Categorization and Optimization of Synchronization Dependencies in Business Processes

Qinyi Wu[1], Calton Pu[1], Akhil Sahai[2], Roger Barga[3]

[1]College of Computing, Gatech
{qxw, calton}@cc.gatech.edu

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

[3]Microsoft Research
barga@microsoft.com

Abstract

The current approach for modeling synchronization in business processes relies on sequencing constructs, such as sequence, parallel etc. However, sequencing constructs obfuscate the true source of dependencies in a business process. Moreover, because of the nested structure and scattered code that results from using sequencing constructs, it is hard to add or delete additional constraints without over-specifying necessary constraints or invalidating existing ones.

We propose a dataflow programming approach in which dependencies are explicitly modeled to guide activity scheduling. We first give a systematic categorization of dependencies: data, control, service and cooperation. Each dimension models dependency from its own point of view. Then we show that dependencies of various kinds can be first merged and then optimized to generate a minimal dependency set, which guarantees high concurrency and minimal maintenance cost for process execution.

1. Introduction

A business process is often created to integrate distributed services, and typically contains a group of activities used to interact with remote services, or perform some computation task within the process. In order to achieve intended scenarios, the execution of the activities must be synchronized according to various sequencing constraints. For example, if two activities exchange data, they create a happen-before constraint between the data producer and the data consumer. The leading process modeling languages use sequencing constructs (sequence, parallel etc.) to specify synchronization constraints. Sequencing constructs [1] are highly imperative, and often use a counter to determine the next activity or subprocess for execution. While effective at depicting the structure of a process, they obfuscate the sources of dependencies. For example, a sequence construct may be created due to a data dependency or other business requirement. Furthermore, programming using sequence constructs normally produce nested structures and scattered code, especially under the existence of concurrency [21]. As a result, there is no easy way to add or delete a constraint in a process without over-specifying necessary constraints or invalidating existing ones.

In this paper, we describe a synchronization modeling approach for business processes in which dependencies are first-class citizens, explicitly modeled to guide activity scheduling. This depen*dency-equal-to-scheduling* style is the core feature of dataflow programming, which has the well-known advantage of using dependencies to locate potential parallelism to improve performance [14]. We assume that the dependency information can be extracted from design products like activity diagrams in UML [17], Program Dependency Graph [5] or web service description requirements in WSCL [25] etc. Otherwise, they are provided by domain experts, such as high-level business requirements for exception handling. Classical examples of dependencies include data dependency and control dependency [2], which we extend with additional dimensions, *service* and *cooperation*, to accommodate the highly interactive and complex characteristics of business processes. To derive an efficient global synchronization scheme using dependencies of various kinds, these dependencies are first represented uniformly as synchronization constraints in DSCL [21]. The constraints are then merged and optimized. We have developed a tool – DSCWeaver that accepts as input a business process and its associated synchronization constraints coded in DSCL. The DSCWeaver automatically translates DSCL code into Petri Nets [13] for validation and finally generates BPEL code for real process deployment and execution. Thus, our approach provides a vertical solution for business process specification, optimization, validation and execution. For validation and execution, please refer to our early work [22]. This paper focuses on specification and optimization, which are summarized below:

We give a systematic categorization of dependencies in business processes: *data*, *control*, *service* and *cooperation*. Each dimension models synchronization

1-4244-0803-2/07/$20.00 ©2007 IEEE.

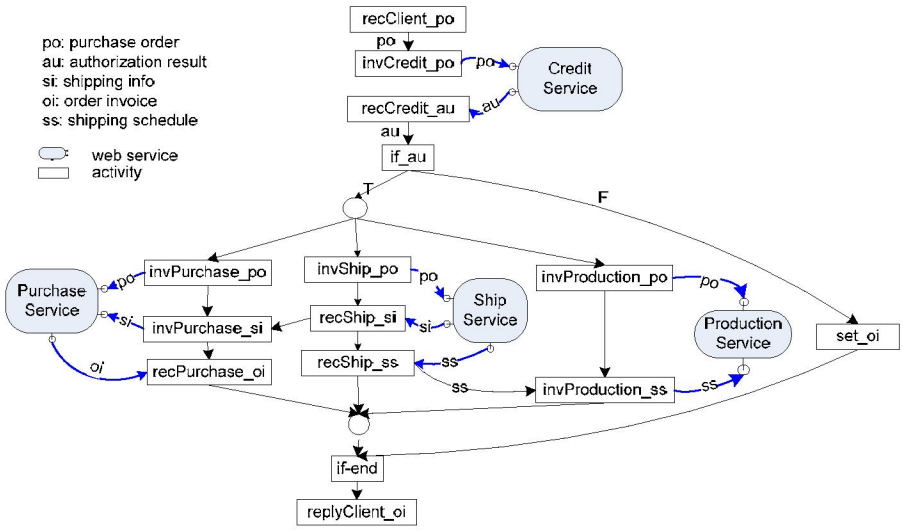

Figure 1. The Purchasing process flowchart

constraints from its own point of view. We then demonstrate that dependencies of different dimensions can be automatically merged together to form a global synchronization scheme. This feature is particularly attractive in automatic service composition, in that participants of service integration can simply submit their dependencies like a WSCL document to a scheduling engine. The scheduling engine will then combine dependencies from all services to infer a global synchronization scheme. For instance, a state-awareness service may require a sequential invocation on its two ports. Instead of passively relying on the correct implementation of a process, the service can now submit it as a service dependency, which tells the scheduling engine to schedule sequentially the corresponding invocation activities in the process. By comparison, there is no easy way to add this constraint if a process is coded in sequencing constructs for the reasons mentioned earlier. As observed in [5], this is very important to relieve programmers from tedious work of managing dependencies involving concurrency and synchronization issues.

