DAG Synchronization Constraint Language for Business Processes

Qinyi Wu, Calton Pu
College of Computing,
Georgia Institute of Technology
{qxw, calton}@cc.gatech.edu

Akhil Sahai
HP Laboratories, Palo Alto, CA
Akhil.sahai@hp.com

Abstract

Correct synchronization among activities is critical in a business process. Current workflow languages such as BPEL specify the control flow of processes explicitly. However, their procedural style may cause inflexibility and tangled code. We propose DSCL (Dag-acyclic-graph Synchronization Constraint Language) to achieve three desirable properties for a synchronization modeling language: declarative syntax, fine granularity and validation support. Instead of composing service out of structured constructs, DSCL declaratively describe the synchronization constraints in three basic relations on activity states. The state relationships collectively determine the execution order of activities in a composite process. The relationships are automatically translated into Petri Nets and simulated in the CPN/Tools, from which several correctness criteria can be validated for the composite process. We illustrate the advantages of DSCL with a Purchasing workflow example from BPEL 1.0 specification, and verify its correctness using CPN/Tools.

1. Introduction

Recent workflow specification languages such as BPEL [21], XPDL [22] have used structured constructs to describe the control flow of business processes. While there are advantages in making process component composition explicit, the procedural style of such languages also has its limitations. There are several known problems in the area of synchronization and concurrency modeling. For example, to specify mutually exclusive execution using XOR-split and XOR-join constructs may require an enumeration of all execution scenarios [1]. Another example is the tangled and scattered synchronization code when using BPEL's *link* construct to describe constraints for activities that are nested within different concurrent subprocesses.

Inspired by parallel programming language research on synchronization [4][19], we adopt three desirable properties for a synchronization modeling language. First, the language should have a ***declarative syntax***, so programmers only need to specify *what* to be synchronized instead of *how* to implement it. As a result, a process specification can be incrementally specified without demanding structural change in the base code. Second, the language should provide ***fine granularity*** control on synchronization constraints. Instead of being regarded as an atomic unit, the life cycle of an activity is a sequence of states and can be synchronized with other activities based on its current states. For instance, it should be possible to specify a synchronization constraint like "*Before starting the activity of closing a purchasing order, the activity of customer satisfaction investigation should have started and a survey has been sent out to the customer*". Third, the language should provide ***validation support***. A process architect needs tools to verify the synchronization behavior of processes, particularly for complex and evolving processes. We refer to three desirable properties (**D**eclarative syntax, **F**ine granularity and **V**alidation support) as *DFV properties* in the rest of the paper.

The main contribution of the paper is the introduction of DAG Synchronization Constraint Language (DSCL). DSCL provides the DFV properties by focusing on the synchronization aspect of processes, in a way similar to aspect-oriented programming applied to modeling [8]. DSCL specifies synchronization constraints in a declarative style using three basic state relations (*HappenBefore, HappenTogether, Exclusive*). They operate over the states of activities. An activity goes through three states during its life span, *start*, *run* and *finish*, each of which is viewed as a synchronization point. By specifying relationships over the states of activities, various synchronization behaviors can be described like *And-split* and *And-join* [22]. To demonstrate the validation of DSCL specifications, we translate DSCL into Colored Petri Nets (CPN). Problems such as deadlock and infinite synchronization sequence in a process can be found using CPN software tools [10]. The expressiveness of DSCL is illustrated with a motivation example (Section 3) and its limitations discussed in Section 4.

The rest of the paper is organized as follows. In section 2, we summarize related work. In section 3, we describe a realistic motivation example process (Purchasing). Section 4 describes the features of DSCL. Section 5 outlined a translation method mapping DSCL to Colored Petri Nets [17]. Section 6 uses DSCL to specify two processes. One is from the Purchasing example. The other is constructed to demonstrate those features not shown in the Purchasing process. Section 7 concludes the paper.

2. Related Work

Several synchronization modeling languages have been proposed in the literature, such as path expressions [4], synchronization expressions [19] and interactive expressions [9]. They provide constructs at different abstract levels to describe the concurrency in parallel application, although they offer limited support for validation. Different from the early work, DSCL models the process state explicitly. Synchronization depending on activity state is an important synchronization constraint in interactive system that can be observed in many real situations [1]. Furthermore, by establishing the mapping between DSCL and CPN, we can use CPN tools. Synchronization expressions have been mapped to Petri Nets [12], but they have not used Petri Net tools for validation. DSCL also contains additional technical features such as the *Exclusive* state relation.

As a domain specific language focusing on synchronization, DSCL supports a Directed-Acyclic-Graph (DAG) flow model. Therefore it characterizes part of the spectrum of synchronization behavior without including loop and conditional branching. We can see other DAG flow model in tools like Symphony [13]. UNICORE [6] is another tool that does not include conditional branching. Nontrivial workflow applications areas such as service deployment workflows have been completely specified without conditional branches [18].

