Formalization of Workflows and Correctness Issues in the Presence of Concurrency *

İSMAİLCEM BUDAK ARPINAR, UĞUR HALICI, SENA ARPINAR, AND ASUMAN DOĞAÇ

{budak,nural,asuman}@srdc.metu.edu.tr
halici@rorqual.cc.metu.edu.tr

Software Research and Development Center
Department of Computer Engineering
Middle East Technical University (METU)
06531, Ankara, Turkiye

Abstract. In this paper, main components of a workflow system that are relevant to the correctness in the presence of concurrency are formalized based on set theory and graph theory. The formalization which constitutes the theoretical basis of the correctness criterion provided can be summarized as follows:

- Activities of a workflow are represented through a notation based on set theory to make it possible to formalize the conceptual grouping of activities.
- Control-flow is represented as a special graph based on this set definition, and it includes serial composition, parallel composition, conditional branching, and nesting of individual activities and conceptual activities themselves.
- Data-flow is represented as a directed acyclic graph in conformance with the control-flow graph.

The formalization of correctness of concurrently executing workflow instances is based on this framework by defining two categories of constraints on the workflow environment with which the workflow instances and their activities interact. These categories are:

- Basic constraints that specify the correct states of a workflow environment.
- Inter-activity constraints that define the semantic dependencies among activities such as an activity requiring the validity of a constraint that is set or verified by a preceding activity.

Basic constraints graph and inter-activity constraints graph which are in conformance with the control-flow and data-flow graphs are then defined to represent these constraints. These graphs are used in formalizing the intervals among activities where an inter-activity constraint should be maintained and the intervals where a basic constraint remains invalid.

A correctness criterion is defined for an interleaved execution of workflow instances using the constraints graphs. A concurrency control mechanism, namely Constraint Based Concurrency Control technique is developed based on the correctness criterion. The performance analysis shows the superiority of the proposed technique. Other possible approaches to the problem are also presented.

Keywords: Workflow Management System, Workflow, Activity, Basic Constraint, Inter-activity Constraint, Time Intervals, Correctness, Concurrency Control.

* This work is partially being supported by the Middle East Technical University, the Graduate School of Natural and Applied Sciences, Project No: AFP-97-07-02-08, and by the Scientific and Technical Research Council of Turkey, Project No: 197E038.

1. Introduction

Today, economic imperatives are forcing enterprises to look for new information technologies to streamline their business processes. Key requirements include integrating heterogeneous information resources of an enterprise, and automating mission-critical applications that access shared information resources. Many of the activities in these enterprises are of long-duration and consist of multiple operations executed over (possibly) heterogeneous systems with very diverse response times. As a consequence of these trends, Workflow Management Systems (WFMSs) are quickly becoming the technology of choice to implement large and heterogeneous distributed execution environments where sets of interrelated activities can be carried out in an efficient and closely supervised fashion [4]. There is also a standardization effort in this respect. The Workflow Management Coalition (WfMC), an industry consortium aims at a unified terminology and a standardization of key components of a workflow management system. The WfMC identified a set of six primitives with which it is possible to describe control-flow and hence construct a workflow specification [39].

A *workflow* is defined as a collection of processing steps (activities) organized to accomplish some business processes. An activity can be performed by one or more software systems or machines (e.g., instruments or robots), by a person or a team, or a combination of these. In addition to collection of activities, a workflow defines the order of activity invocations or condition(s) under which activities must be invoked (i.e., control-flow) and data-flow between these activities. Activities within a workflow can themselves again be a workflow. In general a workflow activity is considered to be an invocation of a local operation which is functional in nature. Furthermore, an activity may be further composed of several calls to local systems (such as in multidatabases [30, 34]), and this fact is hidden at the workflow level.

The activities could be *transactional* or *non-transactional*. Transactional activities are those that access data controlled by Resource Managers (RMs) with transactional properties (i.e., ACID). These activities minimally support the atomicity property and maximally support all ACID properties of traditional transaction models [46]. These activities typically include those that interact with a DBMS by using *Commit* and *Abort* operations, stored procedures, and two-phase commit (2PC) activities. In addition, activities that use the XA-Protocol [33] based Remote Procedure Call (RPC) to communicate with transactional processing entities such as a TP-Monitor [16, 15] in a distributed environment can also be included in this category.

Non-transactional activities access data controlled by RMs without transactional properties. These non-transactional processing entities include file systems, humans, legacy systems, HTTP servers, word processors, and spreadsheets. Yet, it may be possible to introduce transactional properties to these systems, for example by wrapping non-transactional RMs to provide transaction and concurrency control services according to OMG's Object Transaction Service (OTS) [50] and Concurrency Control Service (CCS) specifications.

1.1. Correctness Issues in WFMSs

As discussed briefly in [29], the person who implements an activity is responsible for ensuring that the activity produces correct results if it is executed alone. However since workflows are long running processes, having the activities terminate (e.g., commit) within the scope of a workflow instance is an accepted practice. Thus the data modified by these activities becomes accessible to the other activities within the same workflow instance as well as to the other workflow instances which may cause inconsistencies due to improper interleavings. Yet many scenarios in the operation of a workflow system require the preservation of consistency of at least some data items. Therefore the workflow execution must address the following two correctness concerns: (i) The consistency of concurrent executions of activities belonging to the same workflow instance; (ii) the consistency of concurrent executions of activities belonging to different workflow instances.

For example consider an *Order Processing* workflow in a manufacturing enterprise. In the processing of the *Order Processing* workflow, raw material stock is checked through a *CheckStock* activity to see whether there is enough raw material in the stock to process the order. If not, the missing raw materials are ordered from external vendors and inserted into stock through an *InsertStock* activity. Yet later in the process when the actual manufacturing is to start for this workflow instance there may not be enough raw material in the stock to process this order, because a concurrently running instance of the same or other workflows might have updated the stock. Of course, executing all these activities within the scope of a single transaction might have solved these problems but workflow systems are there to prevent the inefficiency of long-running transactions [32].

Another example to the data inconsistency problems is as follows: Consider the *Withdraw-Deposit* workflow of a bank involving two branches. *Withdraw* activity withdraws the given amount of money from an account at a branch, and the *Deposit* adds this amount to an account at another branch. Let us consider an *Audit* workflow which checks the balance of these accounts. If *Withdraw-Deposit* activities and activities of the *Audit* workflow are interleaved incorrectly *Audit* misses the money being transferred between the two accounts.

The current state of the art for workflows lacks a clear theoretical basis, correctness criteria and support for consistency of concurrent workflows to handle such problems [63]. In this paper exactly these issues are addressed. We provide a theoretical basis for the formalization of workflows, and define a correctness criterion for the consistency of concurrently executing workflows based on this formalization, and present a concurrency control technique to provide the correctness.

The main contributions of the paper are as follows:

(1) A workflow in conformance with the control-flow primitives of WfMC model is formalized based on set theory and graph theory.

We start by defining a special set whose elements may also be sets, called a nested hyperSet, and use this set in representing the conceptual groupings of activities in a workflow system. The control-flow is imposed on this set by introducing the

related edges and the resulting graph is called hyperNodeGraph. Split and join nodes are introduced into this graph from where control-flow splits into multiple branches and merges into a single flow later respectively. Data-flow in a workflow is represented through a simple directed acyclic graph which is in conformance with the control-flow graph. Having thus set the necessary background, we provide a formal definition of a workflow.

(2) This formalization is used in defining a correctness criterion for concurrently executing workflows based on the semantic information available.

Workflow activities access resources which denote the set of all objects constituting the workflow environment. We define correct execution of activities in terms of their input and output conditions, which are the sets of constraints on the workflow environment. An input condition may involve two types of constraints: basic constraints that specify the correct states of a workflow environment and inter-activity constraints that define the semantic dependencies between activities, such as an activity requiring the validity of a constraint that is set or verified by a preceding activity. For example a basic constraint can state that the money being transferred between two branches of a bank through a *Withdraw-Deposit* workflow should not be destroyed during this transfer. This basic constraint remains invalid between the executions of *Withdraw* and *Deposit* activities for obvious reasons. Furthermore, consider *InsertStock* activity in the manufacturing example. Since the resulting amount of raw materials after the termination of *InsertStock* must remain in the stock until the beginning of manufacturing process that ordered it, this requirement is represented as an inter-activity constraint between *InsertStock* and the activity which is responsible from actual manufacturing process.

The intervals among activities where an inter-activity constraint should be maintained and the intervals where a basic constraint remains invalid are formalized through the graphs corresponding to these constraints. These graphs are then used in developing a correctness criterion for interleaved execution of workflows which is formally represented through a complete execution history. Simply stated, the correctness criterion requires two conditions to hold:

i. The inter-activity constraints should be preserved in the related intervals by preventing the activities that invalidate these constraints from executing.

ii. The activities that require the correctness of related basic constraints should be prevented from executing during the intervals where these constraints do not hold.

(3) A correctness technique, namely Constraint Based Concurrency Control (CBCC) technique, is developed based on this correctness criterion.

CBCC technique which is based on locking in conjunction with validation, controls activity interleavings in such a way that two conditions above hold. The inter-activity constraints are locked during the time interval where they should remain valid in the shared mode. An activity that falsifies these constraints acquire a lock in the conflicting mode (i.e., exclusive mode). Through these conflicting locks activities that falsify inter-activity constraints are prevented from executing. If

more than one activity require the same inter-activity constraint to be true at the same time, their locks do not conflict. Similarly, activities that falsify the same constraint at the same time do not conflict either. Note that we use the term "exclusive lock" differently than its conventional meaning in that, two exclusive locks on the same constraint do not conflict with each other in our approach.

Some activities on the other hand *may* falsify inter-activity constraints depending on the instantiation of the variables in the constraints and in their parameters. For the activities that *may* falsify inter-activity constraints, we prefer to use an optimistic scheme rather than locking with the intention of increasing the performance, since there is a probability that the activity will not falsify these constraints. If these constraints evaluate to true at the end of an activity, the activity is allowed to terminate, otherwise it is aborted and resubmitted. Continuing with the example provided, since raw materials may be withdrawn from the stock by the concurrently executing *WithdrawFromStock* activities of some other workflows, the inter-activity constraint between *InsertStock* and the manufacturing activity may be invalidated. To prevent this, *InsertStock* obtains a shared lock on this constraint which will be released by the manufacturing activity and if a *WithdrawFromStock* activity is executed between them it goes through a validation phase.

However, it is also possible to use a more conservative approach in which activities acquire locks on the inter-activity constraints they *may* falsify in addition to the constraints they certainly falsify. We call this conservative technique based solely on locking as Constraint Locking Concurrency Control (CLCC) technique. For example, *WithdrawFromStock* activity can obtain an exclusive lock on the inter-activity constraint in CLCC technique instead of going through a validation phase.

The basic constraints specify the correct states of a workflow environment but they can be invalidated by an activity to be revalidated later through an activity or through a set of activities. The activities that require the validity of these basic constraints should not be allowed to execute in the interval where the basic constraints remain invalid, and for this purpose exclusive locks are placed on the basic constraints during these intervals by the activities that falsify these constraints. On the other hand, the activities that require the validity of the basic constraints acquire locks in the conflicting mode (shared mode). For example, *Withdraw* activity obtains an exclusive lock on the basic constraint which it falsifies, and this lock is released after *Deposit* activity terminates. Since and *Audit* activity needs a shared lock on the same constraint, its execution is prevented between *Withdraw* and *Deposit* activities. The shared locks of activities which require correctness of the same basic constraint at the same time do not conflict with each other, and the same is true for the exclusive locks of activities which falsify the same basic constraint at the same time.

(4) A performance analysis of the CBCC and CLCC techniques is presented.

A performance comparison of the proposed techniques with some other approaches to the problem is also presented. The performance results indicate that our techniques result in better performance than the other techniques.