We analyze the interactions between dependencies from different dimensions, and extract a minimal set containing only the essential dependencies to be preserved for the correct execution of a process. Sometimes dependencies can impose the same set of synchronization constraints on activity scheduling. In the previous example, the service dependency imposes a sequential synchronization constraint on the invocation activities. If data are exchanged between them, the corresponding data dependency will introduce another sequential constraint. The goal of dependency optimization is to find a minimal dependency set that only keeps the necessary ones. The removal of redundant dependencies results in a lightweight implementation, enabling higher flexibility for adaptation and opportunities for concurrent execution.

The rest of the paper is organized as follows. In Section 2, we describe the Purchasing process, a running example throughout this paper. Section 3 categorizes different types of dependencies. The dependency optimization is explained in Section 4. We review related work in Section 5 and conclude in Section 6.

2. Motivating Example

Let's consider the purchasing business process illustrated in Figure 1. We borrowed this example from BPEL 1.0 specification [24] and extended it with a conditional branch to illustrate a more interesting scenario.

The purchasing process interacts with four web services: *Credit* Service, *Purchase* Service, *Ship* Service and *Production* Service. The process consists of activities, each represented in the format of *actionService_parameter* if it is related to a remote service interaction or *action_parameter* if it is related to local computation. For example, the activity of invoking *Credit* Service with parameter *po* is represented by *invCredit_po*. Now let's describe the process scenario. After receiving a purchase order from a client (*recClient_po*), the process sends it to *Credit* Service for credit card authorization (*invCredit_po*). If successful, three subprocesses are instantiated concurrently: *PurchaseSubprocess* (*invPurchase_po*, *invPurchase_si*, and *recPurchase_oi*), *ShipSubprocess* (*invShip_po*, *recShip_si*, and *recShip_ss*), and *ProductionSubprocess* (*invProduction_po*, and *invProduction_ss*). If it fails, the

order invoice is set with failure information (*set_oi*). As the last step, the order invoice will be sent back to the client (*replyClient_oi*) after the execution of *ShipSubprocess* and *ProductionSubprocess* finishes. The execution of these three subprocesses is not independent. They synchronize at intermediate steps, particularly the synchronization between *recShip_si* and *invPurchase_si*, and between *recShip_ss* and *invProduction_ss* due to data dependencies.

- *PurchaseSubprocess* sends the purchasing order (*invPurchase_po*) and shipping invoice (*invPurchase_si*) to *Purchase* service that calculates the final invoice and sends it back to the process (*recPurchase_oi*). *Purchase* service is state-aware. It requires a sequential invocation at its two ports so that it does not receive a shipping invoice without receiving the corresponding purchase order information.
- *ShipSubprocess* sends the purchasing order (*invShip_po*) to *Ship* service that computes and sends back shipping invoice (*recShip_si*) and shipping scheduling (*recShip_ss*) back to the process for further processing.
- *ProductionSubprocess* sends the purchasing order (*invProduction_po*) and shipping information (*invProduction_ss*) to *Production* service for corresponding product processing.

This purchasing process example, though simple, exemplifies a typical scenario in business processes. We will use this as a running example to demonstrate different types of dependencies and the procedure of dependency optimization. Without loss of generality, we assume that all service interactions are asynchronous because a synchronous invocation can always be translated to a pair of send and receive asynchronous calls.

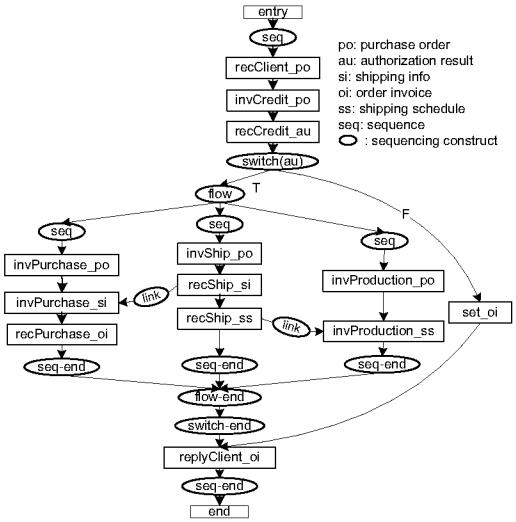

Figure 2. The Purchasing process implemented in sequencing constructs

To better motivate our work, we show a sequencing-construct implementation based on [24] in Figure 2. The constructs are chosen from BPEL. There are several limitations associated with this approach. First, programming in this manner requires that process designers are aware of all dependencies, both local and remote, and they must choose the proper constructs for coding them. The choice becomes delicate if a construct must accommodate the combination of a variety of dependencies. Second, it is not obvious to tell the source of synchronization by directly looking at the implementation, which further complicates the task of maintenance and adaptation, especially for evolving processes. For example, people may question why there is a sequential synchronization between *invPurchase_po* and *invPurchase_si*. This is actually due to an invocation constraint imposed by the remote *Purchase* service. Third, sequencing constructs may under- or over- specify synchronization constraints, which is hard to detect without dependency information. For instance, the sequencing between *invProduction_po* and *invProduction_ss* is an over-specified dependency because they do not exchange data and there are no other constraints associated with them. By comparison, the sequencing between *invPurchase_po* and *invPurchase_si* is required because of the remote service invocation constraint from *Purchase* service, even though they do not change data either. Finally, manual analysis is highly error-prone. It is better to explicitly model dependencies and rely on automatic dependency inference to generate a synchronization scheme.

