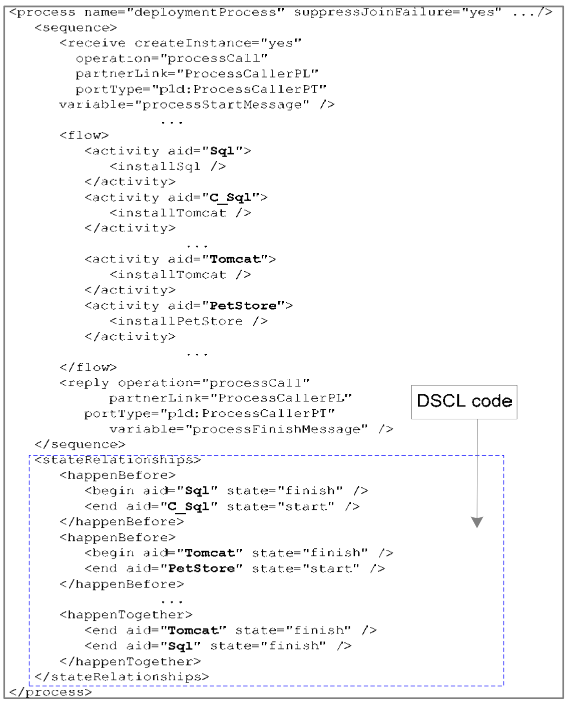
### 4.1. BPEL Syntax Extension

The BPEL specification is extended with new XML tags to express the state relationships among activities. We call it DSCL+BPEL, or DSCL+ for abbreviation when there is no confusion. Table 1 summarizes its syntax and semantics.

**Table 1. Tag extenstion for BPEL**

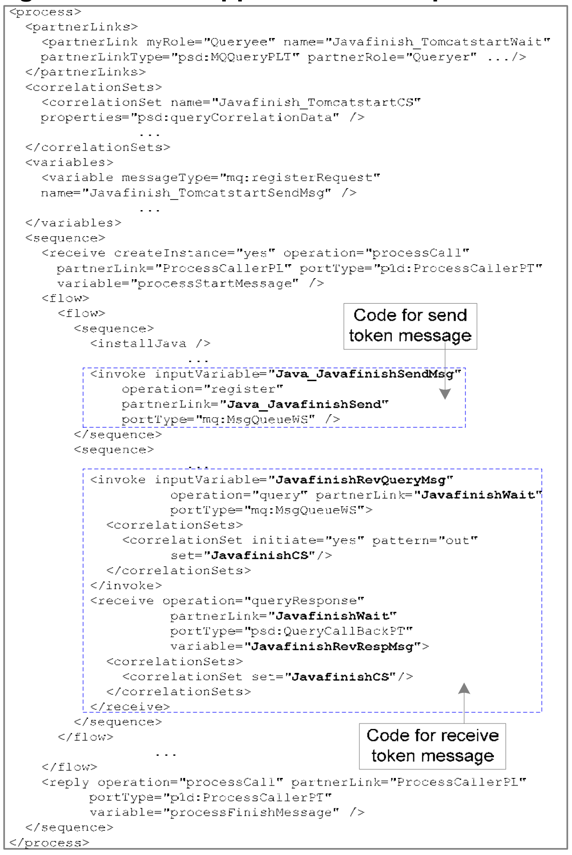
| Tag | Syntax | Semantics |
|---|---|---|
| activity | `<activity aid="qname">`<br>`Activity`<br>`</activity>` | It demarcates the boundary of an activity with a unique id. |
| HappenBefore | `<happenBefore>`<br>`<begin aid="A1" state="ncname" />`<br>`<end aid="A2" state="ncname" />`<br>`</happenBefore>` | It defines $\longrightarrow$. State is one of {start, finish} |
| HappenTogether | `<happenTogether>`<br>`<end aid="A1" state=" ncname " />`<br>… … …<br>`<end aid="A2" state=" ncname " />`<br>`</happenTogether >` | It defines $\longleftrightarrow$. State is one of {start, finish} |
| Exclusive | `< Exclusive >`<br>`< end aid="A1" state=" ncname " />`<br>… … …<br>`<end aid="A2" state=" ncname " />`<br>`</ Exclusive >` | It defines ◊. State is {run} |