In the work presented in this paper, semantic information about activities and workflow environment is used. In the case where this semantic information is not available, activities should be treated as black boxes and since isolation of a whole workflow execution is unacceptable because of performance reasons, smaller units of isolation should be discovered. The individual activities of a workflow are isolated by concurrency control mechanisms of local systems, and hence the main concern is to observe the concurrency control requirements between these individual activities and satisfy these requirements when required. These requirements may be determined by checking the data and control-flow dependencies between the activities. These dependencies are available at design-time, and therefore *spheres of isolations* each of which includes a subset of activities of a workflow can be determined in advance and correctness of workflows can be guaranteed through the isolation of these spheres. The approaches that use this idea [7, 53, 58] are explained in Section 2. It should be noted that these approaches are much more restrictive compared to the techniques presented in this paper which make use of semantic information.

After setting the research context in the first section, the paper is organized as follows: In Section 2, the related work is given. In Section 3 we present a motivating example to explain main concepts of our approach and identify the general workflow features covered by our model. Section 4 provides formal characterization of workflows in terms of data and control-flow dependencies. Section 5 defines correctness of concurrently executing workflows and activities. In Section 6, concurrency control techniques based on this correctness definition are proposed, and the performance analysis of the techniques is given. Section 7 gives concluding remarks.

2. Related Work

Although some research has been done on the correctness problem of workflows, neither a widely accepted correctness notion nor a correctness mechanism have been reported in the literature. In the following, we confine ourselves to summarizing the related research in workflow management systems and transaction processing systems. And in spite of this research, most commercial WFMSs provide very limited capabilities for correctness and concurrency control issues [56].

In the ConTract model [55, 60], the user is given the sole responsibility for maintaining the consistency of the database with which activities interact. In order for activities to work correctly, predicates named as entry and exit invariants are defined to hold on the database. At run-time, these predicates are verified before and after an activity respectively. If exit or entry invariant evaluate to false, a conflict resolution algorithm is executed and this may involve changing values of objects in the predicates in such a way that they are satisfied. However, an inevitable result may be cancellation of activity and compensation of some previously terminated activities.

In [13] to ensure data consistency, semantic serializability of workflows is proposed as the correctness criterion. A human expert declares a compatibility matrix for

activities of a workflow. Compatibility of two activities means that the ordering of two activities in an execution history is insignificant from an application point of view. If two activities are not defined as compatible they are in conflict. An execution history is semantically serializable if an equivalent serial execution exists with the same ordering of conflicting activities.

In [3], the consistency is specified in the same way as the compatibility relationships are expressed with the added complexity of having to express compatibility relations between sequences of activities instead of between individual activities. For instance, how different workflow instances should be interleaved in the system is given as a matrix. The main idea is based on signatures of workflow instances that they leave on the objects they access. This signature specifies which other workflows are allowed to access the object.

In Transaction Specification and Management Environment (TSME) [28] using a transaction specification language, correctness as well as state dependencies can be specified between the activities of workflows. Different correctness dependencies such as *serializability, temporal*, and *cooperative dependencies* can be specified. To define conflicts, each object is associated with a conflict table. *Serialization dependencies* are specified as acyclic serialization order dependencies between activities. *Temporal order dependencies* are specified by giving specific serialization orders between the activities. *Cooperation* between activities is provided by using breakpoints or augmenting conflict tables of shared objects.

In [7], activities are treated as black boxes and to determine concurrency control requirements between activities, data and control-flow dependencies between them are analyzed at design-time. Using this information *spheres of isolation*, each of which involves a subset of activities in a workflow, are determined and the notion of correctness is based on the isolation of these spheres. Furthermore, a technique to handle correctness of hierarchically structured workflows consisting of compound activities is proposed in [7]. In [53, 54], *M-serializability* is defined as a correctness criterion for concurrent execution of workflows. In this model, related activities of a workflow are grouped into *execution-atomic units. M-serializability* assumes that an activity involves a single site and it requires that activities belonging to the same *execution-atomic unit* of a workflow have compatible serialization orders at all sites they access. A similar approach is proposed in [58]. In this work, a set of activities are grouped into a *consistency unit* and traditional correctness techniques are used to provide serializable execution of this unit.

2.1. Semantics Based Concurrency Control

Although semantics based concurrency control mechanisms do not directly cover workflow correctness, they are related to the approach proposed in this paper. Semantics based concurrency control protocols can be broadly classified into three categories depending on whether they are based upon the semantics of transactions or upon the semantics of objects or both as described in [1]: Approaches of Gray [32], Garcia-Molina [26], Lynch [47], Weikum [61], Beeri [10], Farrag and Ozsu [24]

```
DEFINE_PROCESS OrderProcessing()
  ...
  GetOrder(OUT productNo, OUT quantity, OUT dueDate, OUT orderNo,
           OUT customerInfo)
  EnterOrderInfo(IN productNo, IN quantity, IN dueDate, IN orderNo)
  CheckBillofMaterial(IN productNo, OUT partList)
  PAR_AND (part = FOR EACH partList)
    SERIAL
       DetermineRawMaterial(IN part.No, IN part.Quantity, OUT rawMaterial,
                            OUT required)
       CheckStock(IN rawMaterial, IN required, OUT missing)
       IF (missing > 0) THEN
         VendorOrder(IN rawMaterial, IN missing)
       WithdrawFromStock(IN rawMaterial, IN required)
       GetProcessPlan(IN part.No, OUT processPlan, OUT noofSteps)
       i:=0
       WHILE (i < noofSteps)
         Assign(IN processPlan[i].cellId, IN orderNo, IN part.No,
                IN part.Quantity, IN rawMaterial, IN required)
       END_WHILE
    END_SERIAL
  END_PAR_AND
  AssembleProduct(IN productNo)
  ...
  Billing(IN orderNo, IN productNo, IN quantity, IN customerInfo)
  ...
END_PROCESS
```

Figure 1. Order Processing Example

can be classified into first category; works of Harder [35], O'Neil [49], Korth and Speegle [45], Herlihy [37], Badrinath and Ramamritham [9] mainly fall into second category. The works in the third category use the advantages of both approaches to increase concurrency. In [1], three semantics based correctness criteria are proposed. In [5] and [12], formal methods to decompose a transaction into smaller units using transaction and object semantics are described. In [5], the notion of semantic histories and successor sets are proposed to describe correct interleavings of these units (i.e., steps). In [12], transaction semantics are used to decompose transactions into steps and a concurrency control technique is described to control step interleavings in such a way that assertions between the consecutive steps are preserved.

3. A Motivating Example

In this section, an order processing example in a highly automated manufacturing enterprise is provided using the workflow definition language of METUflow [7, 18, 31, 42, 43].

```
DEFINE_PROCESS VendorOrder(IN rawMaterial, IN missing)
   ...
   SendOrder(IN rawMaterial, IN missing, OUT shipmentNo)
   SuppliesArrival(IN shipmentNo)
   InsertStock(IN rawMaterial, IN missing)
END_PROCESS

DEFINE_PROCESS GetProcessPlan(IN part.No, OUT processPlan, OUT noofSteps)
   ...
   DetermineNoofCells(IN partNo, OUT cellNo)
   SelectBestCells(IN cellNo, OUT qualifiedCells)
   ConstructProcessPlan(IN qualifiedCells, OUT processPlan, OUT noofSteps)
END_PROCESS

DEFINE_PROCESS Billing(IN orderNo, IN productNo, IN quantity, IN customerInfo)
   ...
   Payment(IN orderNo, IN productNo, IN quantity, IN customerInfo, OUT amount,
            OUT paymentStatus)
   IF (paymentStatus = unpaid) THEN
      UpdateUnpaidBalance(IN customerInfo, IN amount, OUT unpaidBalance, OUT U)
      IF(unpaidBalance > U) THEN
         XOR
           RejectShipping(IN orderNo)
           MoreCredit(IN customerInfo, IN unpaidBalance, IN U)
         END_XOR
   END_IF
END_PROCESS
```

Figure 2. Order Processing Example (Cont.)

An incoming customer request causes a product order to be created and inserted into an order entry database by *GetOrder* and *EnterOrderInfo* activities respectively (Figure 1). The next step is to determine required parts to assemble the ordered product by *CheckBillofMaterial* activity. A part is the physical object which is fabricated in the manufacturing system. For each part, *DetermineRawMaterial* activity is executed to find out the raw materials required to manufacture that part, and a *CheckStock* activity is initiated afterwards to check stock database for the availability of these raw materials. If the required amounts of these raw materials do not exist in the stock, they should be ordered from the external vendors through *VendorOrder* (Figure 2). After all missing raw materials are obtained, required raw materials to fabricate the part is withdrawn from the stock to be sent to the manufacturing cells. This is accomplished by *WithdrawFromStock* activity by decrementing the available amount of the withdrawn raw material (i.e., $quantity(m)$) in the stock database. The required steps to manufacture a part, and the manufacturing cells where these steps are performed are obtained as a result of *GetProcessPlan*. Actual manufacturing activity is initiated by assigning the work to the corresponding cells for each step in *Assign*. Finally, manufactured

```
DEFINE_PROCESS WarehouseAllocation()
   ...
   GetAllocationOrder(OUT rawMaterial, OUT quantity, OUT source, OUT destList)
   RetrieveMaterial(IN rawMaterial, IN quantity, IN source)
   PAR_AND (destination = FOR EACH destList)
      UpdateMaterialLocation(IN rawMaterial, IN quantity, IN destination)
   END_PAR_AND
END_PROCESS

DEFINE_PROCESS StockControl(IN stockDBList)
   ...
   WarehouseEvaluation(IN stockDBList, OUT materialSum)
   PrintMaterialReport(IN materialSum)
END_PROCESS
```

Figure 3. WarehouseAllocation and StockControl Workflows

parts are assembled to form the product that the customer had ordered by the activity *AssembleProduct*. Further downstream activities include a billing activity. *Billing* itself is another workflow which is responsible from collecting bills of ordered products. The details of *Billing* workflow is explained in Section 5.

We further consider two other workflows defined in the system (Figure 3): *WarehouseAllocation* and *StockControl*. *WarehouseAllocation* distributes raw materials among different warehouses and reallocates the materials according to demand and delivery schedules. *RetrieveMaterial* retrieves the given amount of raw material from the stock of the source warehouse and *UpdateMaterialLocation* transfers these raw materials to the stocks of the destination warehouses in *destList*. *StockControl* workflow checks the available raw materials of different types in stocks of all warehouses through *WarehouseEvaluation* activity and prints a stock report.

4. Formal Characterization of Workflows

In this paper, we first attempt to formalize the correctness issues of workflow systems in the presence of concurrency and then provide a correctness technique based on the theory developed. In order to formalize the correctness issues, we first formalize the related concepts of workflows.

Currently, specification of workflows is realized through the following types of methods [48]: Script languages, net-based methods, logic-based methods, algebraic methods, and event-condition-action (ECA) rules. Most script languages and net-based methods lack a formally founded semantics. The notable exceptions are state charts [36, 62] and Petri nets [27, 20]. For a logic-based specification, temporal logic is a commonly used method [23], e.g., computational tree logic (CTL) is used to define control-flow dependencies [8]. Similarly, ECA rules are used to specify control-flow (e.g., [41]). As a final remark, many of these methods do not have

either a solid formal foundation or are often not intuitive and hard to understand. Thus, a formal yet simple formalization of workflows is needed.

A workflow in the most general sense describes groupings of activities that are executed sequentially or in parallel and defines data that may be exchanged between these activities. In formalizing a workflow, we define special graphs to express this data and control-flow information. We first define a hyperSet which represents the groupings of activities in a workflow and constitutes the basis of the graph to define the control-flow. In order to introduce control-flow relations between activities, edges are introduced into a hyperSet and then a graph which is named as a hyperNodeGraph is obtained. Data-flow between the activities is represented through a simple directed acyclic graph (DAG). Since control-flow and data-flow should be in conformance with each other, consistency relation between the graphs that represent them is defined. In our model, control-flow is not permitted to contain cycles, therefore a hyperNodeGraph is refined to a hyperNodeDAG. In addition, in order to define activities from where control-flow splits into multiple branches and merges into a single flow later, split and join nodes are introduced into a hyperNodeDAG, resulting in a split-join hyperNodeDAG.