3. Dependency Categorization

In general, data dependency and control dependency are dimensions solely for program analysis, as in compiler theory [2] and workflow modeling [15]. However, we argue that the complexity and highly interactive nature of business processes render them inadequate. For differentiation, we refer to traditional programs as computation-centric programs, and business process programs as interaction-centric programs. There are two main reasons for this inadequacy. First, compared to computation-centric programs that have only one thread of execution, there may be multiple execution threads in an interaction-centric program. One thread is within the main process, while the others are within remote services. These execution threads interact with each other through either synchronous or asynchronous calls that need to be synchronized properly. Second, when programming a business process, each service is treated as a black box. A programmer can normally assume that he only needs to care about the input and the output of a service. However, there are cases that the execution of a service has a side effect on other invocations. As a result, its corresponding

invocation activities need to be synchronized. This implicit dependency can not be captured by either data or control dependencies, and as a result it demands special treatment.

In this section, we start with the explanation of *data* and *control* dependencies. Then we introduce *service* and *cooperation* dependencies along with an explanation of why *data* and *control* dependencies alone are inadequate to model a business process. Among these four dimensions, *data*, *control* and *cooperation* dependencies are used to model synchronization constraints within the process. *Service* dependency is used to model synchronization constraints between a process and remote services, and within remote services.

3.1. Data Dependency and Control Dependency

Data dependency arises between a data producer activity and a data consumer activity. For example, in Figure 3, data *y* creates a definition-use data dependency between $\boldsymbol{a}_2$ and $\boldsymbol{a}_3$, represented by a dotted line in Figure 4. In a general programming language, the handling of data dependencies in a program is complicated due to pointers, procedural calls, and name dependencies [2]. By comparison, the data dependencies in process programming language are relatively simple for several reasons. First, the parameter reference in the invocation of remote service is call-by-value. Second, the execution of remote service is always local. There is no side effect on the process state. Thus, the definition-use type of data dependencies is dominant in activity scheduling.

Control dependency changes the sequential execution of the process by condition and iteration. For example, in Figure 3, the execution path after $\boldsymbol{a}_1$ is decided by the value of flag, which creates control dependencies on all the activities $\boldsymbol{a}_2$, $\boldsymbol{a}_3$, $\boldsymbol{a}_4$, $\boldsymbol{a}_5$ and $\boldsymbol{a}_6$ located along its descendant branches, represented by solid lines in Figure 4. However, a_7 dominates all the paths from $\boldsymbol{a}_1$ to ***stop***. It is not control dependent on $\boldsymbol{a}_1$ any more. The edges are annotated with control condition: "*T*", "*F*", or "*NONE*".

Normally, data and control dependencies can be automatically extracted from document products at the end of the design stage. For example, in the dataflow programming approach [14], dependency information can be directly extracted from the dataflow diagram. In the imperative programming approach, a process is implemented in procedural code, such as sequencing constructs. In this case, we can use program analysis techniques like Program Dependency Graph (PDG) to extract dependency information [5]. Or in meta-modeling approach like UML, dependency information is available in activity diagrams, use case diagrams etc.

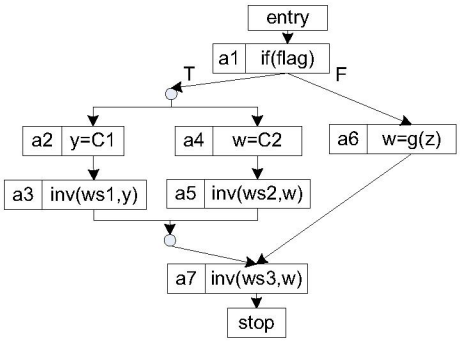

Figure 3. A process specification

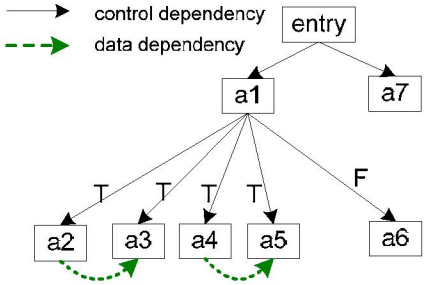

Figure 4. Data and control dependency graph

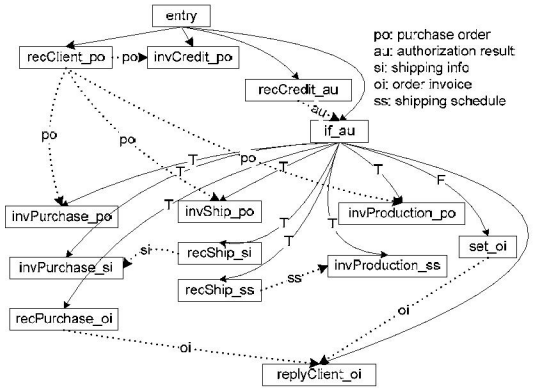

Figure 5. Data and control dependency graph for the Purchasing process

Coming back to the Purchasing process, we can represent its data and control dependencies coherently as shown in Figure 5. An example of data dependency is between *recClient_po* and *invCredit_po* in that the purchase order, *po*, is transferred between them. An example of control dependency is between *if_au* and *invPurchase_po* because the execution of *invPurchase_po* is controlled by *if_au*. By examining Figure 5, we found that the current dependencies do not satisfy all the requirement of the original specification. For example, the

process requires that the *ShipSubprocess* and *ProductionSubprocess* should be finished before sending back the invoice to the client. In the current synchronization scheme, *recPurchase_oi* does not synchronize with these two subprocesses and will send the invoice back to the client as soon as the invoice data becomes available. Also, even though there is a sequential constraint on the invocation of *Purchase* service, there is no enforcement of sequential ordering between *invPurchase_po* and *invPurchase_si*. Therefore, we need look at other two types of dependencies to capture the missing information.