A code snippet for PetStore's DSCL+BPEL specification is shown in Figure 6. We highlight the relevant activities for clarity.



```
<process name="deploymentProcess" suppressJoinFailure="yes" .../>
   <sequence>
      <receive createInstance="yes"
        operation="processCall"
        partnerLink="ProcessCallerPL"
        portType="pld:ProcessCallerPT"
      variable="processStartMessage" />
                    ...
      <flow>
         <activity aid="Sql">
            <installSql />
         </activity>
         <activity aid="C_Sql">
            <installTomcat />
         </activity>
                       ...
         <activity aid="Tomcat">
            <installTomcat />
         </activity>
         <activity aid="PetStore">
            <installPetStore />
         </activity>
                       ...
      </flow>
      <reply operation="processCall"
            partnerLink="ProcessCallerPL"
         portType="pld:ProcessCallerPT"
            variable="processFinishMessage" />
   </sequence>
   <stateRelationships>
      <happenBefore>
         <begin aid="Sql" state="finish" />
         <end aid="C_Sql" state="start" />
      </happenBefore>
      <happenBefore>
         <begin aid="Tomcat" state="finish" />
         <end aid="PetStore" state="start" />
      </happenBefore>
                    ...
      <happenTogether>
         <end aid="Tomcat" state="finish" />
         <end aid="Sql" state="finish" />
      </happenTogether>
   </stateRelationships>
</process>
```

**Figure 6. Code snippet for DSCL+ specification**



```
<process>
  <partnerLinks>
    <partnerLink myRole="Queryee" name="Javafinish_TomcatstartWait"
    partnerLinkType="psd:MQQueryPLT" partnerRole="Queryer" .../>
  </partnerLinks>
  <correlationSets>
    <correlationSet name="Javafinish_TomcatstartCS"
    properties="psd:queryCorrelationData" />
                ...
  </correlationSets>
  <variables>
    <variable messageType="mq:registerRequest"
    name="Javafinish_TomcatstartSendMsg" />
                ...
  </variables>
  <sequence>
    <receive createInstance="yes" operation="processCall"
      partnerLink="ProcessCallerPL" portType="pld:ProcessCallerPT"
      variable="processStartMessage" />
    <flow>
      <flow>
        <sequence>
          <installJava />
                      ...
          <invoke inputVariable="Java_JavafinishSendMsg"
              operation="register"
              partnerLink="Java_JavafinishSend"
              portType="mq:MsgQueueWS" />
        </sequence>
        <sequence>
                      ...
          <invoke inputVariable="JavafinishRevQueryMsg"
                    operation="query" partnerLink="JavafinishWait"
                    portType="mq:MsgQueueWS">
            <correlationSets>
              <correlationSet initiate="yes" pattern="out"
                    set="JavafinishCS"/>
            </correlationSets>
          </invoke>
          <receive operation="queryResponse"
                    partnerLink="JavafinishWait"
                    portType="psd:QueryCallBackPT"
                    variable="JavafinishRevRespMsg">
            <correlationSets>
              <correlationSet set="JavafinishCS"/>
            </correlationSets>
          </receive>
        </sequence>
      </flow>
                ...
    </flow>
    <reply operation="processCall" partnerLink="ProcessCallerPL"
          portType="pld:ProcessCallerPT"
          variable="processFinishMessage" />
  </sequence>
</process>
```

**Figure 7. Snippet code after code-weaving.**

## 4.2. BPEL Code Weaving

From the previous discussion, we know that at each state an activity waits for token messages carrying the status of state it depends on. After all required messages have been received, the activity can transit to its next state. It announces this event to dependent states via token messages. This is a typical message exchange scenario and can be supported by the built-in facilities of BPEL. In particular, we use the following tags.

- `<invoke>`: invokes a operation on a web service
- `<receive>`: specifies message it expects to receive in synchronous mode.
- `<sequence>`: provides sequential execution for all nested subprocesses.
- `<flow>`:provides concurrency and synchronization. It exits when all the activities in the flow have completed. In the translation, each activity is wrapped in a `<flow>`. Each state is a subprocess in the `<flow>`. The execution order is synchronized by the token messages.

Figure 7 is the woven result of Figure 6.

## 5. Evaluation

### 5.1. Developing Effort Evaluation

Similar to the programming language community that typically compares programming languages in terms of lines of code, ease of use, etc. [16], we introduce two metrics to measure the developing effort of a process designer. One is the number of *Lines of Code* (LoC) that measures number of lines of code that he need to write to express the synchronization constraints of a process. The second metric is the number of *Places to Visit* (PtV) that measures the number of places a process designer has to jump back and forth to specify the synchronization constraints.

