Hierarchical Process Verification in a Semi-trusted Environment

Ganna Monakova

SAP Research Karlsruhe
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe, Germany
ganna.monakova@sap.com

Abstract. In a business collaboration different parties work together to create a compositional process that incorporates internal processes of the participants to achieve a common goal. It must be assured that certain requirements, such as legal regulations, are fulfilled by the overall composition. Verification of such requirements requires knowledge about the internal functionalities of the involved parties, who in turn do not want to reveal their processes. This work presents a technique for hierarchical verification of the requirements over the process composition based on the guarantees, called *property assertions*, provided by the collaboration participants.

1 Introduction

Business collaborations require participants to work together to create a composite process that incorporates internal processes of the involved parties. It has to be assured that the resulting process composition satisfies certain requirements, such as applicable legal regulations or service agreements between the parties. If participants fully trust each other, then they can reveal their internal processes. In this case verification of a requirement over composition (called compositional requirement) is no different from verification of a requiement over a local business process, which has been widely discussed in the literature, see Section 5. If, however, the operating environment is not fully-trusted, then participants will not reveal their internal processes as they might contain sensitive information, such as business decisions captured in the process structure. In this case the problem of *How to verify a requirement over process composition without knowledge of the processes* arises.

In this paper we propose the following solution to this problem: instead of revealing the processes, participants provide *property assertions* that reflect certain characteristics of their processes [1]. A property assertion specifies *what* is guaranteed by a process, as opposite to the process structure that reveals *how* a

[1] A semi-trusted environment is required, as some of the (abstracted) information must be revealed to enable verification of the process composition. A non-trusted environment would prohibit revelation of any information.

guarantee is achieved. Provided property assertions only need to contain information required by the compositional requirement. Therefore property assertions allow to hide implementation details but still provide sufficient information to enable verification of compositional properties.

The derivation of an appropriate process abstraction that only reveals process properties necessary for a requirement verification is the subject of a different work. In this paper we show how the compositional requirement can be proven based on the provided property assertions. The presented technique can be used for the bottom-up verification, as well as for the top-down process design:

- In a bottom-up approach the existing processes abstract their implementation with the property assertions. This on one hand allows for verification of the requirements in a semi-trusted environment, on the other hand reduces the complexity of the verification problem due to the abstraction of the unnecessary implementation details.
- In a top-down process design the top-level requirement is broken down into sub-requirements that are assigned to the different parties as requirements on the future process implementation. In this case the requirements become the property assertions and can be used as a contract between the participants.

In both cases it is necessary to be able to prove compositional requirements based on the given property assertions and on the structure of the composition, which is the scope of this paper.

1.1 Running Example

As a running example consider a supply chain process shown in Fig. 1. The example collaboration includes three participants: *Retailer*, *Producer* and *Reseller*. The internal processes of the participants are not visible. Instead, participants specify a number of property assertions that are fulfilled by the internal process implementations. We use a semi-formal notation for the property assertions at this point, later in Section 3 we define the formal property assertion language. The flow of the process is the following: *Retailer* computes product demand and, depending on the number of required items, sends an order request either to *Reseller* or to *Producer*. *Reseller* does not reveal their process structure, but guarantees that if the number of ordered items is less than 5000, then the products can be delivered via an express delivery. *Reseller* also specifies that the express delivery is *preceded by* quality check. *Producer* specifies that an order will be either rejected or the goods will be produced and delivered. *Producer* also guarantees that any order above 10.000 will be accepted. They furthermore guarantee that the production of goods is *followed by* a quality check.

As an example requirement over the collaboration we consider the *Product quality must be checked before it can be sold* requirement. In this paper we show how this requirement can be formally proven based on the provided property assertions.

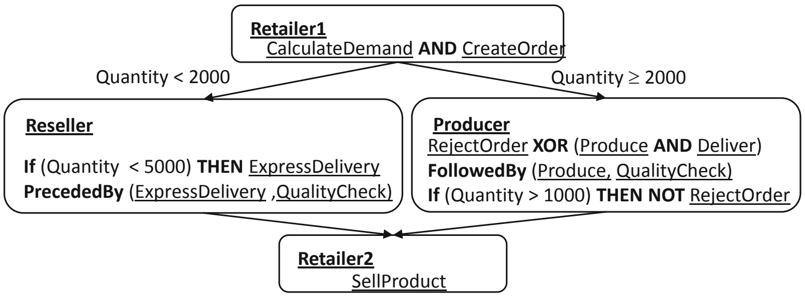
Retailer1
CalculateDemand AND CreateOrder
Quantity < 2000
Quantity ≥ 2000
Reseller
If (Quantity < 5000) THEN ExpressDelivery
PrecededBy (ExpressDelivery ,QualityCheck)
Producer
RejectOrder XOR (Produce AND Deliver)
FollowedBy (Produce, QualityCheck)
If (Quantity > 1000) THEN NOT RejectOrder
Retailer2
SellProduct

Fig. 1. Example Service Composition

1.2 Approach Overview and Paper Structure

To reflect hierarchical structure of a collaboration, where internal processes are composed to the overall collaboration, we use nested graphs as the formal language for collaborative models. A node in a nested graph can be a graph itself, in which case it is called a hypernode. A nested graph naturally captures semantics of scopes and reflects local and global process visibility through the notion of hypernodes. Hiding the structure of a hypernode on a higher level corresponds to abstraction of an internal process on a higher level.