Notice that, building the required properties of workflows through graphs in a top-down fashion with starting with the most general graph and refining it to include further properties of workflows, provides a formal and clear definition of a workflow. The solid mathematical and graph theory based foundation of this formalization make it appropriate for developing a correctness theory and a favorable reference model. It should be noted that, the primitives defined by Workflow Management Coalition (WfMC) [39] are taken into consideration in our model.

In the following, definition of a hyperSet that reflects the groupings of activities is provided. These groupings of activities are called as execution blocks or conceptual activities. When proper control-flow edges are imposed on this set, the resulting graph shows the execution structure of the workflow process.

Definition 4.1 [HyperSet] A *hyperSet* S is a set whose elements are simple elements or hyperelements which are simple sets or hyperSets. □

Notation: The notation $S_i \in S$ is used to denote that S_i is an element of S; the notation $S(\varepsilon_i)$ is used to denote the element ε_i of S; *size(S)* is used to denote the number of elements in S; *simple(S)* and *hyper(S)* are used to denote the set of simple elements of S and the set of hyperelements of S respectively. S_i, which may be a simple element or a hyperelement, is a *subelement* of a hyperSet S, denoted as $S_i \underline{\in} S$, iff $S_i \in S$ or $S_i \underline{\in} S_j$ for some $S_j \in S$. The notation $\varepsilon_{(i_1,i_2,\ldots,i_{k-1},i_k)}$ is used to denote a subelement which satisfies $\varepsilon_{(i_1,i_2,\ldots,i_{k-1},i_k)} \in \varepsilon_{(i_1,i_2,\ldots,i_{k-1})} \in \ldots \in \varepsilon_{(i_1,i_2)} \in \varepsilon_{i_1} \in S$. We shall drop parentheses and comas between indexes when it is clear in the notation. The *level of set* S is zero; the elements $S_i \underline{\in} S$ are called *level k elements* for which the parent is *level k-1* element. The set of base elements of hyperSet S, denoted as *base(S)*, is a flat set which contains all the simple subelements of S. A hyperSet S is a *flat set* if it has no hyperelement, that is any $S_i \in S$ is a simple element.

Observe that elements in a hyperSet are not disjoint. In a workflow system however, each instantiation of the same activity type should be treated as a new element at each invocation (e.g., with different set of parameter values). Furthermore, participation of the same activity instance to more than one execution block is similar to improper nesting of blocks in a procedural language. For these reasons, a *nested hyperSet* with disjoint elements is defined, and it constitutes the nodes of the hyperGraphs to be defined for representing different components of a workflow.

Definition 4.2 [**Nested HyperSet**] A hyperSet is *nested* if $base(S_i) \cap base(S_j) = \emptyset$ for any $S_i, S_j \subseteq S$. □

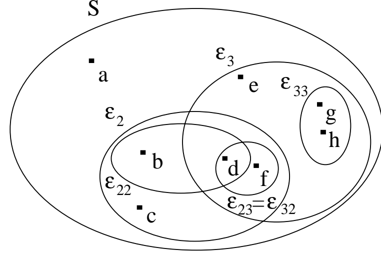


Figure 4. A HyperSet

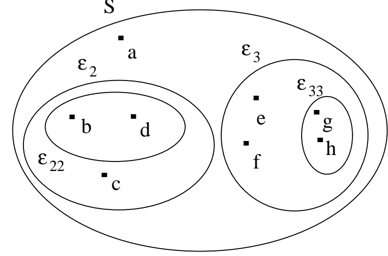


Figure 5. A Nested HyperSet

Example 4.1 Let $S = \{a, \{c, \{b,d\}, \{d,f\}\}, \{e, \{d,f\}, \{g,h\}\}\}$; elements of S are $\varepsilon_1 = a$, $\varepsilon_2 = \{c, \{b,d\}, \{d,f\}\}$, $\varepsilon_3 = \{e, \{d,f\}, \{g,h\}\}$; subelements of S are $\varepsilon_{21} = c$, $\varepsilon_{22} = \{b,d\}$, $\varepsilon_{23} = \{d,f\}$, $\varepsilon_{31} = e$, $\varepsilon_{32} = \{d,f\}$, $\varepsilon_{33} = \{g,h\}$, $\varepsilon_{221} = b$, $\varepsilon_{222} = d$, $\varepsilon_{231} = d$, $\varepsilon_{232} = f$, $\varepsilon_{321} = d$, $\varepsilon_{322} = f$, $\varepsilon_{331} = g$, $\varepsilon_{332} = h$ in addition to ε_1, ε_2, ε_3; $base(S) = \{a,b,c,d,e,f,g,h\}$; $simple(S) = \varepsilon_1$, $hyper(S) = \{\varepsilon_2, \varepsilon_3\}$; $size(S) = 3$; $size(base(S)) = 8$. Figure 4 shows this hyperSet. $S = \{a, \{c, \{b,d\}\}, \{e, f, \{g,h\}\}\}$ is a nested hyperSet which is depicted in Figure 5. □

Having defined a nested hyperSet which represents individual and conceptual activities, we can now define other components of a workflow. In the definition of a workflow we use four different graphs, namely a control-flow graph, a data-flow graph, and two constraints graphs. In a control-flow graph, precedence relations between individual and conceptual activities are provided, e.g., if an activity should be started after the termination of another activity this is represented by a directed edge from the former activity to the latter activity in the control-flow graph. In order to represent these control-flow dependencies, we introduce edges into a nested hyperSet and thus obtain a graph which we call as a hyperNodeGraph.

Data-flow between individual activities occurs if output parameter of an activity is involved in the input parameter of a successor activity in the control-flow. Data-flow is represented through a simple directed acyclic graph (DAG) in the formalization.

In Section 5, we develop a theory in which an input condition for an activity to execute correctly is specified in terms of constraints on the workflow environment with which a workflow and its constituting activities interact. The intervals among activities where a constraint should be maintained and intervals where a

constraint (may) remains invalid along a workflow execution are formalized using two constraints graphs which are 2-level hyperGraphs. Although construction and usage of a 2-level hyperGraph are explained in detail in Section 5, its definition is provided here for the sake of completeness. Furthermore, to keep the formalization at a general level we also provide the definition of a hyperGraph.

Definition 4.3 [HyperGraph, HyperNodeGraph, 2-level HyperGraph] A *hyperGraph* $G = (S, E)$ is a directed graph in which S is a hyperSet and edges E are defined on $S \times S \cup \{S_a \times S_a\}$ for any $S_a \Subset S$. Notice that the graph itself can be thought as a node at an abstract level. Any $S_a \in S$ is called a *node* and $S_a \Subset S$ is called a *subnode*. A *hyperNodeGraph* is a hyperGraph $G = (S, E)$, where S is a nested hyperSet. A *2-level hyperGraph* $G = (S, E)$ is a hyperGraph, where any $S_a \in S$ satisfies $S_a \subseteq base(S)$. □

In the following, these definitions are clarified through examples.

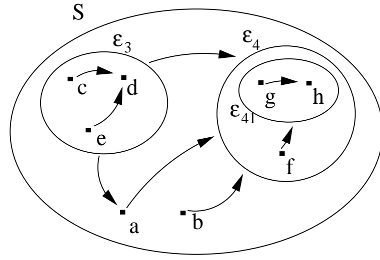


Figure 6. A HyperNodeGraph

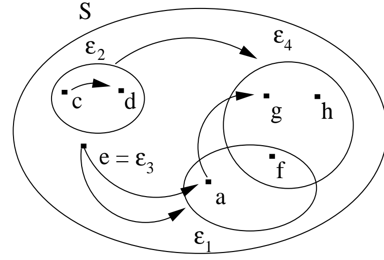


Figure 7. A 2-level HyperGraph

Example 4.2 Let $G = (S, E)$ be a hyperNodeGraph, where $S = \{a, b, \{c, d, e\}, \{\{g, h\}, f\}\}$ is a nested hyperSet and $E = \{\langle\varepsilon_3, \varepsilon_1\rangle, \langle\varepsilon_1, \varepsilon_4\rangle, \langle\varepsilon_3, \varepsilon_4\rangle, \langle\varepsilon_2, \varepsilon_4\rangle, \langle\varepsilon_{31}, \varepsilon_{32}\rangle, \langle\varepsilon_{33}, \varepsilon_{32}\rangle, \langle\varepsilon_{42}, \varepsilon_{41}\rangle, \langle\varepsilon_{411}, \varepsilon_{412}\rangle\}$. Figure 6 demonstrates this hyperNodeGraph.

Let $G = (S, E)$ be a 2-level hyperGraph, where $S = \{\{a, f\}, \{c, d\}, e, \{g, h, f\}\}$, and $E = \{\langle\varepsilon_2, \varepsilon_4\rangle, \langle\varepsilon_3, \varepsilon_1\rangle, \langle\varepsilon_3, \varepsilon_{11}\rangle, \langle\varepsilon_{11}, \varepsilon_{41}\rangle\}$. This graph is shown in Figure 7. □

Observe that the difference between a hyperGraph and hyperNodeGraph is that a nested hyperSet constitutes the nodes of a hyperNodeGraph. Therefore, only edges between the simple or hyperelements at the same level are possible. In this way, when we use a hyperNodeGraph to specify control-flow, anomalies in precedence relations are prevented. For example, if control splits into several flows and these flows are joined together within a hyperNode, control-flow can not jump into the middle of such flows from outside of this hyperNode.

Notice that level of elements in S is not greater than 2 in a 2-level hyperGraph $G = (S, E)$, i.e., level of S is 0, level of a $S_i \in S$ is 1, and level of a $S_j \in S_i$ is 2.

In a workflow, data-flow should be in conformance with its control-flow, that is, there can be data-flow between two activities only when there is a control-flow between them. Therefore a directed acyclic graph (DAG) which represents data-flow should be consistent with the hyperNodeGraph which represents corresponding control-flow. Informally, a DAG is said to be consistent with a hyperNodeGraph iff

for any edge between the two nodes of a DAG, there corresponds an edge between the same nodes or hyperNodes that include them in the transitive closure of the hyperNodeGraph. Transitive closure of a hyperNodeGraph $G = (S, E)$, denoted as $G^* = (S, E^*)$, can be obtained by taking transitive closure of the edges within every hyperNode of the graph. A more formal definition can be found in [6].

Furthermore, a 2-level hyperGraph which represents constraints graphs of a workflow should be consistent with its control-flow. The reason behind this requirement is explained in Section 5.

Definition 4.4 [Consistency with a HyperNodeGraph] A DAG $D = (T, V)$ is said to be *consistent* with a hyperNodeGraph $G = (S, E)$ iff the following condition is satisfied:

- For any $\langle T_a, T_b \rangle \in V$, $\exists \langle S_i, S_j \rangle \in E^*$, where $S_i = T_a$ or $S_i = T_A$ such that $T_a \Subset T_A \Subset S$, and $S_j = T_b$ or $S_j = T_B$ such that $T_b \Subset T_B \Subset S$.

A 2-level hyperGraph $D = (T, V)$ is said to be *consistent* with a hyperNodeGraph $G = (S, E)$ iff for any $\langle T_k, T_l \rangle \in V$ the following condition is satisfied:

- For any $T_a \in T_k$ and $T_b \in T_l$, $\exists \langle S_i, S_j \rangle \in E^*$, where $S_i = T_a$ or $S_i = T_A$ such that $T_a \Subset T_A \Subset S$, and $S_j = T_b$ or $S_j = T_B$ such that $T_b \Subset T_B \Subset S$. □

In the following, we introduce various useful operations on a nested hyperSet and a hyperNodeGraph. With these operations it becomes possible to focus on a hyperNode representing an execution block in a control-flow and conversely simplify it when its internals are not in the scope of our consideration.