For LoC, we consider three situations: effort in specifying original process specification, effort in modifying the original specification, and effort in implementing synchronization control decentralization. In BPEL, each structured construct, like `<sequence>` and `<flow>`, counts as 1 LoC and each unstructured construct, like `<link>`, counts as 2 LoC because they requires extra code to declare. In DSCL+, each synchronization statement counts as 1 LoC. The result for our PetStore example is shown in Table 2.

**Table 2. Number of LoC in PetStore process**

| | BPEL | Centralized DSCL+ | Decentralized DSCL+ |
|---|---|---|---|
| **Specification** | 10 | 16 | 16 |
| **Spec. Adaptation** | 7 | 1 | 1 |
| **Sync. Control Decentralization** | N/A | N/A | 16 |

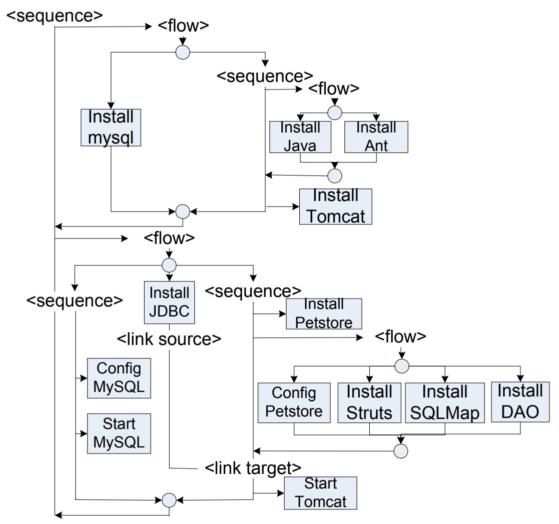


**Figure 8. PetStore Process in BPEL**

**Original process specification.** BPEL requires 8 LoC (four `<flow>/` four `<sequence>;` one `<link>`), shown in Figure 8, while DSCL+ requires 16 LoC for each relationship statement, shown in Figure 2. DSCL+ requires more code than BPEL because DSCL+ uses finer-grained primitives than structured constructs. For example, to specify the parallel between $\boldsymbol{a}_{\text{java}}$ and $\boldsymbol{a}_{\text{ant}}$ we need only one `<flow>` in BPEL, while we need four relationship statements in DSCL+. There is a tradeoff between the flexibility of DSCL and the ease of expression from the high-level construct of BPEL. Language choice depends on the requirements of particular processes. In our opinion, when describing a process without much concurrency, BPEL is preferred. Otherwise, DSCL+ is better.

**Process adaptation.** The advantage of DSCL+ becomes obvious during adaptation. Imagine that the restriction "*the middleware installation subprocess should finish before the application installation subprocess*" has been removed. In BPEL, we must remove the `<flow>` for middleware installation subprocess, insert $\boldsymbol{a}_{\text{sql}}$ into the database subprocess, and insert $\boldsymbol{a}_{\text{java}}$, $\boldsymbol{a}_{\text{ant,}}$ and $\boldsymbol{a}_{\text{tomcat}}$ into the application server subprocess. To manage the constraint between $\boldsymbol{a}_{\text{tomcat}}$, $\boldsymbol{a}_{\text{petstore}}$, and $\boldsymbol{a}_{\text{jdbc}}$, we need one <sequence> to execute $\boldsymbol{a}_{\text{tomcat}}$ first and one `<flow>` for the parallel execution between $\boldsymbol{a}_{\text{petstore}}$ and $\boldsymbol{a}_{\text{jdbc}}$. That is 7 LoC in total. In DSCL+, we simply remove one statement: $\boldsymbol{F}_{\text{mysql}} \leftrightarrow \boldsymbol{F}_{\text{tomcat}}$. That is 1 LoC.

**Synchronization control decentralization.** BPEL cannot coordinate activities nested in distributed processes because there is no easy way to universally identify an intermediate activity in a process and to specify its relationship to other activities. Developers may split the BPEL specification for decentralization according to the approach in [6], but that would result in the creation of a subprocess for each intermediate activity to be synchronized. Furthermore, it could not handle the synchronization constraint for exclusive execution. By comparison, in the decentralized DSCL+, each activity is uniquely tagged. The DSCWeaver inserts synchronization code pertinent to each activity in a subprocess. The designer only needs to specify the constraint for each subprocess. Therefore the LoC remains the same as the centralized version.

Now let's look at the metric of PtV. One PtV counts each time a designer has to jump to a different place in a process specification when implementing a task. The result is shown in Table 3.