To be able to prove requirements over the nested process graphs, we introduce a set of generic axioms in Section 2 that reflect execution semantics of nested process graphs. The presented axioms are used to create a set of assertions that model execution semantic of a particular process model. We call such assertions a *verification basis*. A hypernode in a nested graph provides some guarantees about the internal processes in form of process assertions. Section 3 introduces property assertion language. Section 4 shows how the provided property assertions are used to generate models of internal processes. Generated models are added to the verification basis to complete the process specification. Finally, we show how the compositional requirement is mapped to an assertion over the process activities. The mapping is based on the activity types that are used in the requirement specification. The satisfiability of the modelled assertions is checked using the Satisfiability Modulo Theories (SMT) solver Z3 [13]. An SMT solver solves satisfiability problems for Boolean formulas containing predicates of underlying theories. Such theories can be, for example, theories of arrays, lists and strings [2]. In addition, an SMT solver can be extended with new theories as shown in [16]. In the proposed approach we use the theory of the linear arithmetic.

Summarizing, the contribution of this paper are three-fold: first, we present a formal theory of hierarchical process models based on the nested graphs and show how to model process execution semantics with logical assertions. Second, we present an assertion language for specifying process properties without revealing the structure and business logic of internal processes. Third, we present a hierarchical verification technique for verifying global properties of distributed business processes based on the hierarchical process theory.

2 Nested Process Graph Theory

A process graph is a directed acyclic nested graph $G = (A, L)$, where A is the set of (hyper)nodes representing process activities and $L = \{L_k = (A_i, A_j) | L_k.\text{source} = A_i \wedge L_k.\text{target} = A_j\}$ is the set of directed edges representing synchronisation links between activities, where L_k.source denotes the source and L_k.target the target activity of link L_k. Activities can be simple or structured activities, in the second case the activity is represented through a directed acyclic subgraph that is viewed as a hypernode from the parent graph. A hypernode is used to represent a subprocess or a process scope.

Each link L_i in a process graph has a transition condition L_i.condition, with default condition being set to *true*. After completion of an activity, states of all outgoing links are evaluated as follows: if activity has been executed and transition condition of an outgoing link L_i evaluates to $true$, then the link status $Status(L_i)$ evaluates to $true$; if activity has been skipped or if the transition condition evaluates to $false$, then link status $Status(L_i)$ evaluates to $false$.

Let $L^{in}(A_i) = \{L_i | L_i.\text{target} = A_i\}$ denote the set of all incoming links for activity A_i. Each activity in the process graph has a join condition $JC(A) = F(Status(L_1^{in}), \ldots, Status(L_k^{in}))$, which is a logical expression over the states of the incoming links. The join condition is evaluated after the states of all incoming links have been evaluated. If the activity join condition evaluates to $true$, then activity starts its execution, otherwise it is skipped. The default join condition of each activity is the OR function over the incoming link states.

2.1 Activity Modelling and State Axioms

Each activity in a process has an execution lifecycle represented through the following activity states: *ready*, *started*, *skipped*, *executed*, *faulted*, *terminated*, and *compensated*. Figure 2 shows possible transitions between these states. Activity reaches state *ready* when control flow reaches this activity, which happens when all predecessor activities complete their execution and the statuses of all incoming links are evaluated. When an activity is reached, its join condition is evaluated. If it evaluates to $true$, then activity is *started*, otherwise it is *skipped*. If an activity has been *started*, then it either successfully finishes its execution and reaches state *executed*, faults and goes into the state *faulted*, or it terminates and goes into the state *terminated*. When an activity reaches states *skipped* or *executed* it is automatically reaches state *completed*. This triggers evaluation of the outgoing link statuses, which means that successor activities no longer have to wait for this activity to complete. The state diagram contains two additional transitions which are depicted with dashed lines: transition from *executed* to *compensated* state happens if a compensation actions are taken at some point after execution of the activity, transition from *faulted* to *completed* happens if a fault handler repairs activity fault and allows for further execution of the process. These two state transitions are only possible if the corresponding constructs (fault handler and compensation handler) have been defined in the process model.

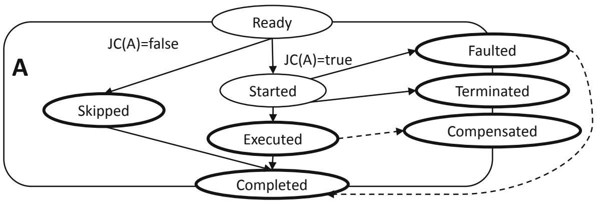

Fig. 2. Activity state transitions

In addition to the state variables, each activity is characterised by the activity type it belongs to. We use A^T notation to refer to an activity type, in contrast to A that is used to denote an activity. Activity types can be arranged into a type-subtype hierarchy. For example, a more fine-granular activity type *AirTransport* is a subtype of a more coarse-granular activity type *Transport*. Each activity type in our approach is represented through an integer interval $\mathit{ActivityType} := \mathit{Record}(\mathit{start} : \mathit{int}, \mathit{end} : \mathit{int})$. The subtype relations are then represented through the interval inclusion:

$$A_i^T \subset A_j^T \Leftrightarrow (A_i^T.\mathit{start} \geq A_j^T.\mathit{start}) \wedge (A_i^T.\mathit{end} \leq A_j^T.\mathit{end})$$

For example, an activity type $Transport$ is the supertype of activity types $AirTransport$ and $GroundTransport$. A possible interval assignment to these types would be $[0, 10]$ for $Transport$ and $[0, 3]$ for $AirTransport$ and $[4, 6]$ for $GroundTransport$.

To reflect all activity properties, we define activity type as a record consisting of the following fields:

$$\mathit{Activity} := \mathit{Record}(\mathit{ready} : \mathit{int}, ..., \mathit{completed} : \mathit{int}, \mathit{atype} : \mathit{int})$$

The value of *atype* field decides to which type this activity belongs: $A \in A^T \Leftrightarrow A.\mathit{atype} \geq A^T.\mathit{start} \wedge A.\mathit{atype} \leq A^T.\mathit{end}$. For example, if a process specifies that it executes an activity of type $Transport$, then the *atype* field of variable A_T that represents this activity will be restricted to $(A_T.\mathit{atype} \geq 0) \wedge (A_T.\mathit{atype} \leq 10)$. Similar, if we assign 2 to the *atype* field, then we specify that this activity is of type $AirTransport$ and $Transport$. We will use activity types to define properties of the subprocesses in Section 3.