A *restriction* of a hyperNodeGraph $G = (S, E)$ to one of its subelements $S_a \Subset S$, denoted as $G(S_a)$, results in a new hyperNodeGraph which involves the node itself, its constituting simple and hyperNodes if they exist and edges between them. The other nodes and edges in the hyperNodeGraph are omitted. Figure 8 depicts the restriction of hyperNodeGraph in Figure 6 to node ε_4.

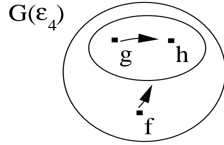


Figure 8. Restriction of the HyperNodeGraph in Figure 6 to Node ε_4

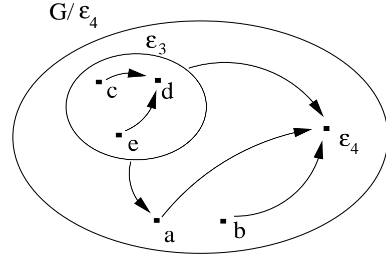


Figure 9. The Abstraction of Node ε_4 in HyperNodeGraph of Figure 6

Abstraction of a subelement S_a in a nested hyperSet S, denoted as S/S_a, is the replacement of S_a with an abstract simple element s_a in S. Let $S = \{a, b, \{c, d, \{e, f\}\}, \{g, h\}\}$. Abstraction of $S_3 = \{c, d, \{e, f\}\}$ in S results in $S/S_3 = \{a, b, s_3, \{g, h\}\}$, where s_3 is representing S_3.

An *abstraction* of a node S_a in a hyperNodeGraph G results in a new graph G/S_a, in which the node under consideration is replaced with a simple node and every

edge involving the former node is replaced with a new edge involving the simple node. Figure 9 shows the abstraction of node ε_4 in hyperNodeGraph of Figure 6.

Some workflow models assume that, all the structural components (i.e., control-flow) can be specified in advance. However, in some workflow applications either the number of activities in a workflow execution or the control-flow dependencies that must be enforced can not be determined in advance. These cases are named as *domain uncertainty* and *structural uncertainty* respectively [59]. Structural uncertainty occurs due to the fact that a workflow specification can contain a condition to allow selections. Our formalization covers this type of uncertainty and this is explained later in this section. Domain uncertainty occurs due to the loops (i.e., iterations) that can occur in a workflow specification. Within a loop workflow activities are repeated as long as a certain condition holds. However, representing loops in a control-flow makes the notation used in the correctness theory complicated. This is due to the fact that each instantiation of an activity within a loop should be treated as a different element for the correctness. Therefore for the sake of simplicity, we assume that a control-flow graph does not contain cycles. With this assumption a hyperNodeGraph is refined to a hyperNodeDAG in the following definition.

Definition 4.5 [HyperNodeDAG] A *hyperNodeDAG* is a hyperNodeGraph $G = (S, E)$ in which the abstraction of all elements results in a simple DAG, and this is recursively valid for any $S_a \underline{\in} S$. □

Example 4.3 The hyperNodeGraph in Figure 6 is a hyperNodeDAG. □

Recall that, a 2-level hyperGraph representing constraints graphs of a workflow should be consistent with the control-flow graph. Since we use a hyperNodeDAG to represent the control-flow, if a 2-level hyperGraph is consistent with this graph it should be acyclic also intuitively, i.e., it should contain no cycles involving its hyperNodes or simple nodes. In this case we name this graph as a *2-level hyperDAG*. A definition of a 2-level hyperDAG is provided in [6].

In the following we provide a path definition for a hyperNodeDAG.

Definition 4.6 [A Path in a HyperNodeDAG] In a hyperNodeDAG $G = (S, E)$, a *path* is a sequence $(e_1, e_2, ..., e_k)$ of edges such that $e_i = \langle s_i, s_{i+1} \rangle \Leftrightarrow \langle S_i, S_{i+1} \rangle \in E$, where $i = 1, ..., k$ and s_i, s_{i+1} are the abstractions of the nodes $S_i, S_{i+1} \in S$ respectively. A path connecting the nodes s_1 and s_{k+1} is denoted as $\langle s_1, s_{k+1} \rangle$*-path*. □

A path definition makes it possible to identify a sequence of individual and conceptual activities which are executed one after another. For example, consider the conditional branches in a workflow specification. The possible flows between a split activity and a join activity can be specified as a set of paths between these activities.

In the following definition, we distinguish initial, final, first, and last nodes of a hyperNodeDAG. These nodes shall correspond to the specialized activities of a workflow. Initial and final nodes are simple nodes for which hyperNodes that include them and themselves have no predecessors and no successors respectively.

Furthermore, if there is a unique initial or a unique final node they are called as first and last nodes respectively. As we provide later in this section, we require a control-flow to include unique initial and final activities, i.e., it should include a first and a last activity.

Definition 4.7 **[Initial, Final, First, Last Nodes]** A simple node $\varepsilon_{in} \in S$ of a hyperNodeDAG $G = (S, E)$ is called *initial*, if $indegree(\varepsilon_{in}) = 0$, and for any S_a such that $\varepsilon_{in} \underline{\in} S_a$, $indegree(S_a) = 0$. A simple node $\varepsilon_{fin} \in S$ of a hyperNodeDAG $G = (S, E)$ is called *final*, if $outdegree(\varepsilon_{fin}) = 0$, and for any S_a such that $\varepsilon_{in} \underline{\in} S_a$, $outdegree(S_a) = 0$. If *initial (final)* node of a hyperNodeDAG $G = (S, E)$ is unique, it is the *first (last)* node of S, denoted as ε_f (ε_l). □

As mentioned previously, workflow activities can be executed sequentially or in parallel. In representing control-flow, the node where the control splits into multiple parallel activities is referred to as *split node*. The node where control merges into one activity is referred to as *join node*. We introduce split and join nodes into a hyperNodeDAG definition to model these issues; the resulting graph is called a split-join hyperNodeDAG.

Definition 4.8 **[Split, Join Nodes, Split-Join HyperNodeDAG]** A *split node* of a hyperNodeDAG $G = (S, E)$ is a simple node $S(\varepsilon_s)$ (i.e., $\varepsilon_s \in S$) for which $indegree(\varepsilon_s) \leq 1$ and $outdegree(\varepsilon_s) > 1$. A *join node* of $G = (S, E)$ is a simple node $S(\varepsilon_j)$ (i.e., $\varepsilon_j \in S$) for which $indegree(\varepsilon_j) > 1$ and $outdegree(\varepsilon_j) \leq 1$. A *split-join hyperNodeDAG* $G = (S, E)$ is a hyperNodeDAG for which the following conditions hold:

- There exist a first and a last element.
- If there is a split element this must be the first element, and there must correspond a join element to this, and this should be the last element.
- For any restriction $G(S_a)$, where $S_a \underline{\in} S$ the conditions above hold. □

In a control-flow graph, a split node from where control splits into two or more flows in order to execute activities in parallel is called an *and-split node*. After the termination of all activities involved in these flows, control merges into a join activity and execution continues from this activity. A split node where a decision is made upon which branch to take when encountered with multiple branches is called an *or/xor-split node*. Some of the branches following an or-split node, and exactly one of the branches following an xor-split node are selected for execution. This selection may depend on a condition. In our model, truth value of a condition is determined by an or/xor-split node (i.e., activity) and according to this value a branch (or some branches) are selected for execution. In this case, we name this condition as a *test condition* and associate it with the branch for which it is verified. More specifically, if s is an or/xor-split node and j is the corresponding join node, each of the branches between them is represented through a path between s and j, i.e., $\langle s, j \rangle$-$path_i$, and if a test condition $\mathcal{T}$ is used to select a branch, we label the corresponding $\langle s, j \rangle$-$path_i$ with $\mathcal{T}$. If a condition is not associated with a path we assume that its label is *true*, i.e., corresponding branch is selected for execution

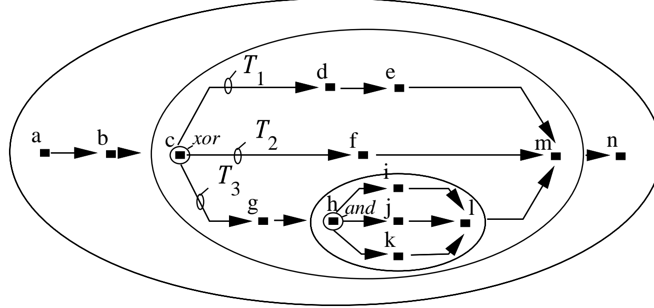


Figure 10. A Labeled Split-Join HyperNodeDAG

unconditionally. Furthermore, since some of the branches are selected for parallel execution starting from an or-split node, at least one of the test conditions of these branches should be true at a time. Similarly, exactly one of the test conditions of the branches following an xor-split node should be true.

Having defined adequate tools and setting the necessary background, a formal definition of a workflow can be provided. A workflow is defined as a 5-tuple with elements representing its activities, control and data-flow and constraints graphs.

Definition 4.9 [Workflow] A *workflow* W is a tuple $W = (N, CF, DF, IC, BC)$, where

- N is a nested hyperSet whose $base(N) = T \cup S \cup J \cup \{f, l\}$ where T is the set of *individual activities*, S is the set of *split activities*, J is the set of *join activities*, and f and l are the *first* and *last activities* respectively, and they are the virtual activities indicating the start and termination of a workflow respectively.
- $CF = (N, E_{CF}, L, TC)$ is a labeled split-join hyperNodeDAG on N corresponding to the *control-flow*. The labels L is a mapping from S to $\{and, or, xor\}$ representing the types of split nodes. The labels TC is a mapping from every $\langle s, j\rangle$*-path* in CF to $\{\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_i, ..., \mathcal{T}_n\}$, where $s \in S$ is an or/xor-split activity, $j \in J$ is the corresponding join activity, and $\mathcal{T}_i$ is a test condition. The following condition holds for every $\langle s, j\rangle$*-path* starting from a common or/xor-split activity s: If $L(s) = or$ then $\vee_{i=1}^{outdegree(s)} TC(\langle s, j\rangle\text{-}path_i) \equiv true$, and if $L(s) = xor$ then $\underline{\vee}_{i=1}^{outdegree(s)} TC(\langle s, j\rangle\text{-}path_i) \equiv true$, where $\underline{\vee}$ denotes xor operator.
- $DF = (T, E_{DF})$ is a DAG indicating the *data-flow* such that DF is consistent with CF.
- $IC = (V_{IC}, E_{IC}, L_{IC})$ is a labeled 2-level hyperDAG representing *inter-activity constraints graph*.
- $BC = (V_{BC}, E_{BC}, CL_{BC}, VL_{BC})$ is a labeled 2-level hyperDAG representing *basic constraints graph*. □

In the following an example is provided to clarify the definition of workflow.

Example 4.4 Figure 10 demonstrates a sample labeled split-join hyperNodeDAG which corresponds to a control-flow. In this graph, $N = \{a, b, \{c, d, e, f, g, \{h, i, j, k, l\}, m\}, n\}$, and $T = \{b, d, e, f, g, i, j, k\}$, $S = \{c, h\}$, $J = \{l, m\}$, $f = a$, $l = n$. Furthermore, $L(c) = xor$, $L(h) = and$, and $TC(\langle c, m\rangle\text{-}path_1) = \mathcal{T}_1$, $TC(\langle c, m\rangle\text{-}path_2) = \mathcal{T}_2$, $TC(\langle c, m\rangle\text{-}path_3) = \mathcal{T}_3$. □

In the above workflow definition, main components of a workflow are formalized. Other properties of workflows such as assignment of agents to activities, assignment of users to roles etc. are not taken into account in the formalization, since they are out of the scope of the main focus of this work. Last two components of a workflow definition, namely inter-activity constraints graph (IC) and basic constraints graph (BC) constitute our basic building blocks to develop a correctness theory for a concurrent execution of workflows. Semantics and construction of these graphs are discussed in the following section.