3.2. Service Dependency and Cooperation Dependency

Service dependency defines those interactions between a process and a remote service, or within a remote service. These external interactions put extra constraints on activity scheduling. For example, after an activity makes an asynchronous invocation to a remote service, a receive activity should be scheduled to listen for the arrival of the asynchronous reply. Otherwise, the callback of the remote service may receive an exception of invalid URL. For another example, the invocation of a remote service requires two subsequent calls at its $port_1$ and $port_2$ respectively. The corresponding invocation activities in a process should be scheduled sequentially. In the sequencing-construct style, these service dependencies are implicitly implemented by constructs. In our approach, they are explicitly represented and used to infer execution order of activities. Service dependency information is likely to be found in standard description documents like WSCL that specifies the XML documents being exchanged, and the allowed sequencing of these document exchanges [25].

Cooperation dependency defines synchronization constraints introduced by applications that are not captured by either *data*, *control* or *service* dependencies. Typically, these constraints describe the cooperation of activities to achieve certain business goals or to support some implicit interactions among activities. An example of this type in the Purchasing process is the dependency between *invProduction_ss* and *replyClient_oi*. It requires that the invoice only be sent back to the client after the execution of *invProduction_ss*. If an exception occurs at *invProduction_ss*, the execution of *replyclient_oi* is postponed until the exception is fixed. From a business point of view, it is desirable to preserve this dependency so that a customer who receives an invoice is guaranteed to receive her product. Since there is neither a data nor a control dependency between these activities, this synchronization constraint must be described by a cooperation dependency. Cooperation dependency can also be used to capture implicit dependencies. Let's look at in Figure 6, a deployment process that will install middleware and application software packages after receiving a deployment configuration (*recClient_Config*). *Deploy* service accepts package installation configuration as input, and then installs the corresponding package. *invDeploy_midConfig* activity extracts middleware configuration and send it *Deploy* service. *invDeploy_appConfig* activity extracts application configuration and sends it to *Deploy* service. Even though there is neither a data nor control dependency between *invDeploy_midConfig* and *invDeploy_appConfig*, *invDeploy_appConfig* must be executed after *invDeploy_midConfig* because *invDeploy_midConfig* will set up certain directory structure for the installation of the application package. As a real life example, an application servlet should be installed under the application server Tomcat's `$Tomcat/webapp` directory. Therefore, there is a *happen-before* synchronization constraint between *invDeploy_midConfig* and *invDeploy_appConfig*. Sometimes cooperation dependency may depend on the state of activity. For example, a business dependency may require that the activity of collecting customer satisfaction survey *collectSurvey* should be started before the activity of closing a purchase order *closeOrder* finishes. In this case, the life spans of two activities overlap with each other. The existence of fine-granularity synchronization put more challenge on synchronization modeling.

Normally cooperation dependencies can not be found or directly inferred from process logic description documents, such as flowcharts or activity diagram. The reason is that they are superimposed over data and control dependencies to achieve more stringent high-level business requirements or prevent exceptions caused by implicit interactions. Therefore, it is likely that they are provided by either process analysts or domain experts.

The existence of both *service* and *cooperation* dependencies highlights a distinctive feature of business process from a computation-centric program that is purely analyzed from a dataflow and control flow point of view.

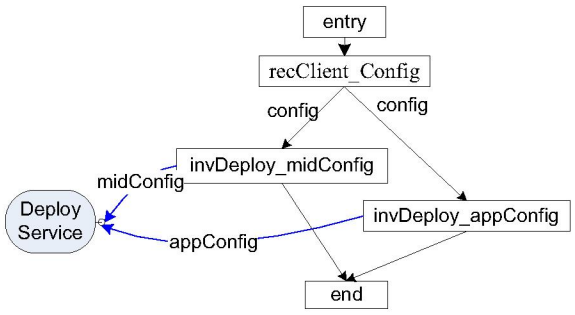

Figure 6. Deployment process

Table 1. The Purchasing process dependencies

type	dependency
data $\{\rightarrow_d\}$	*recClient_po* $\rightarrow_d$ *invCredit_po*, *recCreditt_au* $\rightarrow_d$ *if_au*, *recClient_po* $\rightarrow_d$ *invPurchase_po*, *recClient_po* $\rightarrow_d$ *invShip_po*, *recClient_po* $\rightarrow_d$ *invProduction_po*, *recShip_si* $\rightarrow_d$ *invPurchase_si*, *recShip_ss* $\rightarrow_d$ *invProduction_ss*, *set_oi* $\rightarrow_d$ *replyClient_oi* *recPurchase_oi* $\rightarrow_d$ *replyClient_oi*,
control $\{\rightarrow_l\}$	*if_au* $\rightarrow_T$ *invPurchase_po*, *if_au* $\rightarrow_T$ *invPurchas_si*, *if_au* $\rightarrow_T$ *recPurchase_oi*, *if_au* $\rightarrow_T$ *invShip_po*, *if_au* $\rightarrow_T$ *recShip_si*, *if_au* $\rightarrow_T$ *recShip_ss*, *if_au* $\rightarrow_T$ *invProduction_po*, *if_au* $\rightarrow_T$ *invProduction_ss* *if_au* $\rightarrow_F$ *set_oi* *if_au* $\rightarrow$ *replyClient_oi*
cooperative $\{\rightarrow_o\}$	*recPurchase_oi* $\rightarrow_o$ *replyClient_oi*, *invShip_po* $\rightarrow_o$ *replyClient_oi*, *recShip_si* $\rightarrow_o$ *replyClient_oi*, *recShip_ss* $\rightarrow_o$ *replyClient_oi*, *invProduction_po* $\rightarrow_o$ *replyClient_oi* *invProduction_ss* $\rightarrow_o$ *replyClient_oi*
service $\{\rightarrow_s\}$	*invCredit_po* $\rightarrow_s$ *Credit*, *Credit* $\rightarrow_s$ *Credit*$_d$, *Credit*$_d$ $\rightarrow_s$ *recCredit_au*, *invPurchase_po* $\rightarrow_s$ *Purchase*$_1$, *invPurchas_si* $\rightarrow_s$ *Purchase*$_2$, *Purchase*$_d$ $\rightarrow_s$ *recPurchase_oi*, *Purchase*$_1$ $\rightarrow_s$ *Purchase*$_d$, *Purchase*$_2$ $\rightarrow_s$ *Purchase*$_d$, *Purchase*$_1$ $\rightarrow_s$ *Purchase*$_2$, *invShip_po* $\rightarrow_s$ *Ship*, *Ship* $\rightarrow_s$ *Ship*$_d$, *Ship*$_d$ $\rightarrow_s$ *recShip_si*, *Ship*$_d$ $\rightarrow_s$ *recShip_ss*, *invProduction_po* $\rightarrow_s$ *Production*$_1$, *invProduction_ss* $\rightarrow_s$ *Production*$_2$