Some projects have introduced synchronization at the granularity of activity state. In [18], a service deployment workflow is specified by putting synchronization constraints on the states of installation activity. In [19], activities are split into two parts, the start and the termination for inferring the correctness of synchronization history. DSCL integrates the support for fine-grain synchronization as part of DFV properties.

As the complexity of the process increases, it becomes more important to understand the behavior of the system under the occurrence of concurrency. Different techniques are used to simulate, test and verify the synchronization behavior of a process. Most work uses formal techniques like Petri net [17], and π-calculus [16] etc. Other work is based on model checking that has been widely used in automated software verification [3] [7]. We use Petri net to validate the synchronization behavior of the process because Petri net model state explicitly.

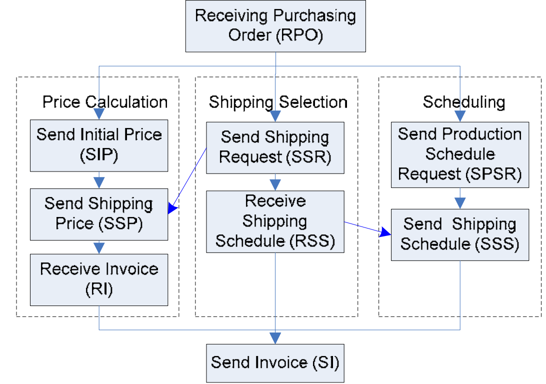

Figure 1. Purchasing process

3. Motivating Example

Let consider a Purchasing business process borrowed from BPEL 1.0 specification [21] in Figure 1.

The Purchasing process coordinates with three Web services: *PurchasingService*, *ShippingService* and *SchedulingService* in several steps to process a purchase. After receiving a purchase order, three subprocesses are instantiated concurrently: Price Calculation (*SIP*, *SSP* and *RI*), Shipping Selection (*SSR* and *RSS*), and Scheduling (*SPSR* and *SSS*). Due to the data dependency, the activities need to be synchronized to avoid race condition. For example, the shipping price produced by *SSR* is required to complete *SSP*. And the shipping schedule generated by *RSS* is required for the completion of *SSS*. We can observe its data flow in Figure 2.

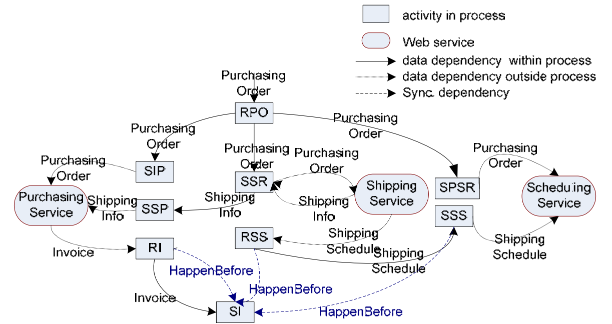

Figure 2. Data flow in the Purchasing process

These data flows introduce several synchronization constraints between the *finish* state of the data producer activities and the *start* state of the data consumer activities. We denote the *start* state and the *finish* state of an activity by *S* and *F* respectively. We introduce a state relation ***HappenBefore*** *(→)* to describe them.

$F_{RPO} \rightarrow S_{SIP}$ $F_{RPO} \rightarrow S_{SSR}$ $F_{RPO} \rightarrow S_{SPSR}$ $F_{SSR} \rightarrow S_{SSP}$
$F_{SSR} \rightarrow S_{RSS}$ $F_{RSS} \rightarrow S_{SSS}$ $F_{SSP} \rightarrow S_{RI}$ $F_{RI} \rightarrow S_{SI}$

The process further requires that the final invoice can be sent out only after the *SSS* successfully finishes such that a customer who receives a final invoice is guaranteed

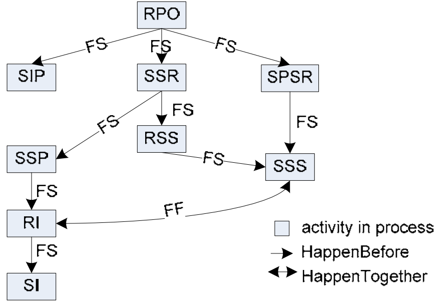

Figure 3: Synchronization relationships in Purchasing process

to receive her product. To describe it, we introduce another state relation ***HappenTogether (↔)***.

$\boldsymbol{F}_{\mathrm{RI}} \leftrightarrow \boldsymbol{F}_{\mathrm{SSS}}$

This statement declares the constraint that *RI* and *SSS* will enter their *finish* state together. When both of them finish, *RI* can notify *SI* to start.