5. Correctness of Activities and Workflows

In this section we formalize the workflow correctness in the presence of concurrency. A workflow involves several activities each of which is performed by an agent. These activities access resources which denote the set of all objects constituting the workflow environment. We define the correct execution of activities in terms of their input and output specifications which are the set of constraints on the workflow environment. These constraints can be classified into two categories in general, namely basic constraints and inter-activity constraints which are formally defined as first-order logic formulas. The constraints that should be satisfied when an activity starts constitute the input condition of the activity. An output condition of an activity on the other hand imposes a constraint upon the workflow environment in which a workflow system must find itself after the execution of this activity.

In order to represent an interleaved execution of workflows we introduce a complete execution history and use the input and output conditions to define the correctness of this history. A complete execution history is correct if input condition of every activity involved in this history is correct when the activity starts and if the basic constraints that hold when the history starts also hold at the end of the history. We then provide a theorem which states that a complete execution history is correct if the inter-activity constraints are preserved in the required intervals and activities that require correctness of related basic constraints are prevented from executing during the intervals where these constraints do not hold. Inter-activity constraints and basic constraints are represented through inter-activity constraints graph and basic constraints graph which are used in formalizing the intervals among activities where an inter-activity constraint should be maintained and the intervals where a basic constraint remains invalid respectively.

In the following, we provide some basic definitions and notations used in representing activity and workflow semantics and in defining the correctness of workflows. We begin by defining the state of the workflow environment.

Definition 5.1 [Workflow Environment, State of the Workflow Environment] Let $RM = \cup_{i=1}^{n} RM^i$ be the set of transactional and non-transactional resource managers involved in a workflow system. The set of all variables (objects) controlled by RM^i is denoted by O^i. $O = \cup_{i=1}^{n} O^i$ denotes the set of all objects of the *workflow environment*, and $dom(o_i)$ represents the domain of an object o_i. A *state* (or *valuation*) of a workflow environment is a function $St : O \rightarrow St_\forall$, where $St_\forall = \times_{i=1}^{size(O)} dom(o_i) = dom(o_1) \times dom(o_2) \times ... \times dom(o_{size(O)})$, and $\times$ denotes the cartesian product. We use $St_\forall$ to represent the set of all possible states. □

An activity t is a mapping from $St_\forall$ to $St_\forall$, i.e., $t : St_\forall \rightarrow St_\forall$. The resulting workflow environment state after an activity t is applied to state St is denoted as $t(St)$. However, this definition of an activity is not sufficient for our purposes since we require some semantic knowledge to define correctness of activities. Activity semantic is defined in terms of constraints on the workflow environment as mentioned previously.

As specification languages, first-order logic has been the dominant choice for the expression of constraints. Therefore, to represent constraints over the objects of the workflow environment we use *First-Order Logic* (*FOL*) formulas which are denoted by calligraphic letters $\mathcal{A}$,...,$\mathcal{Z}$. More information on FOL formulas can be found in [25].

Notation: Let $\mathcal{F}$ be a FOL formula and St be a particular state of the workflow environment. We use notation $St \models \mathcal{F}$ to mean that $\mathcal{F}$ is true for the state St. If $\mathcal{F}$ is false in St this is represented as $St \not\models \mathcal{F}$. We denote the set of states that satisfy a formula $\mathcal{F}$ as $\mathcal{F}(St)$, i.e., $\mathcal{F}(St) = \{St \mid St \models \mathcal{F}\}$. The set of objects (variables) involved in a formula $\mathcal{F}$ is represented as $O(\mathcal{F})$.

Now, we can give the formal definition of a workflow activity in terms of its parameters, objects accessed, and its specification.

Definition 5.2 [Activity] An activity t is a tuple $t = (IP, OP, RS, WS, AS)$, where IP is the set of *input parameters*, OP is the set of *output parameters*, RS is the set of *objects read by* t, WS is the set of *objects updated by* t, AS is the *activity specification*. □

In the above definition, we assume that $WS \subseteq RS$. The last item, specification of an activity, is clarified through the following definition.

Definition 5.3 [Specification of an Activity] A *specification of an activity* t is a tuple $AS(t) = (I_t, O_t)$, where I_t and O_t are the set of FOL formulas on O (i.e., objects of the workflow environment). $\mathcal{I}_t \equiv \wedge_i \mathcal{I}_{t,i}$, where $\mathcal{I}_{t,i} \in I_t$, is called the *input specification* or *input condition* of t and $\mathcal{O}_t \equiv \wedge_j \mathcal{O}_{t,j}$, where $\mathcal{O}_{t,j} \in O_t$, is called the *output specification* or *output condition* of t. □

In the above definition, $\mathcal{I}_t$ ($\mathcal{O}_t$) is obtained by taking conjunction of all formulas in the set I_t (O_t). An activity is said to be *correct* with respect to a specification $AS(t) = (I_t, O_t)$ if any terminating execution of t starting from an initial state St satisfying $\mathcal{I}_t$ ends in some final state $St' = t(St)$ satisfying $\mathcal{O}_t$, i.e., $(\forall St \in St_\forall) : ((St \models \mathcal{I}_t) \Rightarrow (t(St) \models \mathcal{O}_t))$. The activities are assumed to be correct and deterministic by intuition. More information about formal specification of programs (e.g., activities) can be found in Hoare [38], and Dijkstra's works [17]. Related work includes modal and temporal logics [23].

An output condition of an activity imposes a constraint upon the workflow environment in which workflow system must find itself after the execution of this activity. The following example demonstrates this situation.

Example 5.1 The output condition of $WithdrawFromStock$ (shortly t_{WFS}) activity whose purpose is to withdraw *required* raw materials of type m_i from the stock is defined as follows:

$$\mathcal{O}_{t_{WFS}} \equiv (quantity(m_i)' = quantity(m_i) - required(m_i)). \quad (1)$$

$\mathcal{O}_{t_{WFS}}$ states that available amount of m_i is decremented by $required(m_i)$. □

The input condition characterizes the set of all initial states such that the termination of an activity will leave the system in a final state satisfying the output condition. In other words, input condition of an activity represents the states of the workflow environment in which the activity can be executed correctly. Depending on the validity of the input condition, the following three possibilities can occur [17]: (1) Activation of t leads a final state satisfying $\mathcal{O}_t$; (2) activation of t leads a final state satisfying $\neg\mathcal{O}_t$; (3) activation of t does not lead a final state, i.e., activity fails to terminate properly. Since an activity t is designed correctly and it is executed in isolation, if its input condition is satisfied then the execution of t yields in first possibility. However, if the input condition is not satisfied the execution of t may result in any of three possibilities. What constitutes the input condition of an activity is described later after possible constraints in a workflow system are introduced. The following is an example to input condition of an activity.

Example 5.2 Input condition of t_{WFS} activity states that sufficient amount of raw material of type m_i should be available in the stock:

$$\mathcal{I}_{t_{WFS}} \equiv (quantity(m_i) \geq required(m_i)). \quad (2)$$

Note that, in order to satisfy the output condition in Formula 1, this input condition must be true prior to execution of t_{WFS}. □

Intuitively, the following conditions should hold to execute an activity t correctly:

- t should read consistent (correct) values of objects in a workflow environment; hence, these consistent values should be displayed to the users and/or used to update other (or same) objects.
- If the correct execution of an activity depends on the validity of constraints that are set or verified by preceding activities, these constraints should still be valid prior to the execution of t.

In the following, we discuss these two conditions in detail. We start by describing what should be understood from correctness of a workflow environment. Correct states of a workflow environment are represented through basic constraints.

Definition 5.4 [**Basic Constraints**] A *basic constraint* $\mathcal{B}_i$ is a FOL formula defined on the objects of the workflow environment. The set of all basic constraints are represented as B and called as the *basic constraints of the workflow system.* $\mathcal{B} \equiv \wedge_i \mathcal{B}_i$, where $\mathcal{B}_i \in B$, partition the set of all possible states $St_\forall$ into two disjoint sets, $\mathcal{B}(St)$ and $St_\forall - \mathcal{B}(St)$. First is the set of *correct states* in which all basic constraints hold, and second is the set of *incorrect states* in which one or more basic constraints are violated. □

Thus, basic constraints specify the correct states of the workflow environment as the following examples demonstrate.

Example 5.3 Suppose that a basic constraint of the stock databases in the order processing example is defined as follows:

$$\mathcal{B}_1 \equiv (\Sigma_{j=1}^{w} quantity(m_{i,j}) = \mathsf{M}_i), \tag{3}$$

where $quantity(m_{i,j})$ represents the amount of raw material m_i in the stocks of warehouse j, and w is the total number of different warehouses in the enterprise. Total amount of raw material m_i currently residing at the stocks is denoted as M_i. Notice that, $\mathcal{B}_1$ does not prevent entering new raw materials of type m_i into stocks or withdrawing them for production; yet $\mathcal{B}_1$ implies that "raw materials should neither be created or destroyed during the transfer of these raw materials between the stocks of different warehouses by a *WarehouseAllocation* workflow". □

Example 5.4 Suppose that balance of unpaid bills of a customer has a predefined upper limit. Thus, a basic constraint is defined as follows:

$$\mathcal{B}_2 \equiv ((\forall c_i \in customerList) : (unpaidBalance(c_i) \leq \mathsf{U}_i)), \tag{4}$$

where *customerList* denotes the customers of the manufacturing enterprise, and $unpaidBalance(c_i)$ and U_i denote the balance of unpaid bills and the upper limit of a particular customer c_i respectively. $\mathcal{B}_2$ implies that "orders invoked by a customer should not cause an overdraft". □

These examples demonstrate that basic constraints require activities to be designed and/or arranged properly in a control-flow in order to rationally update a workflow environment, so that these basic constraints are not violated during their execution. For example, activities of *Billing* workflow should be designed properly, so that balance of unpaid bills of a customer does not cause an overdraft. The restrictions induced by basic constraints in the design of a workflow are clarified later in this section through Definition 5.9.

Some activities require that some of the basic constraints must hold to execute them correctly. Thus these basic constraints are involved in the input conditions of these activities. The set of basic constraints to be involved in the input condition of an activity t is denoted as $B(t)$, and defined as follows:

$$(\forall \mathcal{B}_i \in B) : ((O(\mathcal{B}_i) \cap RS(t) \neq \emptyset) \Rightarrow (\mathcal{B}_i \in B(t))). \tag{5}$$

According to Formula 5 if an object involved in a basic constraint $\mathcal{B}_i$ is also an element of the read set of t (i.e., $RS(t)$), $\mathcal{B}_i$ is included in the input condition of t. So activity t accesses correct states of objects in the workflow environment; otherwise t may produce incorrect results or update workflow environment erroneously.

The following example demonstrates a case in which a basic constraint is included in the input condition of an activity.

Example 5.5 Consider the basic constraint $\mathcal{B}_1$ (Formula 3), and *StockControl* workflow and its *WarehouseEvaluation* (shortly t_{WE}) activity which evaluates the available raw materials of type m_i in the stocks of all warehouses. This information is printed as a report later. Since $O(\mathcal{B}_1) \cap RS(t_{WE}) = \cup_{j=1}^{w} quantity(m_{i,j})$, (i.e., all $quantity(m_i)$ objects in w warehouses) $\mathcal{B}_1$ should be an element of basic constraints involved in the input condition of t_{WE}, i.e., $\mathcal{B}_1 \in B(t_{WE})$. Since t_{WE} should see a correct state related to amount of raw material m_i in the stocks and $\mathcal{B}_1$ describes the corresponding set of correct states, $\mathcal{B}_1$ must hold for the correct execution of t_{WE} activity. □

Assume that an incorrect state is also acceptable for a particular *WarehouseEvaluation* activity. Hence a report about approximate quantity of a raw material in the stocks is allowed. In this case, basic constraint $\mathcal{B}_1$ can be excluded from $B(t_{WE})$ although implied by the Formula 5. In this way, flexibility in the specification of incorrect but acceptable states for an activity t can be achieved. This approach resembles the *isolation levels* provided by some database management systems [33].