3.3. Dependency Analysis on the Purchasing Process

After introducing four types of dependencies, we can now systematically examine the dependencies in the Purchasing process. For ease of discussion, we use $\rightarrow_d$, $\rightarrow_l$, $\rightarrow_s$ and $\rightarrow_o$ to represent ***d**ata*, *control*, ***s**ervice* and *c**o**operative* dependency respectively, and $\{\rightarrow_d\}$, $\{\rightarrow_l\}$, $\{\rightarrow_s\}$ and $\{\rightarrow_o\}$ to represent their corresponding sets. For each service, ***s***, if it has more than one port, we name them as s_1, s_2, …, s_n in order. If a service accepts asynchronous call, it will call back to the original invoker through a dummy port, named as s_d. Table 1 represents and categorizes all of the dependencies in the Purchasing process. The data dependencies and control dependencies are obtained from Figure 5. Cooperative dependencies come from the constraint that the invoice should be sent back to the client after both *ShipSubprocess* and *ProductionSubprocess* finish. This constraint is most likely specified by a process analyst. Service dependencies come from the interaction with remote services. They can be obtained from service description documents.

4. Dependency Optimization

Each dependency imposes a sequencing constraint. The task of activity scheduling is to synchronize activities in such a way that all dependencies are maintained. However, some of the dependencies may impose the same set of sequencing constraints. This is the case of *recPurchase_oi* $\rightarrow_o$ *replyClient_oi*. If the activity scheduler is already monitoring the data dependency *recPurchase_oi* $\rightarrow_d$ *replyClient_oi*, it does not need to monitor *recPurchase_oi* $\rightarrow_o$ *replyClient_oi* any more. These redundant constraints incur unnecessary maintenance and computation costs if added to the scheduling engine. In this section, we detail how to systematically remove all redundant dependencies to obtain a minimal dependency set. We first explain DSCL for the representation of synchronization constraints of dependencies in Section 4.1. Then we introduce two pre-processing steps before the step of dependency optimization: DSCL representation of dependencies and service dependency translation in Section 4.2 and Section 4.3 respectively. After that, we formally describe the procedure of dependency optimization.

4.1. DAG Synchronization Constraint Language

DAG Synchronization Constraint Language (DSCL) is a synchronization modeling language inspired by research of parallel programming in distributed systems [20]. DSCL treats the life cycle of an activity, ***a***, as a sequence of states, start (*S*), run (*R*), and finish (*F*), and synchronizes an activity with others depending on its current state. It has a declarative syntax and defines three relations to describe the synchronization relationships on activity states. The synchronization scheme described in DSCL can be mapped to Petri Nets for validation and finally translated to a process modeling language for execution [22]. In our early work, it assumes the single output of an activity execution. In order to use DSCL to represent the synchronization constraints of different dimensions, we need to extend it with the value of states

in order to handle the control dependency which has multiple output result. This extension is the same as the extension from basic Petri Nets [13] to Colored Petri Nets [10] that differentiate the type of tokens. The three synchronization relations are described below:

- *HappenBefore* ($\rightarrow_c$): the state at the beginning of the arrow should happen before the state at the end under condition *c*. *c* is omitted if this relation is unconditional.
- *HappenTogether* ($\leftrightarrow_c$): the two states at both ends should be reached together under condition *c*. *c* is omitted if this relation is unconditional.
- *Exclusive* (*O*): states at both ends must not be concurrent.

DSCL can describe a wide variety of synchronization behavior, like *sequence, parallel split, synchronization, interleave parallel routing,* and *milestone* [1]. For example, a data dependency between $\boldsymbol{a}_i$ and $\boldsymbol{a}_j$ can be expressed as a *HappenBefore* relationship between the finish state of data producer, $\boldsymbol{F}$, and the start state of data consumer, $\boldsymbol{S}$, represented as $\boldsymbol{F}_i \rightarrow \boldsymbol{S}_j$. Furthermore, it can also describe constraints when the life cycle of two activities are overlapped with each other. For instance, the cooperation dependency mentioned in Section 3.2 can be described as $\boldsymbol{S}_{\text{collectSurvey}} \rightarrow \boldsymbol{F}_{\text{closeOrder}}$.

There are several reasons that we choose DSCL as the synchronization modeling language. First, a dependency essentially imposes a synchronization constraint on activity scheduling. For example, a data dependency essentially impose a happen-before constraint and can be represented by the *HappenBefore* relationship. Therefore, DSCL can be used as an intermediate language to represent all the dependencies. Second, DSCL expresses synchronization at the granularity of activity state, which may be required for expressing certain cooperation and service dependencies. Finally, the synchronization constraint expressed in DSCL can be validated and mapped to existing process modeling language for real execution. This means that conflict dependencies like infinite synchronization sequence can be detected during design stage, which is important to guarantee the correct execution of a business process at run time.