To represent activity state model and capture semantic of the state transitions, we use eight integer fields: A.ready, A.started, A.skipped, A.executed, A.completed, A.faulted, A.terminated and A.compensated. The value of the state field represents the point of time, starting from 0 denoting the process start, when activity reaches this state. If activity does not reach a certain state, the value of the corresponding state field is -1. Axioms from Table 1 capture state transition semantics of each activity A by defining dependencies between the field values:

- Axiom (N_1) defines that each activity will eventually be started or skipped if it has been reached. We assume that skipping happens in the next step $(+1)$, this assumption can be changed if required. Starting an activity can

Table 1. Activity state transitions axioms

$A.\text{ready} > 0 \rightarrow (A.\text{started} \geq A.\text{ready} \wedge A.\text{skipped} = -1) \vee (A.\text{skipped} = A.\text{ready} + 1 \wedge A.\text{started} = -1)$	(N_1)
$A.\text{started} > 0 \rightarrow ((A.\text{faulted} > A.\text{started}) \vee (A.\text{terminated} > A.\text{started}) \vee (A.\text{executed} > A.\text{started}))$	(N_2)
$A.\text{started} > 0 \rightarrow ((A.\text{faulted} = -1 \wedge A.\text{terminated} = -1) \vee (A.\text{faulted} = -1 \wedge A.\text{executed} = -1) \vee (A.\text{executed} = -1 \wedge A.\text{terminated} = -1))$	(N_3)
$A.\text{started} > 0 \leftarrow (A.\text{faulted} > 0 \vee A.\text{terminated} > 0 \vee A.\text{executed} > 0)$	(N_4)
$A.\text{compensated} > 0 \rightarrow A.\text{executed} > 0 \wedge A.\text{compensated} > A.\text{executed}$	(N_5)
$A.\text{skipped} > 0 \rightarrow A.\text{completed} = A.\text{skipped}$	(N_6)
$A.\text{executed} > 0 \rightarrow A.\text{completed} = A.\text{executed}$	(N_7)
$A.\text{completed} > 0 \rightarrow A.\text{executed} > 0 \vee A.\text{skipped} > 0$	(N_8)

on the other hand take longer if we consider resource requirements, therefore we use the $\geq$ operator.
- Axioms (N_2) – (N_4) state that if an activity has been started, then it will either complete successfully, fail, or be terminated, whereby only one of these states can be reached.
- Axiom (N_5) specifies that an activity can only be compensated if it has been executed.
- Axioms (N_6) – (N_8) state that an activity reaches state completed only if and as soon as it has been skipped or executed.

Table 2. Fault, termination, and compensation handler axioms

$A_F.\text{ready} = A.\text{faulted}$	(F_1)
$A_T.\text{ready} = A.\text{terminated}$	(F_2)
$A_C.\text{started} > 0 \rightarrow A.\text{executed} > 0$	(F_3)
$A_C.\text{executed} > 0 \rightarrow A.\text{compensated} = A_C.\text{executed}$	(F_4)

If there is a fault handler A_F, compensation handler A_C or termination handler A_T defined for activity A, then the additional state transition axioms depicted in Table 2 need to be added to the general state transition axioms.

2.2 Parent-Child Axioms

In addition to the relations between the states of a single activity, relations between parent and child activities represented through the hypernodes, which represent process scopes, need to be specified. If A_i is a node inside the hypernode A, then we call A the parent activity of child activity A_i.

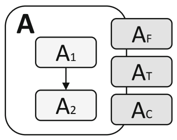

Fig. 3. Parent-Child

Figure 3 shows an example hypernode with two children activities, Figure 4 shows relations between the states of the parent and child activities.