For easy understanding, instead of writing down the state relationships in statement, we draw a synchronization graph to illustrate its synchronization constraints. We use a rectangle to represent an activity and a directed line to represent a state relationship. The line is labeled with states that the state relationship operates on. Figure 3 shows all the constraints for the Purchasing process.

From the Purchasing example, we can see that the orchestration of a process could be well captured in a set of relationship statements and incrementally developed without side-effect on previous constraints. In real life, the synchronization constraints in a process can be much more complicated than those in the Purchasing process. For example, two activities may update the backend system like database on shared data. In that case, these two activities need to be executed exclusively to guarantee the data consistency. A process architect needs assistance in describing, simulating and analyzing the synchronization behavior in a complicated process. In the rest of the paper, we detail each of these issues.

4. DAG Synchronization Constraint Language

In this paper, we use *business process* and *process* interchangeably. Individual steps in the process are called *activities*. In a process, each activity goes through three states: *start*, *run*, and *finish* (States related to exception handling such as abort, failure are subject of ongoing research and beyond the scope of this paper.). The activities interact and synchronize their transition from one state to another (in order).

DSCL is designed to describe Directed-Acyclic-Graph (DAG) synchronization behavior at activity level. A process architect can use it to declaratively specify the synchronization constraint in the form of relationship statements informally introduced in Section 3. DSCL omits loop and conditional branching synchronization relations to avoid the typical problems associated with such "procedural" facilities [14]. Similar to functional programming languages and logic components [5], we expect to combine DSCL with other aspect-oriented workflow specification languages as suggested in [20]. For example, it is desirable to decompose a workflow system into separate aspects (functional, control, data, organizational) for the purpose of creating easily adapted and fully distributed workflow. One possible design choice is to embed DSCL within a "general purpose" workflow specification language such as BPEL, in a way similar to some aspect-oriented domain specific languages embedded in Java. For example, Jeeg [15] decouples the synchronization code from the other code in java *Class*. The synchronization code is declaratively expressed in linear temporal logic and later woven into the rest of the code by a compiler.

In DSCL, We denote the state space (start, run, finish) by $N=\{S, R, F\}$. For every pair of states, we observe three basic relations between them: 1) the two states should be reached one after another; 2) the two states should be reached together. In other words, if one of the states cannot be reached due to some missing conditions, the other state should not be reached; 3) the two states are not allowed to coexist simultaneously at any point in time. We summarize their notation and semantics below.

- 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.

One advantages of modeling a process by these three relations is that they naturally reflect the way how business people use causal relationships to describe their business process in real applications.

We use ***a*** denoting an activity in a process, and *X* as the range of state space *N* for the state relations. X_i refers to some state *X* of activity $\boldsymbol{a}_{\mathrm{i}}$. We denote domain of relations by *Dom*. Below we give the definition of the state relations.

Definition 1: State relation is defined by $(X_{\mathrm{i}}, X_{\mathrm{j}}) \in \psi$, where $X_{\mathrm{i}}, X_{\mathrm{j}} \in \{S, R, F\}, \psi \in \{\rightarrow, \leftrightarrow, \mathrm{O}\}$. $Dom(\rightarrow)=\{S, F\}$, $Dom(\leftrightarrow)=\{S, F\}$ and $Dom(\mathrm{O}) = \{R\}$. $N(\psi)$ denotes all the states occurring in ψ.

ψ can describe a rich set of synchronization patterns. For example, $\boldsymbol{F}_{\mathrm{i}} \rightarrow \boldsymbol{S}_{\mathrm{j}}$ describes *sequence*. $\boldsymbol{S}_{\mathrm{i}} \leftrightarrow \boldsymbol{S}_{\mathrm{j}}$ and $\boldsymbol{F}_{\mathrm{i}} \leftrightarrow \boldsymbol{F}_{\mathrm{j}}$ together express *And-split* and *And-join*. More than that, they can describe those synchronization patterns that are fairly hard to express in traditional approach. For instance, if we want to specify the constraint of $\boldsymbol{S}_{\mathrm{i}} \rightarrow \boldsymbol{F}_{\mathrm{j}}$

such that activity a_j should not finish until activity a_i starts, this cannot be described by existing tools in that each activity is treated as an atomic synchronization unit. We could not specify those synchronization constraints for activity a_i and a_j when their life spans are partially overlapped. As a result, only sequential execution can be specified between them, which may make the process unnecessary sequential. In Section 5, Ψ is translated to Petri net. Intuitively, it is able to describe those workflow patterns that can be described by Petri net for activity state like *sequence, parallel split, synchronization, interleave parallel routing,* and *milestone* [1].

Next we declare two propositions that state the properties of Ψ. These two propositions can help us reduce the complexity of mapping from the state relations to Petri Net. We also use them to define the correctness criteria of synchronization constraints in terms of state relations.