Although activities are usually execution-atomic (i.e., isolated) steps by their nature, there may be semantic dependencies between them that must be observed and preserved. For example, an activity may cause that a constraint to be satisfied on the workflow environment after its termination, and a successor activity may be executed with the assumption of the validity of this constraint. Furthermore, another activity may evaluate a constraint and determine its truth value, and this value may be used in the workflow specification to allow branching. Activities relying on the selected branch are likely to require validity of the constraint associated with their branch when they are executing. Both cases impose dependencies between activities. We represent such dependencies between individual activities as a set of inter-activity constraints on the workflow environment.

Definition 5.5 [**Inter-activity Constraints**] Let $W = (N, CF, DF, IC, BC)$ be a workflow, and t_i and t_j be the particular activities of this workflow, i.e., $t_i \in base(N)$, $t_j \in base(N)$. The *inter-activity constraints* between t_i and t_j, denoted as $C_{\{t_i,t_j\}}$, is a set of constraints on the workflow environment which satisfy the following conditions:

(1) t_i precedes t_j in CF.

(2) $(\forall \mathcal{D} \in C_{\{t_i,t_j\}}) : (\mathcal{D} \in I_{t_j})$.

(3) $(\forall \mathcal{D} \in C_{\{t_i,t_j\}}, \exists \mathcal{F} \in O_{t_i}) : (\mathcal{F} \Rightarrow \mathcal{D})$. □

In the above definition, if a constraint $\mathcal{F}$ in the output condition of a preceding activity t_i implies a constraint $\mathcal{D}$ in the input condition of a successor activity t_j, the

latter constraint is included in the set of inter-activity constraints between these two activities. Note that we require implication instead of equivalence between constraints $\mathcal{F}$, and $\mathcal{D}$. This is due to the fact that, validity of $\mathcal{F}$ already guarantees the validity of $\mathcal{D}$, and $\mathcal{D}$ is the constraint that is involved in the input condition of the successor activity. Thus the inclusion of the less restrictive constraint $\mathcal{D}$ in the set of inter-activity constraints is enough.

Notation: If the conditions in Definition 5.5 hold we say that constraint $\mathcal{D}$ is *emanating from* activity t_i and *incoming to* activity t_j. We use these terms to provide the reader the ability to pictorially imagine the constraint relations between activities. The set of inter-activity constraints incoming to and emanating from an activity t_j are denoted as $C_{in}(t_j)$ and $C_{out}(t_j)$ respectively and defined as follows: $C_{in}(t_j) = \cup_i C_{\{t_i,t_j\}}$, $C_{out}(t_j) = \cup_k C_{\{t_j,t_k\}}$. We denote the set of all inter-activity constraints in a workflow as C, i.e., $C = \cup_j C_{in}(t_j) = \cup_j C_{out}(t_j)$.

The following examples present some inter-activity constraints in the order processing example.

Example 5.6 Consider *CheckStock* (shortly t_{CS}) and *WithdrawFromStock* (t_{WFS}) activities. t_{CS} checks whether the required amount of raw material of type m_i (i.e., $required(m_i)$) to manufacture a particular part is available in the stock. Thus the current value of $quantity(m_i)$ (e.g., n) is determined and using this value the missing raw materials (i.e., $missing(m_i)$) that should be ordered from external vendors are calculated. Ordered raw materials are inserted into stock through *InsertStock* (t_{IS}) activity of *VendorOrder* workflow. Thus the output condition of t_{CS}, and input and output conditions of t_{IS} are defined as follows:

$$\mathcal{O}_{t_{CS}} \equiv ((quantity(m_i) = n) \wedge (missing(m_i) = required(m_i) - n)), \quad (6)$$

$$\mathcal{I}_{t_{IS}} \equiv (quantity(m_i) \geq n), \quad (7)$$

$$\mathcal{O}_{t_{IS}} \equiv ((quantity(m_i)' = quantity(m_i) + missing(m_i)) \wedge (quantity(m_i)' \geq required(m_i))), \quad (8)$$

where $quantity(m_i)'$ is the new quantity of m_i when t_{IS} is completed. Since output condition of t_{CS} implies input condition of t_{IS}, i.e., $\mathcal{O}_{t_{CS}} \Rightarrow \mathcal{I}_{t_{IS}}$, and output condition of t_{IS} implies input condition of t_{WFS} (Formula 2), i.e., $\mathcal{O}_{t_{IS}} \Rightarrow \mathcal{I}_{t_{WFS}}$, the constraints $(quantity(m_i) \geq n)$, and $(quantity(m_i) \geq required(m_i))$ are included in the sets $C_{\{t_{CS},t_{IS}\}}$, and $C_{\{t_{IS},t_{WFS}\}}$ respectively. In other words, if n particular materials of type m_i are available in t_{CS}, at least this amount of material should be available in the corresponding t_{IS} also, so $quantity(m_i)$ becomes larger than or equal to $required(m_i)$ after the insertion of missing materials into stock. $Required(m_i)$ materials should remain in the stock, so $\mathcal{I}_{t_{WFS}}$ holds when t_{WFS} is executed. Notice that $(quantity(m_i) \geq n)$ is an element of $C_{in}(t_{IS})$, and $C_{out}(t_{CS})$, and $(quantity(m_i) \geq required(m_i))$ is an element of $C_{in}(t_{WFS})$, and $C_{out}(t_{IS})$. Furthermore, both of these constraints are elements of C. □

The following is also an example from order processing workflow to further clarify inter-activity constraints.

Example 5.7 Consider $GetProcessPlan$ workflow, and its $SelectBestCells$ (t_{SBC}) activity. t_{SBC} evaluates the manufacturing cells in the factory and selects the required number of the best qualified cells to manufacture a particular part. Thus,

$$\mathcal{O}_{t_{SBC}} \equiv ((\forall cell_i \in qualifiedCells, \forall cell_j \in (cells - qualifiedCells)) : (rank(cell_i) \geq rank(cell_j))), \quad (9)$$

where $qualifiedCells$, and $rank(cell_i)$ denote the set of selected cells, and rank of a particular cell respectively. The $rank$ is obtained by evaluating qualifications, workload, capacity, etc. of a particular cell. $Cells$ denotes the set of all operational cells in the factory. Since the selected best cells should remain so until the work is actually assigned to them in the corresponding $Assign$ (t_A) activities, the input condition of a t_A activity for $cell_i$ should be defined as follows:

$$\mathcal{I}_{t_{A(cell_i)}} \equiv ((\forall cell_j \in (cells - qualifiedCells)) : (rank(cell_i) \geq rank(cell_j))). \quad (10)$$

Since $\mathcal{O}_{t_{SBC}} \Rightarrow \mathcal{I}_{t_{A(cell_i)}}$, Formula 10 should be an element of $C_{\{t_{SBC}, t_{A(cell_i)}\}}$. □

In order to represent inter-activity constraints graphically in a workflow, we use a special graph, namely *inter-activity constraints graph* which is a labeled 2-level hyperDAG defined in Section 4. In this way, inter-activity constraints can be represented in the way control and data-flow are represented.

Let $W = (N, CF, DF, IC, BC)$ be a workflow; inter-activity constraints between the activities of W are represented as a labeled 2-level hyperDAG $IC = (V_{IC}, E_{IC}, L_{IC})$, where V_{IC} and E_{IC} denote the nodes and edges respectively. V_{IC} is a hyperSet, and for any $S_a \in V_{IC}$, $S_a \subseteq base(N)$, and for any $\langle S_a, S_b\rangle \in E_{IC}$, $S_a \in base(N)$ and $S_b \subseteq base(N)$. L_{IC} are the labels of the edges and it is a mapping from the edges in E_{IC} to the inter-activity constraints in C. For a given set of inter-activity constraints between activity pairs, if there is a constraint $\mathcal{F}$ between t_i and t_j, this is represented through an edge $\langle t_i, t_j, \mathcal{F}\rangle$ in IC. If a constraint $\mathcal{F}$ emanating from an activity t_i is incoming to more than one activity, these activities are grouped into a hyperSet $S_{(t_i,\mathcal{F})}$ and this situation is represented through the edge $\langle t_i, S_{(t_i,\mathcal{F})}, \mathcal{F}\rangle$. The following example demonstrates the construction of an inter-activity constraints graph.

Example 5.8 Let $C = \{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \mathcal{F}_4, \mathcal{F}_5, \mathcal{F}_6, \mathcal{F}_7\}$, and $C_{\{t_1,t_2\}} = \{\mathcal{F}_1\}$, $C_{\{t_1,t_3\}} = \{\mathcal{F}_1\}$, $C_{\{t_2,t_4\}} = \{\mathcal{F}_2, \mathcal{F}_3, \mathcal{F}_4\}$, $C_{\{t_2,t_5\}} = \{\mathcal{F}_3, \mathcal{F}_4\}$, $C_{\{t_3,t_5\}} = \{\mathcal{F}_5\}$, $C_{\{t_3,t_6\}} = \{\mathcal{F}_5\}$, $C_{\{t_4,t_5\}} = \{\mathcal{F}_6\}$, $C_{\{t_7,t_8\}} = \{\mathcal{F}_7\}$. Therefore, as explained above, t_2 and t_3 are grouped into a hyperSet and $\langle t_1, \{t_2, t_3\}, \mathcal{F}_1\rangle$ is included in IC. Eventually, IC corresponding to C is obtained as depicted in Figure 11. □

Note that IC is consistent with control-flow graph (CF) due to Condition 1 of Definition 5.5.

An inter-activity constraints graph can be simplified by removing redundant edges from it. In general if an edge covers another edge in an inter-activity constraints

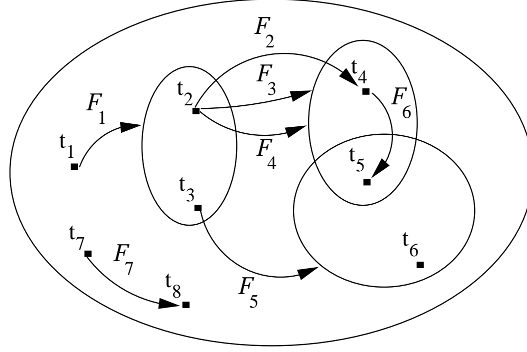


Figure 11. Inter-activity Constraints Graph

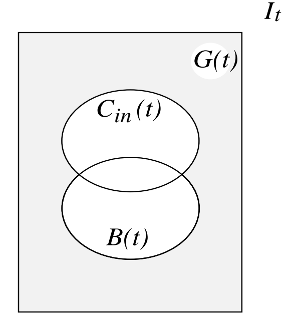


Figure 12. Relations Between Inter-activity, Basic, and Extensional Constraints

graph and constraint of the former edge implies the constraint of the latter edge, the latter edge can be removed from the graph. This is due to the fact that if first inter-activity constraint is valid between the executions of activities in its source and sink, validity of second constraint is automatically guaranteed. Furthermore, some inter-activity constraints can be removed from an inter-activity constraints graph through human intervention. If invalidity of an inter-activity constraint is acceptable for a particular activity, the edge corresponding to this constraint can be excluded from the graph by a workflow designer. This is similar to exclusion of some basic constraints from the input condition of an activity. Details of the simplification process and elimination of constraints are provided in [6].

We use an inter-activity constraints graph to develop a correctness criterion for workflows. Since inter-activity constraints contribute to the input condition of an activity, constraints in an IC graph should be preserved between the nodes of the graph during execution of the workflow since only activities are isolated not the whole workflow.

Up to this point, we have defined *basic constraints* and *inter-activity constraints.* Having defined these two types of constraints, we can now formally provide the semantic of an input condition of an activity t as follows:

$$\mathcal{I}_t \equiv (\wedge_i \mathcal{B}_i) \wedge (\wedge_j \mathcal{F}_j) \wedge (\wedge_k \mathcal{G}_k), \tag{11}$$

where $\mathcal{B}_i \in B(t)$, and $\mathcal{F}_j \in C_{in}(t)$, and $\mathcal{G}_k \in G(t)$. Intuitively, input condition of an activity is the conjunction of the basic constraints, inter-activity constraints, and constraints in $G(t)$ which are required to execute this activity correctly. $G(t)$ is composed of a set of constraints to execute t correctly which are not included in neither in $B(t)$ nor in $C_{in}(t)$ as depicted in Figure 12. Therefore constraints in $G(t)$ refer to state information which is not transferred from preceding activities or can not be represented through basic constraints. For example, consider $WithdrawFromStock$ activity (shortly t_{WFS}) and its input condition which is defined in Formula 2. Furthermore, suppose that a $CheckStock$ activity is not placed before it in the control-flow; therefore quantity of missing materials can not be

determined and inserted into stock before the execution of t_{WFS}. In this case, $(quantity(m_i) \geq required(m_i))$ is not in $B(t_{WFS})$ and $C_{in}(t_{WFS})$. This type of constraints are called as an *extensional constraints*, and included in the set $G(t)$ as depicted in Figure 12.