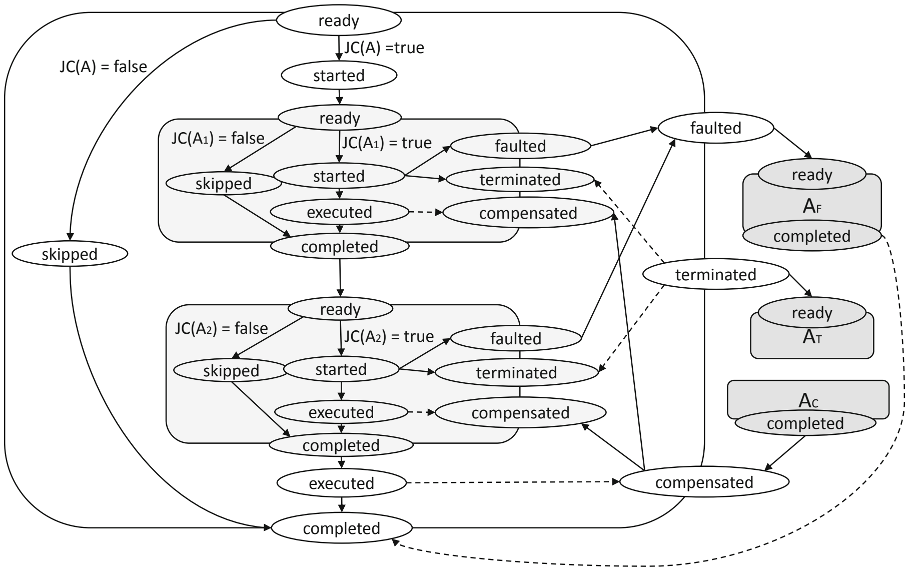

Fig. 4. Parent-Child State Transitions

Table 3. Parent-child state dependencies axioms

$A_i.\text{ready} > 0 \rightarrow A.\text{started} > 0 \wedge A_i.\text{ready} \geq A.\text{started}$	(H_1)
$A.\text{skipped} > 0 \rightarrow A_i.\text{skipped} > 0 \wedge A_i.\text{skipped} = A.\text{skipped}$	(H_2)
$A.\text{completed} > 0 \rightarrow \bigwedge_{i\in[1..n]}(A_i.\text{completed} > 0 \wedge A.\text{completed} \geq A_i.\text{completed})$	(H_3)
$A.\text{terminated} > 0 \rightarrow (A_i.\text{completed} > 0 \wedge A_i.\text{completed} < P.\text{terminated}) \vee (A_i.\text{started} > 0 \wedge A.\text{terminated} = A_i.\text{terminated}) \vee (A_i.\text{ready} < 0)$	(H_4)
$A_i.\text{faulted} > 0 \rightarrow A.\text{faulted} > 0 \wedge A.\text{faulted} = A_i.\text{faulted}$	(H_5)
$\forall i \in [1..n] : A_i.\text{faulted} > 0 \rightarrow (\forall j \in [1..n] : j \neq i \rightarrow ((A_j.\text{ready} > 0 \wedge A_j.\text{terminated} = A_i.\text{faulted}) \vee (A_j.\text{completed} > 0 \wedge A_j.\text{completed} < A_i.\text{faulted}) \vee A_j.\text{ready} < 0)$	(H_6)
$A_C.\text{ready} > 0 \rightarrow A_{iC}.\text{ready} > A_C.\text{ready}$	(H_7)
$A_C.\text{completed} \geq A_{iC}.\text{completed}$	(H_8)
$\forall (A_i, A_j) \in L : (A_i \in Children(A) \wedge A_j \in Children(A)) \rightarrow (A_{iC}.\text{ready} > A_{jC}.\text{completed})$	(H_9)

Table 3 contains a set of axioms that capture parent-child state relations:

- Axiom (H_1) specifies that a child activity can only be reached if the parent activity has started.
- Axiom (H_2) specifies that if parent activity is skipped, then all children activities are skipped too.
- Axiom (H_3) specifies that a parent activity completes when all child activities complete
- Axiom (H_4) specifies the termination semantic: If the parent activity is terminated, all of it children activities, which are still running, are terminated. Therefore all children activities will either be completed at the point of parent activity termination, will not be reached yet, or will be terminated together with the parent activity (immediate termination).
- Axiom (H_5) specifies the fault propagation semantic: Fault propagation is applied if there is no explicit fault handler defined for activity A_i. In this case any fault of any child activity is propagated to the parent activity. If a child activity has a fault handler, then the default fault propagation behaviour is overwritten by the fault-handler execution semantic. In this case activities defined in the fault handler are modelled using the same set of axioms, and dependency between fault handler and corresponding activity is modelled according to the fault axiom F_1.
- In addition to the fault propagation, axiom (H_6) specifies that if an activity has faulted, then all still running activities in the same scope are terminated.
- Axioms (H_7) – (H_9) specify the default compensation handler semantic, which invokes compensation handlers of the child activities in a reverse order. Here A_C denotes the compensation handler of the parent activity A, and A_{iC} denotes compensation handler of a child activity A_i.

2.3 Process Structure Axioms

Table 4 lists axioms that reflect dependencies between activities based on the process structure.

- (S_1) models synchronisation dependencies of a process activity A_i: An activity in a process graph is executed if it has been reached and its join condition evaluates to *true*. An activity is reached if all predecessor activities have completed their execution, which means A.completed > 0 is *true*. This can happen through activity execution, activity skipping, or reparation of a faulted activity through a fault handler as described in previous section. Each link in a flow graph represents a synchronisation dependency between the source and the target nodes. We model each link in a process graph as a variable of type $\mathit{Link} := \mathit{Record}(\mathit{source} : \mathit{Activity}, \mathit{target} : \mathit{Activity}, \mathit{condition} : \mathit{bool})$, where *condition* represents the link transition condition. $D(A_i)$ denotes the set of nodes that activity A_i is synchronised on. This set is computed as $D(A_i) = \{A_j | \exists L_k \in L^{in}(A_i), L_k.\text{source} = A_j\}$.
- Axiom (S_2) specifies that if all predecessor activities of activity A_i have completed, then A_i will reach state ready.

- S_3 and S_4 specify the link status evaluation rules for any link L.
- Axioms (S_5) and (S_6) reflect the control flow after an activity has been reached: if activity join condition evaluates to *true* , then activity will start as defined by (S_5), otherwise it will be skipped as defined by (S_6).

Table 4. Process structure axioms

$\forall A_j \in D(A_i)$: $(A_i.\text{ready} > 0 \rightarrow A_i.\text{ready} \geq A_j.\text{completed} \wedge A_j.\text{completed} > 0)$	(S_1)
$\bigwedge_{A_j \in D(A_i)} A_j.\text{completed} > 0 \rightarrow A_i.\text{ready} > 0$	(S_2)
$L.\text{source.executed} > 0 \rightarrow \mathit{Status}(L) = L.\text{condition}$	(S_3)
$L.\text{source.skipped} > 0 \rightarrow \mathit{Status}(L) = \mathit{false}$	(S_4)
$A.\text{ready} > 0 \wedge JC(A) \leftrightarrow A.\text{started} > 0$	(S_5)
$A.\text{ready} > 0 \wedge \neg JC(A) \leftrightarrow A.\text{skipped} > 0$	(S_6)

3 Property Assertions

A property assertion guarantees certain behaviour of a process without revealing how it has been implemented. Therefore property assertions allow participants in a process collaboration to hide the process implementation details and offer a natural form of abstraction. Due to their descriptive, yet formal nature, property assertions can be used as a declarative description of a subprocess in the bottom-up process composition, as well as a requirement specification for a subprocess in the top-down process design, called process refinement.