Proposition 1:

i. Transitivity

$(X_i, X_j) \in \Psi_{\rightarrow} \wedge (X_j, X_z) \in \Psi_{\rightarrow} \Rightarrow (X_i, X_z) \in \Psi_{\rightarrow}$

$(X_i, X_j) \in \Psi_{\leftrightarrow} \wedge (X_j, X_z) \in \Psi_{\leftrightarrow} \Rightarrow (X_i, X_z) \in \Psi_{\leftrightarrow}$

ii. Symmetry

$(X_i, X_j) \in \Psi_{\leftrightarrow} \Rightarrow (X_j, X_i) \in \Psi_{\leftrightarrow}$

Proof The follows straightforward for Definition 1. □

Proposition 1 implies that → or ↔ only defines a partial relationship between two states. We need to form a global view to maintain the correct synchronization behavior. For example, if $S_i \leftrightarrow S_j$ and $S_j \leftrightarrow S_z$, it means that the *start* state of a_i and a_z should also be reached together. One abbreviation is $S_i \leftrightarrow S_j \leftrightarrow S_z$.

We use $\{X_i\}^+_{\rightarrow}$ to denote the transitive closure of states relation →, and use $\{X_i\}^+_{\leftrightarrow}$ to denote the transitive and reflective closure of state relation ↔. Notice that $\forall X_j \in \{X_i\}^+_{\leftrightarrow}$, $\{X_i\}^+_{\leftrightarrow} = \{X_j\}^+_{\leftrightarrow}$. That is if $X_1 \leftrightarrow X_2 \leftrightarrow X_3$, we can use any of them to denote the closure set $\{X_1\}^+_{\leftrightarrow} = \{X_2\}^+_{\leftrightarrow} = \{X_3\}^+_{\leftrightarrow}$.

Proposition 2: ↔ can be simulated by →. Given $X_1 \leftrightarrow X_2$, …, $\leftrightarrow X_k$ whose transitive and reflective closure is denoted as $\{X_1\}^+_{\leftrightarrow}$, the state relationships can be simulated in → by creating a coordinator activity a_c and replace $\{X_1\}^+_{\leftrightarrow}$ with new state relationships as follows:

(1) $\forall\ X_i \in N(\psi)$, if $\exists\ X_j \in \{X_1\}^+_{\leftrightarrow} \wedge X_i \rightarrow X_j$, then

$$\psi = \psi \cup \{X_i \rightarrow S_c\} - \{X_i \rightarrow X_j\}$$

(2) $\forall\ X_j \in \{X_1\}^+_{\leftrightarrow}$. Let $\psi = \psi \cup \{F_c \rightarrow X_j\}$

Proof

From (1) the activity a_c synchronizes with those states X_i that need to be reached before the states in $\{X_1\}^+_{\leftrightarrow}$ can be reached at its *start* state S_c. Then a_c notifies the states in $\{X_1\}^+_{\leftrightarrow}$ to be reached through (2) by synchronizing them at its *finish* state F_c. □

For example, given the following relationship statements $F_1 \rightarrow S_2$, $F_3 \rightarrow S_4$, $S_2 \leftrightarrow S_4 \leftrightarrow S_5$ for activity a_1, a_2, a_3, a_4 and a_5, we can replace $S_2 \leftrightarrow S_4 \leftrightarrow S_5$ by introducing a_c, removing $F_1 \rightarrow S_2$, $F_3 \rightarrow S_4$, and finally adding the additional relationships $F_1 \rightarrow S_C$, $F_3 \rightarrow S_c$, $F_c \rightarrow S_2$, $F_c \rightarrow S_4$, $F_c \rightarrow S_5$.

Proposition 2 implies that ↔ is a "syntax sugar" and we can always do preprocessing to translate ↔ to → by introducing coordination activity. Thus without loss of generality, the translation method in the next section focuses on → and O only.

Finally we define the correctness criteria for the state relationships in a process. There are two situations to consider. One is the dead end. The other is infinite synchronization sequence. Both of them are caused by the existence of cyclic synchronization relationships. Definition 2 gives the formal definition.

Definition 2: (correctness criteria of state relationships)

Given a process $A = \{a_0, a_1, ..., a_n\}$ with a_0 as the starting activity and the associated state relationships Ψ, Ψ is correct for process P if and only if it satisfies the following properties:

1. $\forall\ a_i \in A$, $\{S_i, R_i, F_i\} \subseteq \{S_0\}^+_{\rightarrow}$
2. $\forall\ X_i \in \Psi$, $X_i \notin \{X_i\}^+_{\rightarrow}$

By condition 1, there is no dead end in synchronization. All the states of activities are reachable from the start state of the root activity. By condition 2, it makes sure that the synchronization constraints are acyclic. There is no deadlock and infinite occurrence sequence. In other words, the state of an activity is not allowed to have a *HappenBefore* relationship with itself. Notice that the *Exclusive* relation O is not considered in the correctness criteria in that it only impacts the scheduling time of associated activities not their reachability.