4.2. DSCL Representation of Dependencies

To derive a complete synchronization scheme out of dependencies of various kinds, we use DSCL as an intermediate language to represent the synchronization constraints inferred from dependencies. The *data*, *service* and *cooperation* dependencies can be represented by unconditional *HappenBefore* relation, represented by $\rightarrow$. The *control* dependencies can be represented by conditional *HappenBefore* relation, represented by $\rightarrow_c$. Here *c* takes value either *T* or *F*. From [21], we know that $\leftrightarrow_c$ is actually a "syntax sugar" and can always be

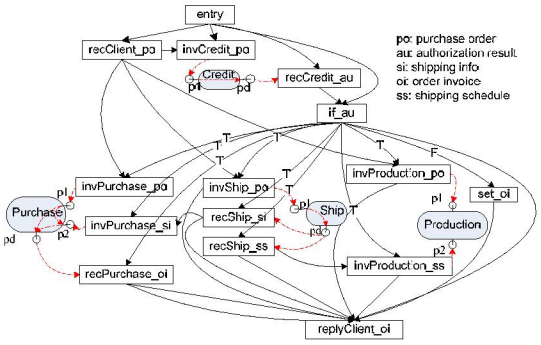

Figure 7. Synchronization Constraints for the Purchasing process

simulated by introducing a coordinating activity and using $\rightarrow_c$ to represent it. For *O*, it is defined to describe transactional cooperation dependency. For example, two concurrent activities access shared data in a backend database. Even though they do not explicitly exchange data within the main process, they must be scheduled in a mutual exclusive way for the purpose of maintaining data consistency. Synchronization constraints described by *O* will be dynamically checked by a scheduling engine at the time of starting an activity, not for statically constructing a synchronization scheme. Therefore, we only discuss $\rightarrow_c$. when explaining synchronization optimization.

We define the synchronization constraint set, *P*, derived out of dependency sets as

$P = \{A \rightarrow B \mid A \rightarrow_d B \in \{\rightarrow_d\} \vee A \rightarrow_o B \in \{\rightarrow_o\} \vee A \rightarrow_s B \in \{\rightarrow_s\}\} \cup \{\rightarrow_1\}$.

Since the *HappenBefore* relation defines a partial order, *P* has transitive property i.e. $A \rightarrow B \wedge B \rightarrow C \Rightarrow A \rightarrow C$. Now we can formally define synchronization constraint set.

Definition 1 (synchronization constraint set)**:** a synchronization constraint set is a triple $SC = \{A, S, P\}$, where

- *A*: denotes all the internal activities
- *S*: denotes all the external services
- *P*: denotes all unconditional ($\rightarrow$) and conditional ($\rightarrow_c$) synchronization constraints

In terms of the Purchasing process, $A = \{$ *recClient_po*, *invPurchase_po*, …, *replyClient_oi*$\}$, $S = \{$*Credit*, $Purchase_1$, $Purchase_2$, …, $Production_2\}$, $P = \{$*recClient_po* $\rightarrow$ *invPurchase_po*, *recClient_po* $\rightarrow$ *invShip_po*, ..., *invProduction_ss* $\rightarrow$ $Production_2$ $\}$. The synchronization constraint set for the Purchasing process is illustrated in Figure 7.

4.3. Service Dependency Translation

Service dependencies control the interaction between a process and its interacting services. Therefore, the implementation of a process must consider these

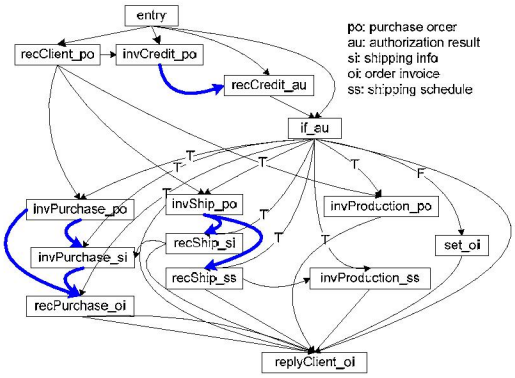

Figure 8. Dependency translation on service dependencies

dependencies in order to correctly cooperate with remote services. As a result, the service dependencies should be translated into those that take effect on corresponding invocation activities. Next, we define transitive path and use it to explain the translation procedure.

Definition 2 (transitive path): Given a synchronization constraint set $SC = \{A, S, P\}$, a transitive path is formed by recursively tracing the *HappenBefore* relationship in P. It contains either internal activity or external services or both.

The idea of service dependency translation is simple. As an example, given the transitive path $\boldsymbol{a}_1 \rightarrow \boldsymbol{a}_2 \rightarrow \boldsymbol{ws}_{1.1} \rightarrow \boldsymbol{ws}_{1.d} \rightarrow \boldsymbol{a}_3 \rightarrow \boldsymbol{a}_4 \ldots$, it is translated to $\boldsymbol{a}_1 \rightarrow \boldsymbol{a}_2 \rightarrow \boldsymbol{a}_3 \rightarrow \boldsymbol{a}_4$. In more detail, for each transitive path that contains external activities, locate its first external activity, in this example being $\boldsymbol{ws}_{1.1}$, and find its closest internal ancestor activity, $\boldsymbol{a}_2$. If its closest internal offspring activity exists, in this example being $\boldsymbol{a}_3$, add a synchronization constraint connecting these two internal activities and remove the external activities in between and their associated dependencies. If its closest internal offspring activity does not exist, simply remove the external activities and their associated dependencies because they will not impact the activity scheduling within the process. Since the translated dependency set only contains internal activities, we call it Activity Synchronization Constraint set, $ASC = \{A, P\}$. A complete translation of the purchasing process is illustrated in Figure 8. The bold edges are translation results. For example, the $Purchase_1 \rightarrow_s Purchase_2$ is translated to *invPurchase_po* → *invPurchase_si.*

4.4. Minimal Dependence Set

A minimal dependency set determines the least sequencing between activities that needs to preserve the correct execution order of a process. In order to formally define the minimal dependency set, we give three preliminary definitions: transitive closure of activity, set cover of two synchronization constraint sets and transitive equivalence of two synchronization constraint sets first.