Later in this section we discuss the cases in which the constraints in the input condition of an activity are violated and therefore its correct execution is sacrificed. To detect these violations we are interested in whether an activity maintains a constraint. The following definition is provided to formalize this issue.

Definition 5.6 [**Preserve Function**] Let t be an activity and $\mathcal{F}$ be a FOL formula on the workflow environment. $Preserve(t, \mathcal{F})$ is a three-valued function which is defined as follows:

(1) $Preserve(t,\mathcal{F}) = true$ (1) if $(\forall St \in St_\forall) : ((St \models \mathcal{F}) \Rightarrow (t(St) \models \mathcal{F}))$. In this case we say that "t preserves $\mathcal{F}$".

(2) $Preserve(t,\mathcal{F}) = false$ (0) if $(\forall St \in St_\forall) : ((St \models \mathcal{F}) \Rightarrow (t(St) \not\models \mathcal{F}))$. In this case we say that "t falsifies (or invalidates) $\mathcal{F}$".

(3) $Preserve(t,\mathcal{F}) = may\ be$ (1/2) if $(\exists St \in St_\forall) : ((St \models \mathcal{F}) \Rightarrow (t(St) \not\models \mathcal{F}))$. In this case we say that "t may falsify (or may invalidate) $\mathcal{F}$". □

Intuitively, $Preserve(t,\mathcal{F}) = 0$ or $1/2$ requires that $WS(t) \cap O(\mathcal{F}) \neq \emptyset$. Result of $Preserve(t,\mathcal{F})$ is not always binary since the effects of an activity on the state of the workflow environment may depend on the actual values of its input parameters and/or the current values of variables in $O(\mathcal{F})$. Thus an activity may not falsify some of the constraints depending on the actual instantiation of these parameters and variables. The following is a simple example to demonstrate this situation.

Example 5.9 Let $\mathcal{F}_1 \equiv (x_1 < x_2)$, and $\mathcal{F}_2 \equiv (x_1 = x_2)$, and $t_1 = increment(x_2)$, $t_2 = decrement(x_1)$, $t_3 = increment(x_1)$, $t_4 = decrement(x_2)$. Assume that $dom(x_1)$, and $dom(x_2)$ are equal to the same totally ordered set with respect to a relation $<$. $Preserve(t,\mathcal{F}_1) = 1$ for $t \in \{t_1, t_2\}$; $Preserve(t,\mathcal{F}_1) = 1/2$ for $t \in \{t_3, t_4\}$; $Preserve(t,\mathcal{F}_2) = 0$ for $t \in \{t_1, t_2, t_3, t_4\}$. □

According to the approach described above, we would like to check activities to see whether they always preserve a constraint $\mathcal{F}$. But, the recent results in the related literature show that it is almost impossible to automatically determine the value of *Preserve* for a given activity and a constraint. As noted in [11], for transactions specified as *select-project-join expressions* of relational algebra and constraints specified as FOL formulas, it is *undecidable* to check if a given transaction preserves a given constraint. Therefore, we simply assume that a workflow system administrator and/or workflow designers can specify the value of $Preserve(t,\mathcal{F})$.

As discussed previously, basic constraints specify the correct states of the workflow environment. Invalidation of basic constraints may be permissible by the individual activities; yet this situation imposes some restrictions (1) on the execution of the workflow in which an activity that invalidates (or may invalidate) a basic constraint resides, and (2) on the execution of activities which require accessing correct states. Since basic constraints represent these correct states, if they are violated during a

workflow execution they should be resatisfied again prior to the termination of this execution. Otherwise the workflow environment is left in an incorrect state. Therefore, a workflow should be designed properly so that, if it includes an activity which falsifies (or may falsify) a basic constraint then it should include another activity (or possibly a set of activities) which certainly guarantees revalidation of this basic constraint. Furthermore, if the same basic constraint is involved in the input condition of another activity, execution of this activity should be prevented between the executions of former and latter activity (or activities). To capture these issues we have defined a validating set of activities for a basic constraint.

Definition 5.7 [**And, Or-Validating Sets**] Let $W = (N, CF, DF, IC, BC)$ be a workflow, and B be the set of basic constraints of the workflow system. Furthermore, let $t_i \in T$, where $VS \subset T$, and T represents the individual activities in N. VS is an *and-validating set for* $\mathcal{B} \in B$ if the following conditions hold:

(1) $Preserve(t_i, \mathcal{B}) = 0$ or $1/2$.

(2) $(\forall t_j \in VS) : (t_i$ precedes t_j in $CF)$.

(3) $\wedge_j \mathcal{O}_{t_j} \Rightarrow \mathcal{B}$, where $t_j \in VS$.

(4) $(\forall t_j \in VS) : (\wedge_k \mathcal{O}_{t_k} \not\Rightarrow \mathcal{B})$, where $t_k \in (VS - t_j)$.

VS is an *or-validating set for* $\mathcal{B} \in B$ if the following conditions hold:

(1) Conditions 1, and 2 above.

(2) $(\forall t_j \in VS) : (\mathcal{O}_{t_j} \Rightarrow \mathcal{B})$. □

Informally, VS is an and-validating set for $\mathcal{B}$ if $\mathcal{B}$ is a basic constraint which is (or may be) invalidated by t_i, and validated collectively by the elements of VS. Condition 4 guarantees that execution of activities in a subset of an and-validating set VS is not a sufficient condition for the validation of $\mathcal{B}$, and therefore VS is the minimum set of activities to validate $\mathcal{B}$. If the execution of at least one element of a set of activities (VS) is a sufficient condition for the validation of $\mathcal{B}$ we call VS as the or-validating set for $\mathcal{B}$.

Notation: We denote the set of basic constraints which are (or may be) invalid between t_i and activities of an and-validating set VS as $SB_{\{t_i, VS, and\}}$. The set of basic constraints which are (or may be) invalid between t_i and at least one activity of an or-validating set VS is denoted as $SB_{\{t_i, VS, or\}}$.

In the following, we clarify these definitions through examples.

Example 5.10 Consider the *WarehouseAllocation* workflow in Figure 3. Output conditions of *RetrieveMaterial* (t_{RM}), and *UpdateMaterialLocation* (t_{UML}) activities of a *WarehouseAllocation* workflow are defined as follows:

$$\mathcal{O}_{t_{RM(w_j)}} \equiv (quantity(m_{i,j})' = quantity(m_{i,j}) - n), \quad (12)$$

$$\mathcal{O}_{t_{UML(w_k)}} \equiv (quantity(m_{i,k})' = quantity(m_{i,k}) + l_k), \quad (13)$$

where w_j represents the source warehouse, and w_k represents a warehouse k in $destList$, i.e., $w_k \in destList$, and $\Sigma_{k=1}^{size(destList)} l_k = n$. Since after n raw materials of type m_i are withdrawn from the stock of warehouse j, $\mathcal{B}_1$ (Formula 3) is no longer true of the workflow environment state. However, $\mathcal{B}_1$ is resatisfied after the termination of the corresponding t_{UML} activities which distribute withdrawn amount to stocks at different warehouses in $destList$. In this case $t_{UML(w_k)}$ activities for each warehouse k constitute an and-validating set for $\mathcal{B}_1$, since after the termination of all activities in this set $\mathcal{B}_1$ is satisfied again, and therefore $SB_{\{t_{RM(w_j)}, \cup_{k=1}^{size(destList)} t_{UML(w_k)}, and\}} = \{\mathcal{B}_1\}$. □

The following is an example to an or-validating set for a basic constraint.

Example 5.11 Consider $Billing$ workflow and its $UpdateUnpaidBalance$ (t_{UUB}), $RejectShipping$ (t_{RS}), and $MoreCredit$ (t_{MC}) activities (Figure 2). Their output conditions are defined as follows:

$$\mathcal{O}_{t_{UUB}} \equiv (unpaidBalance(c_i)' = unpaidBalance(c_i) + b) \quad (14)$$

$$\mathcal{O}_{t_{RS}} \equiv ((unpaidBalance(c_i)' = unpaidBalance(c_i) - b) \wedge (orderStatus = rejected)) \quad (15)$$

$$\mathcal{O}_{t_{MC}} \equiv ((\mathsf{U}_i' = \mathsf{U}_i + c) \wedge (\mathsf{U}_i' \geq unpaidBalance(c_i))), \quad (16)$$

where U_i' denotes the new upper limit after t_{MC} is terminated. If a customer c_i does not pay the bill of an ordered product, her/his balance of unpaid bills (i.e., $unpaidBalance(c_i)$) is updated in t_{UUB} activity (Formula 14 above). Since $Preserve(t_{UUB}, \mathcal{B}_2) = 1/2$, basic constraint $\mathcal{B}_2$ (Formula 4) may be invalid at this moment. In this case either shipping of ordered product is rejected (or delayed) and $unpaidBalance(c_i)$ is decremented in t_{RS} activity (Formula 15), or if responsible branch of the enterprise grants more credit to this customer, her/his upper limit (U_i) is incremented in t_{MC} activity, thus $\mathsf{U}_i \geq unpaidBalance(c_i)$ holds (Formula 16). Observe that $\mathcal{B}_2$ is certainly satisfied after the termination of either t_{RS} or t_{MC} activity. Therefore t_{RS} and t_{MC} activities constitute an or-validating set for $\mathcal{B}_2$, and $SB_{\{t_{UUB}, \{t_{RS}, t_{MC}\}, or\}} = \{\mathcal{B}_2\}$. □

As the previous examples demonstrate activities of an and/or-validating set guarantee revalidation of a basic constraint. Yet to achieve this, there is a prerequisite which is a natural outcome of our definition of activity semantic: Input conditions of activities of an and/or-validating set should hold when they are executed. Only in this way Condition 3 for an and-validating set, and Condition 2 for an or-validating set in Definition 5.7 can be satisfied. To achieve this, required inter-activity constraints between the activity which (may) invalidate a basic constraint and activities in the corresponding validating set should be preserved. The following example demonstrates this requirement.