5. Translation of DSCL to Petri Net

As a descriptive language, DSCL acquires its formal foundation by establishing a mapping to Petri nets. Petri nets are a formal and graphical language for modeling system behavior with concurrency. In this section, we demonstrate that state relations in DSCL can be translated to Petri net. By studying the properties of translated Petri net, we are able to check the synchronization correctness for a process. We first give a self-contained review for Petri net. For more details, please refer to [17].

5.1 Petri Net

Definition 3: (Petri Nets). A Petri Net is a four-tuple $C = (P, T, I, O)$:

i. P is a finite set of places.
ii. T is a finite set of transitions.
iii. $I : T \rightarrow P^{\infty}$ is the input function, a mapping from transitions to bags of places.

iv. $O: T \rightarrow P^{\infty}$ is the output function, a mapping from transitions to bags of places.

A transition is enabled if each of its input places has at east as many tokens in it as arcs from the place to the transition. The inputs and outputs of a transition t_j allow a place to be a multiple input or a multiple output of a transition. We use $\#(p_i, I(t_j))$ and $\#(p_i, O(t_j))$ to denote the number of occurrences of the place in the input bag of the transition and the place in the output bag of the transition respectively.

Definition 4: (marking). A marking μ of a Petri net $C=(P,T,I,O)$ is a function from the set of places P to the nonnegative integers *N*. $\mu=(n_0, n_1, ..., n_n)$ where the number of tokens in place p_i is n_i.

Definition 5: (transition firing and occurrence sequence). The firing of any enabled transition t_j at marking μ causes the change of the marking to a new marking μ' defined by

$\mu'(p_i)=\mu(p_i)-\#(p_i, I(t_j))+\#(p_i, O(t_j))$.

Two sequence results from the execution of a Petri net: the sequence of markings $(\mu_0, \mu_1, ..., \mu_n)$ and the sequence of transitions $(t_{j0}, t_{j1}, ..., t_{jn})$. The firing of t_{jk} under the marking μ_k leads to marking μ_{k+1}. We use $\delta(\mu_k, t_{jk})=\mu_{k+1}, for\ k=0,1,...$ to denote this relation. We also write it as $\mu_0[t_{j0} \succ \mu_1[t_{j1} \succ \mu_2 ...$ and call it occurrence sequence (*OS*). If the sequence has infinite length, we call it infinite occurrence sequence (OSI).

Definition 6: (reachability and reachability set). A marking μ' is said to be reachable from μ iff there is a firing sequence $t_1, t_2, ..., t_n$ such that $\mu[t_0 \succ \mu_1[t_2 \succ ... \succ \mu'$. The reachability set $R(C,\mu)$ for a Petri net $C=(P,T,I,O)$ with marking μ is the smallest set of markings defined by

1. $\mu \in R(C,\mu)$
2. if $\mu' \in R(C,\mu)$ and $\mu'' \in \delta(\mu', t_j)$ for some $t_j \in T$, then $\mu'' \in R(C,\mu)$

Finally we introduce the concepts of liveness, which have been considered in studies of deadlock. Liveness is categorized at four levels [17]. For the purpose of this paper, we are interested in *level 1* and *level 4*. For a Petri net $C=(P,T,I,O)$ with marking μ:

Definition 7 (liveness level 1): A transition is live at level 1 if it is potentially fireable. That is, if there exists $\mu' \in R(C,\mu)$ such that t_j is enabled in μ'.

Definition 8 (liveness level 4): A transition is live at level 4, if for each $\mu' \in R(C,\mu)$ there is a firing sequence σ such that t_j is enabled in μ'.

We can see that liveness level 4 is much strong than level 1. Liveness level 1 is used to identify whether there is unreachable transitions. Liveness level 4 is used to identify whether there is infinite firing sequence in the Petri net. Both of them can help us verify the correctness of synchronization relationships. We shall see this in Section 5.3. Before that, we make a useful statement.

Proposition 3: If a Petri net $C=(P,T,I,O)$ with marking μ is at level 4, there exists an infinite occurrence sequence.

Proof Prove by contradiction. If no infinite occurrence sequence exists, C must halt at a marking μ', at which no transition is enabled. This contradicts to Definition 8.

5.2 Translation from State Relationships to Petri Net

This section describes a method to derive a Petri net from state relationships of a process. Due to Proposition **2**, we can always do preprocessing to replace ↔ with **→.** Thus the method includes → and O only. Below we give an algorithmic description of the translation method.