Definition 3 (transitive closure of activity): Given an $ASD = \{A, P\}$, for each activity a in A, its transitive closure $\boldsymbol{a}^+$ under P contains all the activities that can be reached by following its transitive path.

For example, given $\boldsymbol{a}_1 \rightarrow \boldsymbol{a}_2$ and $\boldsymbol{a}_2 \rightarrow \boldsymbol{a}_3$, $\boldsymbol{a}_1^+ = \{\boldsymbol{a}_2, \boldsymbol{a}_3\}$. If the transitive path contains conditional constraint, the associated activities should be annotated with the conditional value. Notice that the annotation procedure will be repeatedly applied to all the activities following the conditional constraint. For example, if $\boldsymbol{a}_1 \rightarrow \boldsymbol{a}_2 \rightarrow_T \boldsymbol{a}_3 \rightarrow \boldsymbol{a}_4$, $\boldsymbol{a}_1^+ = \{\boldsymbol{a}_2, \boldsymbol{a}_3(T_2), \boldsymbol{a}_4(T_2)\}$. Here both $\boldsymbol{a}_3$ and $\boldsymbol{a}_4$ are conditional on $\boldsymbol{a}_2$.

We say two transitive closures are the same if they contain the same set of activities and their corresponding conditional annotations.

Definition 4 (set cover of two synchronization constraint sets)**:** A set of synchronization constraints P is said to cover another set of synchronization constraints Q if for each activity $\boldsymbol{a} \in Q$, its transitive closure $\boldsymbol{a}^+$ satisfies $\boldsymbol{a}^+$ under $Q \subseteq \boldsymbol{a}^+$ under P.

Definition 5 (transitive equivalence of two synchronization constraint sets): Two sets of synchronization constraints P and Q are transitive equivalent if P covers Q and Q covers P.

Now we formally define the minimal dependency set.

Definition 6 (minimal synchronization constraint set): A minimal synchronization constraint set P^* for a synchronization constraint set P satisfies the following properties:

- P and P^* is transitive equivalent.
- We cannot remove any synchronization constraints from P^* and still have a set that is transitive equivalent to P.

Notice that similar to the minimal set of functional dependencies in database, the minimal set of synchronization constraint is not necessarily unique. Below is the algorithm for obtaining it.

> **Algorithm** Finding a minimal dependency set P^* for a set of partial ordering P:
> *1.* $P^* = P$
> 2. For each partial ordering $\boldsymbol{a}_i \rightarrow \boldsymbol{a}_j$ in P
> If $P^* - \{\boldsymbol{a}_i \rightarrow \boldsymbol{a}_j\}$ is transitive equivalent to P,
> Then $P^* = P^* - \{\boldsymbol{a}_i \rightarrow \boldsymbol{a}_j\}$

The minimal dependency set for the Purchasing process is illustrated in Figure 9.

Table 2 tells us the number of dependencies before and after the dependency inference. There are 23 constraints removed from the original synchronization constraints set in Table 1. This is a significant reduction in terms of number of constraints to be monitored.

Table 2. Number of dependency Comparison

	original constraints	minimized constraints
Purchasing process	40	17

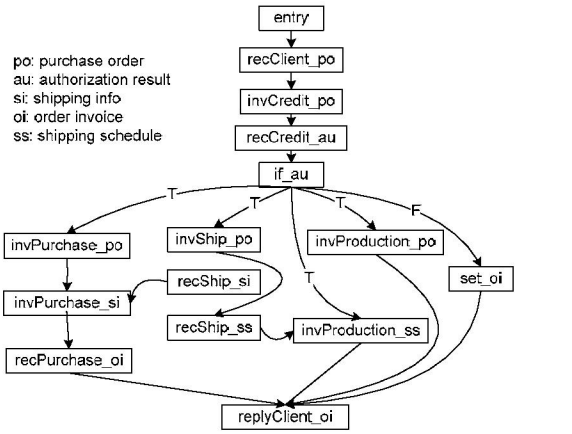

Figure 9. Minimal synchronization constraints

5. Related Work

Much work on program analysis and optimization has been done on computation-centric programs by using techniques from compiler theory [5] [11]. However, little work on these perspectives has been done on interaction-centric programs. In process modeling languages, people more focus on workflow patterns [1] or the interaction of various aspects in a workflow system like implementation, roles, actors [9], not from the view of multiple dimensions of dependencies. One work related to us is [12] that uses PDG to analyze dataflow, control flow and constructs in a process to decentralize execution control with the goal of minimizing communication overhead. We use a different approach in rewriting the constructs and focus on the interaction of multiple dependencies with the goal of removing redundant ones.

There are two paradigms in workflow programming [15]. One is dataflow paradigm, mostly seen in scientific workflows. The other is imperative paradigm, commonly seen in business processes. Our work bears lots of features with dataflow programming in the way of using dependency to determine execution instead of using constructs. The limitation of introducing concurrency constructs into programming languages have been demonstrated in the literature of dataflow programming in that the explicit placement destroys one of the appealing features of implicit parallelism in the dataflow concept [14]. Moreover, our approach can be applied to process implemented in imperative paradigm as well. A process implemented in workflow patterns [1], which essentially follows the imperative programming paradigm, can be parsed to a dependency graph such as PDG and use rewriting rules [22] to translate constructs into synchronization constraints, and then participates in the step of dependency inference and optimization. Therefore, similar to [3] that proposed the way of translating imperative language to data flow graph to establish the connection between these two paradigms, our work can be regarded as an intermediate representation for both paradigms in business process implementation and optimization. Recently we can see an emergence of these two paradigms. Scientific workflow starts to adopt control flow constructs [4]. Business processes also start to using dependencies for scheduling, decentralization [12].