Property assertions make statements over activity types rather than actual activities. This supports specification of properties with the different abstraction levels, depending on the granularity of the chosen activity type. If we define a property assertion PA over activity type $Transport$ which is defined by interval $[0, 10]$, we would restrict the set of activities these assertions apply to as follows: $\forall A : (A.atype \geq 0 \wedge A.atype \leq 10) \rightarrow PA(A)$. Using more fine-granular activity types allows for specification and verification of more specific constraints, but it might reveal unnecessary details.

In this work we use an SMT solver to prove collaboration properties. Usage of SMT solvers allows us to specify requirements in propositional logic, as well as use additional theories, such as linear arithmetic theory. First Order Logic (FOL) over finite domains can be mapped to the propositional logic over the elements of the corresponding domain. The domain of an activity type is defined by the number of activities of this type present in the process model. As there is a limited number of activities in a process model, we will use FOL over activity types to express properties of the corresponding activities.

In the following sections we will show some examples of the process properties that can be expressed using combination of linear arithmetic with the logical operators. The examples are chosen to demonstrate expressiveness of the language. To ease property specification for a business user, property templates for the common properties can be defined similar to the Count and *FollowedBy* templates that are defined in the following sections.

3.1 Activity Occurrence Specifications

A restriction on the number of activity executions of a specific type is specified through $N_{min} \leq \text{Count}(A^T) \leq N_{max}$ constraint, which allows execution of minimum N_{min} and maximum N_{max} activities of type A in any process run. We define Count function of an activity A_i and of activity type A^T as follows:

$$\text{Count}(A^T) = \sum_{A_i \in A^T} \text{Count}(A_i) = \sum_{A_i \in A^T} \begin{cases} 1 & \text{if } A_i\text{.executed} > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

Although activity state .executed is the one that will be most commonly used to express activity occurrence, we can extend count function to allow count of any other activity states, that would allow to count the number of, e. g., failed or compensated activities. The extended count function $\text{Count}(A^T, S)$ counts the number of activity of type A_T that reach state A_S:

$$\text{Count}(A_i, S) = \begin{cases} 1 & \text{if } A_i.S > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

In some cases we will want to restrict time period for an activity state transfer. For this purposes we can refine Count template with the time parameters:

$$\text{Count}(A^T, S_1, S_2)_{[t_1,t_2]} = \sum_{A_i \in A^T} \begin{cases} 1 & A_i.S_1 \geq t_1 \wedge A_i.S_2 \leq t_2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

Where t_1 and t_2 can be relative or absolute timestamps.

Using refined count template we can specify that minimum MAX and maximum MIN activities executions of type A^T must happen in time period $[t_1, t_2]$ as follows:

$$MAX \leq \text{Count}(A^T, \textit{started}, \textit{executed})_{[t_1,t_2]} \leq MIN \tag{3.4}$$

3.2 Temporal Properties Specifications

To capture temporal relations between activities we specify relations between activity state fields using linear arithmetic operators $>$, $<$, $\geq$, $\leq$ and $=$. In addition, we use operators $+$ and $-$ over activity state variables and integers representing time intervals.

Having different states for each activity allows us to restrict activity execution duration as follows:

$$\begin{aligned} A\text{.executed} > 0 \rightarrow A\text{.completed} - A\text{.started} \geq MinDuration(A) \\ \wedge A\text{.executed} - A\text{.started} \leq MaxDuration(A) \end{aligned} \tag{3.5}$$

Similarly, temporal dependencies between different activities and their states can be specified using arithmetic operators. To demonstrate the expressiveness of the requirement specification language we consider *FollowedBy* example constraint that has been often used in the process verification approaches and show how different semantic variations of this constraint can be specified using FOL over activities, predicates and operators from linear arithmetic over activity states, and Count function defined above.

The *FollowedBy* constraint used in [19] is the A^T *must be followed by* B^T. To express such a requirement in our model we use activity state variables and linear arithmetic to specify the $FollowedBy$ template as follows:

$$\begin{aligned} FollowedBy(A^T, S_1, B^T, S_2) := \forall A \in A^T : A.S_1 > 0 \rightarrow \\ \exists B \in B^T : B.S_2 > A.S_1 \end{aligned} \tag{3.6}$$

Similar to $FollowedBy$ template, $PrecededBy$ template can be defined as follows:

$$\begin{aligned} PrecededBy(A^T, S_1, B^T, S_2) := \forall A \in A^T : A.S_1 > 0 \rightarrow \\ (\exists B \in B^T : B.S_2 > 0 \wedge B.S_2 < A.S_1) \end{aligned} \tag{3.7}$$

We assume that $FollowedBy$ template has default parameters $S_1 = executed$ and $S_2 = started$, as well as that $PrecededBy$ has default parameters $S_1 = started$ and $S_2 = executed$. This means that semantic of $FollowedBy(A^T, B^T)$ is equivalent to $FollowedBy(A^T, executed, B^T, started)$. By varying activity states in these templates we can slightly change their semantic, for example $FollowedBy(A^T, started, B^T, started)$ would allow parallel executions of $A \in A^T$ and $B \in B^T$ as long as B starts after A has started.

To extend the $FollowedBy$ requirement with the time restrictions as used in [8] to express B^T *must follow* A^T *within N time units* constraint, we can define the $FollowedByWithin$ template as follows:

$$\begin{aligned} FollowedByWithin\ (A^T, S_1, B^T, S_2, T) := \forall A \in A^T : \\ A.S_1 > 0 \rightarrow \exists B \in B^T : (B.S_2 > A.S_1) \\ \wedge (B.S_2 < A.S_1 + T) \end{aligned} \tag{3.8}$$

Constraint $FollowedByWithin(A^T, executed, B^T, started, 2)$ specifies that B must start after A if it has been executed and has to finish its execution within 2 time units. It can be varied to specify that B must be *completed* within 2 hours after execution of A or that it has to reach a certain state in a time frame relative to start of A. In general, any states defined in Section 2.1 can be used to parameterise these templates.