The method follows three steps: the *intrastate relation construction*, which manages the state relations within an activity, and the *interstate relation construction*, which establish the state relations between activities and finally the *exclusive relation construction*, which handles the *Exclusive* relation. During the mapping, a place represents a state of an activity. Each transition controls the conditions that need to be satisfied before the activity can reach that state. The firing of a transition means that the dependent states at the inbound places have been reached. The correspondent activities can transmit to the next state.

Step 1: Intrastate relation construction

Intrastate relations reflect the state transition of an activity in its life span. Each activity goes through three states{*S, R, F*} that can be described in → relation. Its Petri net is shown in Figure 4 (a).

Step 2: Interstate relation construction

For each $X_i \rightarrow X_j$, i≠j, a place and two arcs are created to connect the outbound transition corresponding to X_i and the inbound transition corresponding to X_j. For example, the state relation $\boldsymbol{F}_i \rightarrow \boldsymbol{S}_j$ is connected by place p_x as illustrated in Figure 4 (b).

Step 3: Exclusive relationship construction

For exclusive relation O, a place with a shared token is added to the transitions that corresponding to the running state of all involved activities. The shared token at place p_y guarantees that only one of them is executed at any point in time, as seen in Figure 4 (c).

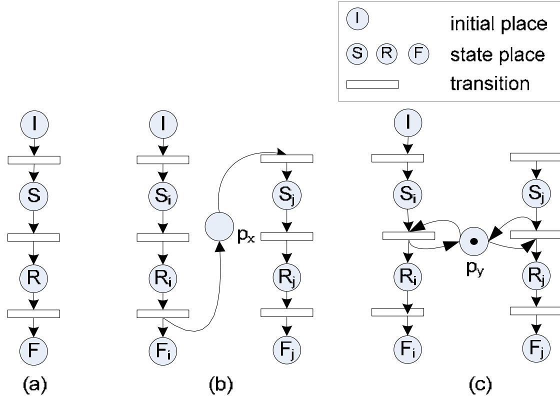

Figure 4: Translation from state relationships to Petri net

Proposition 4: Given a set of state relationships, the translated CPN correctly reflect the synchronization constraints prescribed by these relationships.

Proof Due to proposition 2, we only need to prove the proposition holds for → and O.

Step 1: Prove →

According to the interstate relation construction, given a *HappenBefore* relation $X_i \rightarrow X_j$, a place p and two arcs, (X_i, p) and (p, X_j), are added to the CPN. Thus the state X_j will not be entered until X_i fires and adds a token to p. Thus the construction maintains the semantic of →.

Step 2: Prove O

With the extension to the basic algorithm, all activities involved in the exclusive relationship share a token. Before an activity enters into its running state, it should obtain the exclusive access to the shared token and return it back after it finishes the execution. Thus the construction maintains the semantic of O.

In order to infer the correctness of a process in terms of synchronization relations by way of Petri net, we need to correlate the Petri net properties to the correctness criteria (Definition 2). Proposition **5** does the job.

Proposition 5: Given a set of activities $\boldsymbol{A} = \{a_0, a_1, \ldots, a_n\}$ and its associated state relationships Ψ, the synchronization constraints are correct if and only if its corresponding Petri net is at liveness level 1 and there is no transition at liveness level 4.

Proof this can be inferred from Definition 2, Definition 7, and Proposition 3 □

5.3 Simulation in CPN/Tools

We choose CPN/Tools [10] as the simulation tool. CPN/Tools is a graphical editor and simulator of Colored Petri Nets (CPNs). It is a high-level net allowing typed token to carry complex data. Basically, in CPN a place has a color indicating the type of the token this place can hold. The type of the token can be arbitrarily complex like a record in programming language. There are two considerations to choose it as the simulation tool. First, Colored Petri Net (CPN) is more expressive than Petri Net to describe a system [11]. This leaves us more flexibility if the DSCL is extended to accommodate richer synchronization behaviors, for instance conditional state transition. Second, the CPN/Tools provides substantial support for validation, i.e. a simulation tool, a state space tool for verifying various properties like liveness, boundedness and fairness [11]. Third, this tool takes a CPN formatted in XML. This facilitates us to translate the Petri net to the input of CPN/Tools. Since CPN requires that each place should have a type, we assign each place a single type *e*, which means no data contained.

In the simulation, we use the *State Space Toolkit*. This toolkit constructs an occurrence graph in which each node stands for a reachable marking and an arc for each firing of transition. Based on the occurrence graph, it generates information like the number of nodes in the graph, the bound for each place, dead and live transitions etc. In the generated report, we are particularly interested in two parts. One is the record for the dead transitions, which corresponds to the transition liveness level 1. If dead transition is none, the Petri net satisfy liveness level 1. The second is the size of state space and the number of Strongly Connected Components (SCC). A SCC is a maximal subgraph in which it is possible to reach from any node to any other node. If the number of SCC is fewer than the state space nodes, it implies that the net has at least one SCC with more than one node, which implies infinite occurrence sequences exist [10]. In other words, the net may not terminate. Thus we can infer the liveness level 4 by comparing the number of nodes in SCC and the number of nodes in the occurrence graph. Another useful tool is the *Simulation Toolkit*. It is useful when the state space of the net is unbounded.