Most of the early process modeling languages has an imperative style and uses control constructs or workflow patterns to specify the skeleton of a process [24]. Until very recently, the necessity of providing a declarative flow language for service scheduling has caught up its pace. [17] proposed an approach of using temporal logic to specify the synchronization constraints between different components. Temporal logic provides a richer syntax for describing and monitoring synchronization dependencies. But its non-determinism makes it impractical for generating a message-exchanging synchronization protocol for synchronization enforcement.

Another related area of research is rule-based service composition [16][18]. Rules are defined to decide role assignment in process execution, message exchange, and flow constraints etc. Most business rules could be recast to dependencies defined in our framework and used as input for service scheduling engine. For example, the structure related rules in [16] could be recast either as control dependencies or data dependencies. These early work focused more on rule classification and process modeling. By comparison, our work identified those dependencies crucial to service scheduling and studied their interaction effect. Therefore, our approach can server as a rule-based scheduling engine and plug into their systems.

A traditional approach to handling dependencies implicitly uses extended transaction models [6][8] that introduce new data manipulation semantics more sophisticated than serializability. Typical methods to implementing extended transaction models, e.g., Reflective Transaction Framework [23], extend algorithms used in database management systems such as concurrency control. In contrast, our research results address the synchronization needs of programs, workflows, and data manipulations in a uniform way.

6. Conclusion

In this paper, we discussed the limitation of sequencing constructs in specifying synchronization constraints in business processes. To address this limitation, we proposed a dataflow programming

approach, in which dependencies are treated as first-class citizens and explicitly modeled to guide activity scheduling. To capture the interactive and complex characteristics of business processes, we extended traditional dependencies, *data* and *control*, with additional dimensions of *service* and *cooperation*. These four dimensions of dependencies provide a systematic framework to describe the synchronization behavior of a process. Furthermore, we show how dependencies from different dimensions can be merged and optimized through an intermediate synchronization constraint modeling language—DSCL. The result is a practical and effective way of specifying and optimizing a business process specification for execution.

7. 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] Alfred V. Aho , Ravi Sethi , Jeffrey D. Ullman, Compilers: principles, techniques, and tools, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1986

[3] M. Beck, R. Johnson and K. Pingali. From control flow to dataflow. J. Parallel Distrib. Comput. 12(2): 118-129. 1991.

[4] S. Bowers, B. Ludascher, A. H.H. Ngu, T. Critchlow. Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow. In Proceedings of IEEE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow 2006)

[5] H. Cervantes and R.S. Hall. Automating Service Dependency Management in a Service-Oriented Component Model. Proceedings of the Sixth Component-Based Software Engineering Workshop. May 2003

[6] A.K. Elmagarmid, editor. Database Transaction Models for Advanced Applications. Morgan Kaufmann, 1992.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3), 1987.

[8] Sushil Jajodia, Larry Kerschberg. Advanced Transaction Models and Architectures. Kluwer 1997

[9] Kwang-Hoon Kim. Workflow dependency analysis and its implications on distributed workflow systems. 17th International Conference on Advanced Information Networking and Applications. p677- 682. 2003. AINA 2003.

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

[11] D. J. Kuck, R. H. Kahn, D. A. Padua, B. Leasure and M. Wolfe. Dependence graphs and compiler optimizations. In 8th Annual ACM Symposium on Principles of Programming

[12] M. G. Nanda, S. Chandra and V. Sarkar. Decentralizing execution of composite web services. Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 170-187. Vancouver, BC, Canada, ACM Press.2004

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

[14] W. M. Johnston, J. R. P. Hanna and R. J. Millar. Advances in dataflow programming languages. ACM Comput. Surv. 36(1): 1-34. 2004

[15] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, Y. Zhao. Scientific Workflow Management and the Kepler System. Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows, to appear, 2005.

[16] B. Orriens, J Yang, and M.P. Papazoglou (2003): A Framework for Business Rule Driven Service Composition. Proceedings of the Fourth International Workshop on Conceptual Modeling Approaches for e-Business Dealing with Business Volatility, Chicago, United States, Oktober 13-16, 2003.

[17] M. Pesic and W.M.P. van der Aalst. DecSerFlow: Towards a Truly Declarative Service Flow Language. In F. Leymann, W. Reisig, S.R. Thatte, and W.M.P. van der Aalst, editors, The Role of Business Processes in Service Oriented Architectures, number 6291 in Dagstuhl Seminar Proceedings. 2006

[18] Steven P. Reiss. Constraining Software Evolution. Proceedings of the International Conference on Software Maintenance (ICSM'02)

[19] James Rumbaugh, Ivar Jacobson, and Grady Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1998.

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

[21] Qinyi Wu, Calton Pu, Akhil Sahai. DAG Synchronization Constraint Language for Business Processes. IEEE Conference on E-Commerce Technology CEC'06

[22] Qinyi Wu, Calton Pu, Akhil Sahai, Roger Barga, Gueyoung Jung, Jason Parekh, Galen Swint. DSCWeaver: Synchronization-Constraint Aspect Extension to Procedural Process Specification Languages. IEEE International Conference on Web Services 2006.

[23] Roger S. Barga and Calton Pu. A Practical and Modular Implementation of Extended Transaction Models. VLDB 1995: 206-217

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

[25] Web Services Conversation Language (WSCL) 1.0. http://www.w3.org/TR/wscl10/