3.3 Data Dependent Properties

A property assertion can be refined by adding data conditions over the process input data D_i that influence execution of activity. To specify that A_i will be executed if and only if $C(D_i)$ evaluates to true, we use following property assertion:

$$(\text{Count}(A_i, executed) \leq 1) \wedge (C(D_i) \leftrightarrow \text{Count}(A_i, executed) = 1)$$

If activity execution condition contains expressions over local variables or if the exact activity execution condition should not be exposed, it can be abstracted to a necessary or a sufficient condition. A sufficient condition $C(D_i)$ leads to execution of the activity: $C(D_i) \rightarrow \text{Count}(A_i, \mathit{executed}) > 0$, while necessary condition is fulfilled when activity is executed: $\text{Count}(A_i) = 1 \rightarrow C(D_i)$. Similar to the activity occurrence constraint, any other property specification can be refined with linear data conditions over the input data.

4 Hierarchical Process Verification Using Process Theory

In this section we will show how a requirement R over the parent process P can be proven or disproven based on the property assertions $PA(P_i)$ of the subprocesses $P_1, ..., P_n$, and on the structure of P.

4.1 Representing Subprocess Abstractions

Property assertions of a subprocess make statements about the internal process behaviour. Such assertions need to be adapted to be included into the global context of the overall process. For example, consider a subprocess P_k that makes a statement $Count(A_i^T, executed) = 1$ with respect to its behaviour. Furthermore, consider a subprocess P_j that makes a statement over its behaviour about the same activity type $2 \leq Count(A_i^T, executed) \leq 3$. Without adjustment of these assertions to the global context, the combined assertion set would contain both statements, which obviously would lead to a contradiction.

To adjust a local assertion to the global context, we create a set of unique variables for each subprocess that represent internal activities of the subprocess. In the above example we would create a variable ${A_i}_1^{P_k}$ that would represent activity of type A_i in process P_k, and three variables ${A_i}_1^{P_j}$, ${A_i}_2^{P_j}$ and ${A_i}_3^{P_j}$ to represent activities of type A_i in process P_k. Count assertion is then mapped to the assertions over the internal activities. For P_k it will be equivalent to

$${A_i}_1^{P_k}.\text{executed} > 0$$

And for P_j it will be equivalent to

$${A_i}_1^{P_j}.\mathit{executed} > 0 \wedge {A_i}_2^{P_j}.\mathit{executed} > 0$$

Without loss of generality we specified that the first two activities are always executed. There is no statement about the third activity, leaving a freedom for SMT solver to decide whether it will be executed or not.

Let us assume a subprocess P_i provides a set of assertions $PA(P_i)$ over activity types $T(PA(P_i)) = \{A_{i_1}^T, ..., A_{i_k}^T\}$. Let us assume that the overall process requirement R is defined over activity types $T(R) = \{A_1^T, ..., A_m^T\}$. In the first step we need to create a generic model of P_i with respect to the overall requirement R. A model consists of the subprocess activities respresented through the corresponding variables, as well as the property assertions of the subprocesses mapped to these activities. As implementation of P_i is unknown, we do not know how many activities of each type can occur in P_i. To generate a suitable number of activity instances, we use the following approach.

1. For each activity type $A^T_{i_l} \in T(PA(P_i))$ we generate as many instances as specified by the maximum count constraint for this activity type in $PA(P_i)$. If no maximum count constraint is defined for this activity type, we generate D number of activity instances, where D is the depth of the verification.
2. For each $A^T_j \in T(R)$ if $A^T_j \subseteq \bigcup_{A^T_{i_k} \in T(PA(P_i))} A^T_{i_k}$, meaning that activity type A^T_j is covered by types mentioned in $PA(P_i)$, then activities for this type have already be generated in the first step. If A^T_j type refers to a type that is not part of or is broader than types from $T(PA(P_i))$, then two strategies can be taken:
   (a) We assume a closed world model (if subprocess does not mention this type, then it does not have it) and generate no additional activities of this type.
   (b) Introduce an additional negotiation step, where each subprocess will be asked to provide information with respect to this type.
   (c) Assume open world model: as property assertions only represent part of the process functionality, it can happen that the subprocess contains activities relevant for the overall requirement that are not reflected in the property assertions. In this case we would generate D activities of type $A^T_j \setminus \bigcup_{A^T_{i_k} \in T(PA(P_i))} A^T_{i_k}$.

4.2 Verifying Process Requirement

Let process P contain sub-processes $P_1, ...P_n$. Let $PA(P_i) = PA_1(P_i), ..., PA_k(P_i)$ denote property assertion of sub-process P_i. Let $SA(P)$ denote structural assertions derived from the structure of process P, which include temporal as well as execution condition dependencies of $P_1, ..., P_n$. Let $Act(P_i)$ denote specifications of internal activities of P_i as described above, including activity state axioms and child-parent relations between P_i and specified internal activities. Then a requirement R on process P is fulfilled iff the following is fulfilled:

$$\Big(SA(P) \wedge \bigwedge_{i \in 1..n} Act(P_i) \wedge \big(P_i.\text{executed} > 0 \to \bigwedge_{PA_j \in PA(P_i)} PA_j\big)\Big) \to R \quad (4.1)$$

Including different activity states allows us to verify certain properties of fault behaviour, but requires additional assertions to be added if we want to restrict property verification to the non-faulty behaviour. For this purpose we need to specify that all of the process activities must reach state *executed* if they reach state *started*. This would force states $faulted$ and *terminated* to be set to -1, which reflects the non-faulty execution of this activity. To force verification of the non-faulty behaviour for any activity A_i, we add the following assertion:

$$A_i.\text{started} > 0 \to A_i.\text{executed} > 0 \quad (4.2)$$

If on the other hand faulty behaviour of certain activities should be taken in consideration during verification, assertion 4.2 should be skipped for such activities.