6. Applications

In this section, we look at two processes to illustrate the DFV properties of DSCL in process modeling. The first process is the Purchasing process introduced in Section 3. The second one is artificially constructed to demonstrate those features that are not reflected in the first process.

6.1 Process 1 (Purchasing Process)

Description Based on the description in Section 3, its synchronization constraint graph is illustrated in Figure 5.

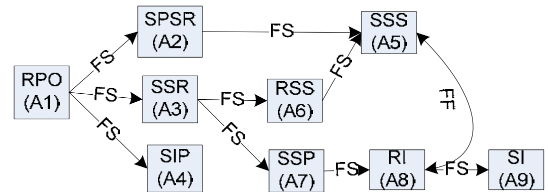

Figure 5: Synchronization constraints

Petri net translation We implemented a parser called SR2PN (State Relationships to Petri Nets) that reads these state relationships and translates them into Petri net. For each activity A_i, we name the Petri net places s_i, r_i, f_i corresponding to its *start*, *run* and *finish* state. The transitions between places are named as S_i, R_i, F_i respectively. The initial place s_1 is assigned with initial marking *1`e* (*1`e* represents one token with the color type *e*). For a transition t_i, if there is more than one outbound place, we call them x_{i0}, x_{i1}, …, x_{in}, $x \in \{s, r, f\}$ respectively. For a *HappenTogether* relationship between A_i and A_j, we name the coordinator activity A_{ij}. Figure 6 is a screenshot from the CPN/Tools.

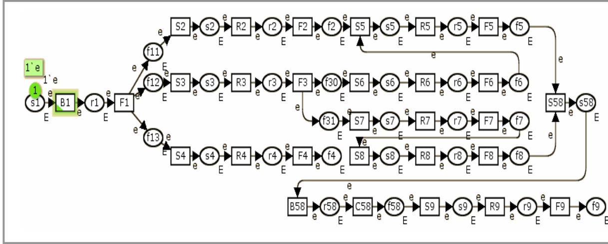

Figure 6. Petri net for process 1

We can see that there are three places and three transitions for each activity. That is why translated Petri net has more nodes than the number of activities. Actually for validation purpose, we can apply reduction rules to reduce its size [11].

Simulation result

```
Statistics
------------------------------------------------------------------------
 State Space
   Nodes:  391
   Arcs:   966
   Secs:   0
   Status: Full
 Scc Graph
   Nodes:  391
   Arcs:   966
   Secs:   0
   ... ... ...
Liveness Properties
------------------------------------------------------------------------
 Dead Transitions Instances: None
   ... ... ...
```

Figure 7. Simulation report from CPN/Tools

Figure 7 is part of the state space report generated by the State Space Toolkit. From the report, we can see that the size of state space is equal to that of SCC and no dead transition exists. Therefore, the synchronization behavior of the Purchasing process is correct.

6.2 Process 2

Description Process 2 consists of four activities A_1, A_2, A_3 and A_4. It has three synchronization constraints.

- $\boldsymbol{S}_{A1} \rightarrow \boldsymbol{S}_{A2}$: A_2 can start after A_1 starts. In other words, the execution of A_2 does not have to wait for A_1 to finish.
- $\boldsymbol{S}_{A1} \rightarrow \boldsymbol{F}_{A3}$: *A3* can only finish after A_1 starts. That is A_3 has to make sure A_1 has been started before it finishes. Otherwise, it has to wait until this event happens.
- $\boldsymbol{F}_{A3} \rightarrow \boldsymbol{S}_{A4}$: A_3 finishes first, and then A_4 starts.

To make it more interesting, suppose that the synchronization constraint $\boldsymbol{F}_{A4} \rightarrow \boldsymbol{S}_{A3}$ is accidentally added to the process. The synchronization constraint graph is portrayed in Figure 8. Intuitively, we would expect that there is a deadlock between A_3 and A_4.

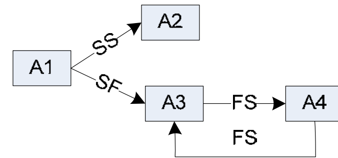

Figure 8. Synchronization constraints process 2

Petri net translation

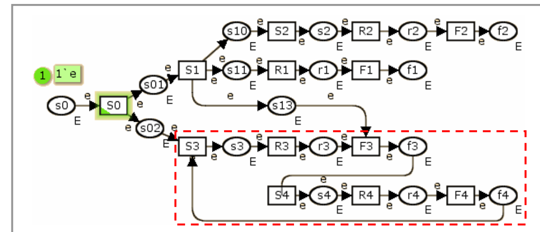

Figure 9. Petri net for process 2

The initial place s_0 is added to start A_1 and A_3 concurrently with the initial marking 1`e.