**Table 3. Number of PtV in PetStore process**

| | BPEL | Centralized DSCL+ | Decentralized DSCL+ |
|---|---|---|---|
| **Specification** | 19 | 1 | 2 |
| **Spec. Adaptation** | 7 | 1 | 1 |
| **Sync. Control Decentralization** | N/A | N/A | 2 |

For the decentralized DSCL+, all synchronization constraints can be specified at one place. Therefore it is only 1 PtV for both the process design stage and the adaptation stage. The designer can totally rely on DSCWeaver to weave the synchronization code into the base code. But he has to edit each subprocess specification in the decentralized scenario. Therefore it requires 2 PtV for the decentralized deployment of PetStore. By comparison, the developing effort in BPEL increases a lot in PtV metric. Take the effort in specifying original specification for instance. Each structured constructs requires two places to visit. Each unstructured construct requires three places to visit. Since we have eight structure constructs and one unstructured construct, the total is 19 PtV. The reduction from 19 PtV to 1 PtV represents a significant reduction in developer effort. Similar results can be observed during adaptation.

## 5.2. Performance

There are two purposes for the performance experiment. One is to evaluate the overhead brought by the DSCL+ for token messages exchange. We deploy and compare the PetStore process by using the standard BPEL constructs and the instrumented code generated by the DSCWeaver. The other is to demonstrate that DSCL+ can describe synchronization constraints for distributed processes and coordinate them in decentralized manner. We also generated the decentralized DSCL code and measure its performance in term of deployment time.

**Experiment setup.** Our experiment uses a cluster of Intel Pentium machines (2x 3Ghz Pentium 4, 1GB memory) in Redhat Linux 9. The BPEL engine is ActiveBPEL [23]. The web service engine is Axis [24]. We call the machine that starts the process the deployment machine and the machine that hosts the PetStore application the target machine. In the centralized setup of Figure 1a, the deployment machine is installed with ActiveBPEL engine for orchestrating process and Axis for providing the PersistentQueue web service. Two client machines for database and application are configured with Axis as the web service engine. In the decentralized setup of Figure 1b, three of the machines were set up with ActiveBPEL and Axis engines. The role of the deployment machine is to coordinate with two processes running on two client machines. The ActiveBPEL engine on the client machines interacts with local Axis to execute those activities that perform the tasks on the local host. Table 4 is the deployment time.

**Table 4: Deployment time for PetStore**

| | ActiveBPEL | Centralized DSCL+ | Decentralized DSCL+ |
|---|---|---|---|
| **Time (s)** | 75 | 83 | 75 |

We can see that centralized DSCL requires more time than BPEL, but the decentralized DSCL is as good as the ActiveBPEL.

## 6. Related Work

Our work complements other projects exploring the use of AOP to improve the flexibility and adaptability of workflow processes. Bachmendo and Unland describe an approach to use aspects for dynamic workflow evolution by changing behavior of structured constructs like Sequence and And-split at run time [3]. The AO4BPEL project [7] uses aspect for service composition and collects auditing information at runtime. Both of these projects work on the run time behavior of composite services. On the other hand, our work targets synchronization aspects during the *design* stage.

Translation from Petri net to code can be seen in early work [8][2]. Grid-Flow [8] provides a Petri net based user interface for workflow modeling in grid and automatically translates a net to the Grid Flow Description Language. Instead of targeting a particular domain specific language, our approach of synchronization by exchanging token messages can easily accommodate different workflow languages because it simply requires support of a messaging mechanism from the host language. Aalst introduced another tool translating Petri net to BPEL [2]. Instead of establishing a mapping between a Petri net and the structured constructs in BPEL, we translate the Petri net to token messages. This enables us to handle the exclusive state relation and also facilitates the task of converting BPEL from centralized control to decentralized control, which is a significant advantage.

DSCWeaver is capable of providing synchronization-aspect extensions to general purpose flow languages. In this sense, our work is similar to [12] that models synchronization in temporal logic and integrates it with Java programming.

## 7. Conclusion

In this paper, we discussed a limitation of BPEL in modeling synchronization constraints, due to its procedural style. To address this limitation we presented DSCWeaver, an integrated tool that provides a synchronization-aspect extension to BPEL. It uses DSCL, a synchronization expression language to express synchronization constraints in the form of relationship statements. These statements can then be individually validated and later woven into the base BPEL code to form a complete process specification.