4.3 Application to the Case Study

In the motivating example we have four subprocesses modelled in the process graph, represented through the hypernodes, which represent abstractions of the corresponding processes. These are $Retailer_1$, *Reseller*, *Producer* and $Retailer_2$. To model these subprocesses we declare four variables $Retailer_1$, *Reseller*, *Producer* and $Retailer_2$ to be of type *Activity* and add the corresponding state relation axioms to our verification basis. The type of these activities has not been specified, therefore we do not restrict *atype* field of these variables. Our example model also contains one data variable *Quantity*, which is added to the verification basis as an *Integer* variable. Next we will show how the abstract processes are modelled using *Reseller* activity and corresponding property assertions as an example.

The *Reseller* does not specify activity occurrence restrictions, therefore to generate internal activities for this abstracted process we need to choose the depth parameter. For this example we choose verification depth $D = 2$ and use closed world assumption, which means that we will create two activities of each type declared by the process abstraction and create no additional activities. In our case we create two activities ED_1, ED_2 of type *ExpressDelivery* and two activities QC_1, QC_2 of type *QualityCheck*.

After activity declaration and addition of the activity axioms, we add parent-child axioms according to Table 3. An example axiom (H_1) applied to activities of type *ExpressDelivery* looks as follows:

$$ED_i.ready > 0 \rightarrow Reseller.\text{started} > 0 \wedge ED_i.\text{ready} \geq Reseller.\text{started}$$
$$ED_2.ready > 0 \rightarrow Reseller.\text{started} > 0 \wedge ED_2.\text{ready} \geq Reseller.\text{started}$$

Next we will specify property assertions of *Reseller* mapped to the generated internal activities. The first property assertion PA_1 of *Reseller* is *If (Quantity ¡ 5000) THEN ExpressDelivery*, which can formally be represented as $(Quantity < 5000) \rightarrow Count(ExpressDelivery, executed) > 0$. This assertion is mapped to the internal activities as follows:

$$PA_1 = (Quantity < 5000 \rightarrow (ED_1.\text{executed} > 0 \vee ED_2.\text{executed} > 0))$$

The second property assertion PA_2 of *Reseller* is *PrecededBy(ExpressDelivery, QualityCheck)* is equivalent to $\forall A \in ExpressDelivery : A.\text{started} > 0 \rightarrow (\exists B \in QualityCheck : B.\text{executed} > 0 \wedge B.\text{executed} < A.\text{started})$ To map the $\forall$ quantor to the internal activities, we need to add the corresponding assertions for each generated activity of type *ExpressDelivery*. In our case we map $PA_2^{ED_i}$, where $i \in [1, 2]$ denotes the index of the generated *ExpressDelivery* activity, to the generated process activities as follows:

$$PA_2^{ED_i} = \begin{array}{l} ED_i.\text{executed} > 0 \rightarrow \\ \big((QC_1.\text{executed} > 0 \wedge QC_1.\text{executed} < ED_i.\text{executed}) \\ \vee (QC_2.\text{executed} > 0 \wedge QC_2.\text{executed} < ED_i.\text{executed})\big) \end{array}$$

In the next step we add process structure axioms to the verification basis according to Section 2.3. For this we first define link variables of type *Link* for each process link:

$$Link := Record(source : Activity, target : Activity, condition : bool)$$

An example link between $Retailer_1$ and *Reseller* is defined as

$$L_1 : Link(source = Retailer_1, target = Reseller, condition = Quantity < 2000)$$

Next we apply axioms from Table 4 to each of the process activities and each of the process links. Here is an example of S_1, S_2 and S_5 axioms applied to the *Reseller*:

$$\begin{aligned} &Reseller.\text{ready} > 0 \rightarrow Reseller.\text{ready} \geq Retailer_1.\text{completed} \\ &\qquad\qquad\qquad\qquad\qquad \wedge\, Retailer_1.\text{completed} > 0 \\ &Retailer_1.\text{completed} > 0 \rightarrow Reseller.\text{ready} > 0 \\ &Reseller.\text{ready} > 0 \wedge Status(L_1) \leftrightarrow Reseller.\text{started} > 0 \end{aligned}$$

In addition, we specify that we only want to verify non-faulty runs through elimination of faulty behaviour using rule 4.2. Applied to *Producer* this rule looks as follows [2]:

$$Producer.\text{started} > 0 \rightarrow Producer.\text{executed} > 0$$

After modelling the process through the assertions according, we can prove the collaboration requirement R: *Product quality must be checked before the product is sold*, which is mapped to the generated activities as follows:

$$\begin{aligned} &SellProduct_1.\text{started} > 0 \rightarrow \\ &\quad \big((QC_1.\text{executed} > 0 \wedge SellProduct_1.\text{started} > QC_1.\text{executed}) \\ &\quad \vee (QC_2.\text{executed} > 0 \wedge SellProduct_1.\text{started} > QC_2.\text{executed}) \\ &\quad \vee (QC_3.\text{executed} > 0 \wedge SellProduct_1.\text{started} > QC_3.\text{executed})\big) \end{aligned}$$

Using SMT solver Z3 [13], we add the process model assertions and negation of the requirement to the verification context and check its satisfiability. The solver returns UNSAT, which means that the requirement R is fulfilled. If we slightly modify property assertions provided by the participants, e.g. change *Reseller* guarantee for express delivery for quantities under 1000, then the SMT solver returns SAT with a model that represents violation of the requirement. The model returned by SMT solver assigns a value between 1000 and 2000 to the quantity variable, and sets QC_1.executed, QC_2.executed, ED_1.executed and ED_2.executed to $false$.

[2] We can skip corresponding assertions for the internal activities as their fault or termination would lead to fault or termination of the parent activities, which is forbidden through the above assertions.