Simulation result

```
Statistics
------------------------------------------------------------------------
  ... ... ...
 Dead Transitions Instances:
 F3 1
 F4 1
 R3 1
 R4 1
 S3 1
 S4 1
```

Figure 10. Simulation report for process 2

From the report, we can see that those transitions involved in the loop are dead (highlight in the dashed rectangle). This is caused by the cyclic synchronization constraint in the state relations, which violates Definition 2. Therefore we should go back and check the correspondent constraints in the original specification for correction. The updated relationships should be feed into the SR2PN parser for another round of validation until the correctness criteria is satisfied.

7. Conclusion and Future Work

In this paper, we describe DSCL, a domain specific synchronization constraint language to support the DFV properties: *declarative syntax*, *fine granularity* and *validation support*. DSCL provides three state relations for a process architect to declaratively specify *what* to be done for synchronization instead of *how* to implement it. The feature of fine granularity is realized by specifying constraint on activity state instead of treating it as an atomic unit. The synchronization constraints specified by DSCL are finally translated to Petri net for simulation, analysis and validation.

We use a Purchasing workflow example (from BPEL 1.0 specifications [21]) to illustrate the expressiveness of DSCL. The synchronization constraints are specified using the three state relations of DSCL: *HappenBefore*, *HappenTogether*, and *Exclusive*. The DSCL specification is translated into CPN and verified using CPN tools, demonstrating the DFV advantages of DSCL.

DSCL could be used to deal with process synchronization in general. Our current work extends the practical application of DSCL. We plan to extend an existing workflow language (e.g., BPEL) with DSCL such that a process architect can model the synchronization aspect of a process at the abstract level of state relationships and let the compiler (or a code generation tool) produce the executable code automatically. For implementation, we may leverage on existing techniques capable of translating Petri net to BPEL [2].

8. Acknowledgements

This work was partially supported by NSF/CISE IIS and CNS divisions through grants IDM-0242397 and ITR-0219902, DARPA IPTO through grant FA8750-05-1-0253, and Hewlett-Packard.

9. 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. van der Aalst and K.B. Lassen. Translating Workflow Nets to BPEL4WS. BPM Center Report BPM-05-16, BPMcenter.org, 2005.

[3] A. Betin-Can, T. Bultan, X. Fu. Design for verification for asynchronously communicating Web services. Proceedings of the 14th international conference on World Wide Web, 750-759. Chiba, Japan, ACM Press. 2005.

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

[5] D. Churches, G. Gombas, et al. Programming Scientific and Distributed Workflow with Triana Services. In Grid Workflow 2004 Special Issue of Concurrency and Computation: Practice and Experience, to be published, 2005.

[6] D. W. Erwin. UNICORE: A Grid Computing Environment. Lecture Notes in Computer Science, Vol. 2150 Volume(Issue): 825-34. 2001.

[7] H. Foster, J. Kramer, J. Magee and S. Uchitel. Model-based Verification of Web Service Compositions. IEEE ASE 2003, Montreal, Canada. October 2003.

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

[9] C. Heinlein. Workflow and process synchronization with interaction expressions and graphs. Proceedings of 17th International Conference on Data Engineering. 243.2001.

[10] K. Jensen. An Introduction to the Practical Use of Coloured Petri Nets. In: W. Reisig and G. Rozenberg (eds.): Lectures on Petri Nets II: Applications, Lecture Notes in Computer Science vol. 1492, Springer-Verlag 1998, 237-292.

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

[12] P. E. Lauer and R. H. Campbell. A description of path expressions by Petri nets. Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Palo Alto, California, ACM Press. 1975.

[13] M. Lorch and D. Kafura. Symphony - A Java-based Composition and Manipulation Framework for Computational Grids. In Proceedings of 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002), Berlin, Germany. May 21-24, 2002.

[14] B. Ludäscher, I. Altintas, et al. Scientific Workflow Management and the Kepler System. Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows, to appear, 2005.

[15] G. Milicia, V. Sassone. Jeeg: temporal constraints for the synchronization of concurrent objects. Concurrency - Practice and Experience 17(5-6): 539-572 2005

[16] R. Milner. Communicating and Mobile Systems: the Pi-Calculus, Cambridge University Press; 1st edition.June 15, 1999

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

[18] A. Sahai, S. Singhal, V. Machiraju and R. Joshi. Automated Generation of Resource Configurations through Policy. Policy-2004, New York. June 7-9, 2004.

[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. Proceedings of the ICSE98 AOP Workshop, pages 41 - 46, Kyoto, April 1998.

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

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