This paper presents two major contributions. First, we demonstrate the practicality and feasibility of providing a synchronization-aspect extension to a "general purpose" flow language by implementing the tool of DSCWeaver and applying it to BPEL. Second, our weaving approach is unique. The synchronization constraints are woven into a set of token messages, the exchange of which forms a synchronization protocol among activities in a process. The token messages can be easily integrated with different flow languages and easily exchanged among distributed processes and web services.

## 8. Reference

[1] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow Patterns," Distributed and Parallel Databases, 14(3), pages 5-51, July 2003.

[2] W. M. P. v. d. Aalst and K. B. Lassen. Translating Workflow Nets to BPEL4WS. BPM Center Report BPM-05-16, BPMcenter.org. 2005.

[3] B. Bachmendo, R. Unland. Aspect-Based Workflow Evolution, Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster, UK, August 23, 2001

[4] B. Benatallah, Q. Z. Sheng, et al. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. Proceedings of the 18th International Conference on Data Engineering (ICDE'02), IEEE Computer Society.2002

[5] R. H. Campbell, A. N. Habermann: "The Specification of Process Synchronization by Path Expressions." Lecture Notes in Computer Science 16, Springer-Verlag, Berlin, 1974, pp 89 – 102.

[6] G. Chafle, S. Chandra, et al. Decentralized Orchestration of Composite Web Services. Proceedings of the 13th International World Wide Web Conference, NY, USA, ACM Press, 2004.

[7] A. Charfi and M. Mezini. Aspect-Oriented Web Service Composition with AO4BPEL In Proc. of the European Conference on Web Services ECOWS 2004, LNCS 3250. Erfurt, Germany, September 2004.

[8] Z. Guan, F. Hernandez, et al. Grid-Flow: A Grid-Enabled Scientific Workflow System with a Petri Net-Based Interface. the Grid Workflow Special Issue of Concurrency and Computation: Practice and Experience. 2005.

[9] J. Gray, T. Bapty, et al. Handling Crosscutting Constraints in Domain-Specific Modeling. Communications of the ACM: pp. 87-93. October 2001

[10] K. Jensen. Coloured Petri Nets. Vol 1: Basic Concepts, Springer-Verlag 1992.

[11] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. Proceedings of the 27th international conference on Software engineering, St. Louis, MO, USA, ACM Press.2005

[12] Giuseppe Milicia, Vladimiro Sassone: Jeeg: Temporal Constraints for the Synchronization of Concurrent Objects. Concurrency - Practice and Experience 17(5-6): pp 539-572 2005

[13] T. Murata. Petri Nets: Properties, analysis and applications. Proc. of the IEEE, 77(4):541– 580, 1989.

[14] S. Narayanan and S. A. McIlraith. Simulation, Verification and Automated Composition of Web Services. Proceedings of the 11th international conference on World Wide Web, Honolulu, Hawaii, USA, ACM Press. 2002.

[15] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. Formal Semantics and Analysis of Control Flow in WS-BPEL. BPM Center Report BPM-05-13, BPMcenter.org, 2005.

[16] L. Prechelt. An Empirical Comparison of Seven Programming Languages. IEEE Computer, vol 33, no 10, October 2000, pp 23-29.

[17] Calton Pu and Galen S. Swint. DSL Weaving for Distributed Information Flow Systems. (Invited Keynote.) Proceedings of the 2005 Asia Pacific Web Conference. (APWeb05). Springer-Verlag LNCS. March 29 - April 1, 2005. Shanghai, China.

[18] A.V. Ratzer, L. Wells et.al. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In W. v.d. Aaalst and E. Best, (Eds.) Application and Theory of Petri Nets 2003.

[19] K. Salomaa and S. Yu. Synchronization Expressions and Languages. Journal of Universal Computer Science Vol. 5: 610-621. 1999.

[20] R. Schmidt, U. Assmann. Extending Aspect-Oriented-Programming In Order To Flexibly Support Workflows. In: Proceedings of Aspect-Oriented Programming Workshop at ICSE'98, ed. Lopes, C., Murphy, G., Kiczales G.

[21] Q. Wu, C. Pu, et al. DAG Synchronization Constraint Language for Business Processes. Technical Report. http://www.cc.gatech.edu/~qxw/academic/pub/DSCLReport.pdf, 2005.

[22] Business Process Execution Language for Web Services (BPEL), Version 1.1. http://www.ibm.com/developerworks/library/ws-bpel.

[23] ActiveBPEL. http://www.activebpel.org/

[24] Axis. http://ws.apache.org/axis/

[25] Workflow Management Coalition: Workflow Process Definition Interface – XML Process Definition Language. Document Number WFMC-TC-1025, October 25, 2002.