4.4 Performance Discussions

SMT solvers have been developed in academia and industry with increasing scope and performance [3]. However, as the problem is NP-hard, the verification of large models can still take quite a lot of time. The presented approach can naturally cope with the large models by abstracting parts of the process through the property assertions. Process abstraction through property assertions not only allows participants to hide their implementation details, but also reduces the size of the model by removing unnecessary details. To additionally improve performance we can further reduce the model size by decreasing the *Depth* parameter for generation of the internal activities (depth of 1 is sufficient in a lot of cases), or through simplification of the activity state model. The number of integer variables required to represent N activities is $N \times K + 1$, where K is the number of activity states (1 is for the activity type variable). We can further control the model size by deciding which activity states are required for each verification case. In this work we use 8 state variables for each activity. Based on the verification requirements this number can be reduced to 3 basic states: *ready*, *executed* and *skipped*, which would suffice to analyse *normal* behaviour of a process; or it can be extended with *repaired*, *cancelled* and any other relevant states if required.

5 Related Work

Constraints have been widely used in the business process area for process specification, annotation, verification and validation. Our process modelling approach is closely related to the $flow$ construct of the Business Process Execution Language (BPEL, [17]). An extensive overview of existing BPEL formalisations and verification approaches is provided in [4]. In [14] the constraints are used to model the semantic of a BPEL process. This allows verification of other constraints against the set of constraints representing a BPEL process by reducing the constraint verification problem to the constraint satisfiability problem. The approach analyses data flow together with the control flow, which allows verification of the data dependent properties. The data mapping approach presented in this paper was applied to the current work. In [15] an approach for modelling and validation of different constraint types based on the geometrical shape of a business process was presented. Property assertion language presented in this paper heavily influenced the property assertion language in the current work. A lot of work [9,1,10,11] exist in the area of business process verification using petri nets. These works concentrate on verification of a workflow where all the implementation details are known. In [19,18] authors present an LTL based constrain specification language, that is used to specify declarative workflows. In every step of a workflow execution a set of the reachable is computed, so that a user cannot execute an activity which does not lead to a successful process termination. While this work is related to the property specification language, our property assertion language allows to specify data-dependent properties, as well as activity deadlines. An approach based on abstract state machines (ASM) is presented in [6]. While the mapping covers scopes, it does not consider the relations between data conditions and activity executions. Approaches based on the π-calculus are presented in [5,12]. As with the ASM approaches, these do not consider the data dependencies of the process. In [7] the authors presented a formal approach for modeling and verification of web service composition using finite state process (FSP) notation. Similarly to the Petri Nets approach this approach does not cover analysis of the activity dependency on the process data and data manipulations.

In contrast to other works, we use proeprty assertions to declaratively describe parts of a process. The presenterd verification approach uses property assertions instead of the process models, which differentiates this work from the related work.

6 Summary and Future Work

In this paper we presented a hierarchical process verification approach that allows process participants to avoid disclosure of their subprocesses and still allow for verification of certain properties of the parent process. We showed how a hierarchical process model can be mapped to assertions, starting from a single activity and its state axioms, through abstracted subprocesses with their guarantees and child-parent axioms, to the complete process structure axioms. Using the gennerated assertions we showed how a requirement over the process composition can be verified using an SMT solver.

Acknowledgments. The research leading to these results has received funding from the German “Federal Ministry of Education and Research” in the context of the project “RescueIT”.

References

1. Van der Aalst, W.M.P.: Verification of Workflow Nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
2. Beckert, B., et al.: Intelligent Systems and Formal Methods in Software Engineering. IEEE Intelligent Systems 21(6), 71–81 (2006)
3. Biere, A., Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, Amsterdam (2009)
4. van Breugel, F., Koshkina, M.: Models and Verification of BPEL (2006), `http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf`
5. Fadlisyah, M.: Using the π-calculus for modeling and verifying processes on web services. Master's thesis, Insitute for Theoretical Computer Science, Dresden University of Technology (2004)
6. Fahland, D., Reisig, W.: ASM-based semantics for BPEL: The negative control flow. In: 12th International Workshop on Abstract State Machines, pp. 131–151 (March 2005)

7. Foster, H., Uchitel, S., Magee, J., Kramer, J.: A Model-Based Approach to Engineering Web Service Compositions and Choreography in Test and Analysis of Web Services. In: Baresi, L., Di Nitto, E. (eds.), ch. 71-91, pp. 72–91. Springer-Verlag Berlin and Heidelberg GmbH & Co. (2007)
8. Giblin, C., Liu, A.Y., Müller, S., Pfitzmann, B., Zhou, X.: Regulations expressed as logical models (realm). In: JURIX, pp. 37–48 (2005)
9. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri Nets. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 220–235. Springer, Heidelberg (2005)
10. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting BPEL Processes. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 17–32. Springer, Heidelberg (2006)
11. Lohmann, N., Massuthe, P., Wolf, K.: Behavioral Constraints for Services. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 271–287. Springer, Heidelberg (2007)
12. Lucchia, R., Mazzara, M.: A pi-calculus based semantics for ws-bpel. Journal of Logic and Algebraic Programming 70(1), 96–118 (2007)
13. Microsoft Research. Z3 an efficient theorem prover, `http://research.microsoft.com/en-us/um/redmond/projects/z3/`
14. Monakova, G., et al.: Verifying Business Rules Using an SMT Solver for BPEL Processes. In: BPSC (2009)
15. Monakova, G., Leymann, F.: Workflow art: A framework for multidimensional workflow analysis. In: Enterprise Information Systems (2012)
16. Nelson, G., Oppen, D.: Simplification by Cooperating Decision Procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
17. OASIS. Web Services Business Process Execution Language Version 2.0 (2007)
18. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-Based Workflow Models: Change Made Easy. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 77–94. Springer, Heidelberg (2007)
19. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 1–23. Springer, Heidelberg (2006)