Example 5.12 In the manufacturing example, a product is composed of parts and parts are further composed of raw materials. Therefore consistency of technical data, i.e., design information belonging to a product and its constituting parts is an essential requirement in a manufacturing process. To state this, a basic constraint

of the system is defined as follows:

$$\mathcal{B}_3 \equiv ((\forall prod_i \in products, \forall part_j \in parts) : ((part_j \in P(prod_i)) \Rightarrow Consistent(design(prod_i), design(part_j)))). \quad (17)$$

Acording to $\mathcal{B}_3$, design of a product, i.e., $design(prod_i)$, should be consistent with designs of its constituting parts, i.e., $design(part_j)$, where $part_j \in P(prod_i)$. Let $UpdatePartDesign$ (shortly t_{UPartD}) and $UpdateProductDesign$ (t_{UProdD}) be two activities whose output conditions are defined as follows:

$$\mathcal{O}_{t_{UPartD}} \equiv ((design(part_j)' = design(part_j) + \Delta) \wedge Consistent(design(prod_i) + F(\Delta), design(part_j)')) \quad (18)$$

$$\mathcal{O}_{t_{UProdD}} \equiv ((design(prod_i)' = design(prod_i) + F(\Delta)) \wedge Consistent(design(prod_i)', design(part_j))) \quad (19)$$

where $design(part_j)'$ and $design(prod_i)'$ represent new designs. t_{UPartD} changes design of a part by Δ, and t_{UProdD} updates corresponding product through a function $F(\Delta)$, so that the consistency of designs for product and its part is achieved again after t_{UProdD}, i.e., $\mathcal{O}_{t_{UProdU}} \Rightarrow \mathcal{B}_3$. In order to get the above result, however, input condition of t_{UProdD} should include the constraint $Consistent(design(prod_i) + F(\Delta), design(part_j))$. That is, prior to execution of t_{UProdD}, change made in $design(part_j)$ must remain the same (i.e., no other activities change the design of the part), so update of $design(prod_i)$ by $F(\Delta)$ should make the design of product consistent with its part again. Note that, the output condition of t_{UPartD} also includes this constraint since this part is redesigned with the assumption that the product design will change accordingly. As a result, the constraint $Consistent$ $(design(prod_i) + F(\Delta), design(part_j))$ is included in the set of inter-activity constraints between t_{UPartD}, and t_{UProdD}, i.e., it is an element of $C_{\{t_{UPartD}, t_{UProdD}\}}$. □

In Definition 5.7, it is assumed that if a basic constraint is (or may be) invalidated by a previously executed activity, its revalidation is guaranteed by successor activities in control-flow. However, this invalidation can be prevented through the execution of a preceding activity or a set of activities. More precisely, if $Preserve(t, \mathcal{B}) = 1/2$ invalidation of $\mathcal{B}$ by the execution of t can be prevented by the execution of some preceding activities in control-flow, thus $\mathcal{O}_t \Rightarrow \mathcal{B}$ [6]. The details are omitted here due to space limitations.

The presented examples provide sufficient guidance for workflow designers, so if their workflow specification includes an activity which (may) invalidates a basic constraint they should also include other activities conforming to the definitions of validating sets or prevent this invalidation by placing preceding activities.

We formally represent and/or-validating sets and intervals at which the basic constraints are (or may be) invalid during the execution of a workflow W, through a labeled 2-level hyperDAG $BC = (V_{BC}, E_{BC}, CL_{BC}, VL_{BC})$, where V_{BC}, and E_{BC} represent nodes, and edges respectively. V_{BC} is a hyperSet, and for any $S_a \in V_{BC}$, $S_a \subseteq T$ and for any $\langle S_a, S_b \rangle \in E_{BC}$, $S_a \in T$ and $S_b \subseteq T$. Recall that T is the set of individual activities of W. CL_{BC} and VL_{BC} are the labels of edges

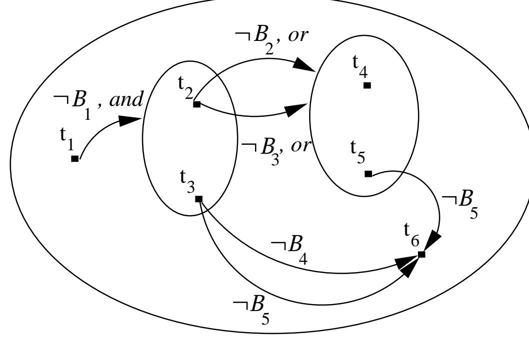


Figure 13. Basic Constraints Graph

in E_{BC}; CL_{BC} is a mapping from E_{BC} to negated elements of B, where B is the set of basic constraints of the workflow system, and VL_{BC} is a mapping from E_{BC} to $\{and, or\}$ denoting the types of validating sets. E_{BC} is constructed through the use of following principles:

- $(\forall \mathcal{B} \in B) : ((\mathcal{B} \in SB_{\{t_i,VS,and\}}) \Rightarrow (\langle t_i, VS, \neg\mathcal{B}, and\rangle \in E_{BC}))$.
- $(\forall \mathcal{B} \in B) : ((\mathcal{B} \in SB_{\{t_i,VS,or\}}) \Rightarrow (\langle t_i, VS, \neg\mathcal{B}, or\rangle \in E_{BC}))$.

According to these principles, if VS is an and-validating set or an or-validating set for $\mathcal{B}$ this situation is represented by the edges $\langle t_i, VS, \neg\mathcal{B}, and\rangle$ and $\langle t_i, VS, \neg\mathcal{B}, or\rangle$ respectively. Note that if VS includes more than one activity it is represented as a hyperSet in BC. If VS has one element, this element is represented with a simple node, and since type of VS (i.e., and/or) is immaterial in this case, label of the edge incoming to VS representing its type is omitted. Furthermore BC is consistent with control-flow graph (CF) due to Condition 2 of Definition 5.7. The following example demonstrates the construction of a basic constraints graph using the principles above.

Example 5.13 Let $B = \{\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3, \mathcal{B}_4, \mathcal{B}_5, \mathcal{B}_6, \mathcal{B}_7\}$, and $SB_{\{t_1,\{t_2,t_3\},and\}} = \{\mathcal{B}_1\}$, $SB_{\{t_2,\{t_4,t_5\},or\}} = \{\mathcal{B}_2,\mathcal{B}_3\}$, $SB_{\{t_3,t_6\}} = \{\mathcal{B}_4, \mathcal{B}_5\}$, $SB_{\{t_5,t_6\}} = \{\mathcal{B}_5\}$. The corresponding basic constraints graph BC is depicted in Figure 13. □

We use basic constraints graph in conjunction with inter-activity constraints graph to develop the notion of correct execution of workflows. To define a correctness criterion we need the definition of a complete execution history of workflow instances. In the following, the definition of a complete execution of a workflow is provided which is then used in defining the history.

In Section 4, control-flow of a workflow is formalized as a labeled split-join hyperNodeDAG. In this graph, or/xor-split nodes cause some activities of the workflow not to take place in the actual execution. This is due to the fact that after the execution of an or/xor-split node a decision is made upon which branch to take. To define the parts of a workflow which are actually executed at run-time, namely a complete

execution of a workflow, the following algorithm is provided. In this algorithm, $G = (T_G, E_G, L_G, TC_G)$ is a labeled split-join hyperNodeDAG which is local to the algorithm itself. The split-join hyperNodeDAG $CE = (N_{CE}, E_{CE})$ is the resulting complete execution graph for a given control-flow graph, $CF = (N, E_{CF}, L, TC)$.

Algorithm 5.1 [Complete Execution Generation Algorithm]

---

procedure PathGenerate(G):
begin
1. $f \leftarrow first(G/\varepsilon_1/\varepsilon_2.../\varepsilon_{size(T_G)})$, *where* $G = (T_G, E_G, L_G, TC_G)$;
2. $l \leftarrow last(G/\varepsilon_1/\varepsilon_2.../\varepsilon_{size(T_G)})$;
3. **if** f *is a split node* **then**
4. **case** $L_G(f)$ **of**
   begin
5. *and* : **for** *every* $\langle f, l\rangle$*-path* $\subseteq E_G$ **do** $E_{CE} \leftarrow E_{CE} \cup \langle f, l\rangle$*-path;*
6. *or* : **for** *some* $\langle f, l\rangle$*-path* $\subseteq E_G$ **do** $E_{CE} \leftarrow E_{CE} \cup \langle f, l\rangle$*-path;*
7. *xor* : **for** *exactly one* $\langle f, l\rangle$*-path* $\subseteq E_G$ **do** $E_{CE} \leftarrow E_{CE} \cup \langle f, l\rangle$*-path;*
   end
8. **else** $E_{CE} \leftarrow E_{CE} \cup \langle f, l\rangle$*-path*
end

program main:
begin
1. $N_{CE} \leftarrow \emptyset$, $E_{CE} \leftarrow \emptyset$;
2. *PathGenerate(CF);*
3. **for** *every node* $\varepsilon_{CE} \in N_{CE}$ *and* $\varepsilon_{CE} \in hyper(N)$ **do**
4. *PathGenerate*$(CF(\varepsilon_{CE}))$
end

The procedure *PathGenerate* accepts a labeled split-join hyperNodeDAG G as an input. In *Steps 1-2*, each hyperNode of G is replaced with an abstract simple element; thus it results in a simple DAG. First and last elements of the DAG are assigned to f and l respectively. If f is an *and-split node* all paths connecting it to l are included in CE; if f is an *or-split node* some of the paths connecting it to l are included in CE; if f is an *xor-split node* exactly one of the paths connecting it to l is included in CE. If f is not a split node, single $\langle f, l\rangle$*-path* is included in CE.

The main program which calls procedure *PathGenerate* is also provided above. After initialization, this main program executes *PathGenerate* for control-flow, CF. For every node ε_{CE} included in CE after this step (i.e., $\varepsilon_{CE} \in N_{CE}$), if this node corresponds to a hyperNode in CF, *PathGenerate* is called with the restriction of CF to this node (i.e., $CF(\varepsilon_{CE})$) as the input. The program executes until there is no element in CE corresponding to a hyperNode in CF. In this way, a complete execution is generated in a top-down fashion.

In the following, a complete execution of a workflow is formally defined as an outcome of the main program above.

Definition 5.8 [Complete Execution of a Workflow] Let $W = (N, CF, DF, IC, BC)$ be a workflow, and $CF = (N, E_{CF}, L, TC)$ be its control-flow, where CF itself is thought as a single node at an abstract level. A *Complete Execution of W* denoted as $CE = (N_{CE}, E_{CE})$ is a split-join hyperNodeDAG which can be generated through the Complete Execution Generation Algorithm (Algorithm 5.1). □

Notice that there could be many complete executions that can be generated from the control-flow graph using Algorithm 5.1. The following example demonstrates the generation of a complete execution from a given control-flow.

Example 5.14 Consider the control-flow graph (CF) in Figure 10. One of the complete executions that is generated from CF, e.g., $CE_1 = (N_{CE_1}, E_{CE_1})$, can be defined as follows: $N_{CE_1} = \{a, b, \{c, g, \{h, i, j, k, l\}, m\}, n\}$, and $E_{CE_1} = \{\langle a, b\rangle, \langle b, \varepsilon_3\rangle, \langle \varepsilon_3, n\rangle, \langle c, g\rangle, \langle g, \varepsilon_{33}\rangle, \langle \varepsilon_{33}, m\rangle, \langle h, i\rangle, \langle h, j\rangle, \langle h, k\rangle, \langle i, l\rangle, \langle j, l\rangle, \langle k, l\rangle\}$, where $\varepsilon_3 = \{c, g, \{h, i, j, k, l\}, m\}$, and $\varepsilon_{33} = \{h, i, j, k, l\}$. □

As stated previously, basic constraints can be violated during a workflow execution; yet as one of the essential conditions to preserve them all complete executions must satisfy the criteria given in the following definition.

Definition 5.9 [Validation Complete Control-Flow] Let $W = (N, CF, DF, IC, BC)$ be a workflow, and $BC = (V_{BC}, E_{BC}, CL_{BC}, VL_{BC})$ be its basic constraints graph. CF is a *Validation Complete Control-Flow* if the following conditions hold for every complete execution $CE_i = (N_{CE_i}, E_{CE_i})$ of W:

(1) $(\forall\langle t, VS, \neg\mathcal{B}, and\rangle \in E_{BC}) : ((t \in base(N_{CE_i})) \Rightarrow (VS \subseteq base(N_{CE_i})))$.

(2) $(\forall\langle t, VS, \neg\mathcal{B}, or\rangle \in E_{BC}) : ((t \in base(N_{CE_i})) \Rightarrow (VS \cap base(N_{CE_i}) \neq \emptyset))$. □

Conditions 1 and 2 state that if an activity (t) does not preserve a basic constraint (i.e., $\mathcal{O}_t \not\Rightarrow \mathcal{B}$), then every complete execution (CE_i) including this activity must contain activities which validate this basic constraint again (i.e., activities of the corresponding and-validating set or at least one activity of corresponding or-validating set). This property must be ensured by the workflow designers. Note that, if $Preserve(t, \mathcal{B}) = 1/2$ and invalidation of $\mathcal{B}$ is prevented by the preceding activities then $\mathcal{O}_t \Rightarrow \mathcal{B}$. In this case, t is not placed in BC.

The following example clarifies the definition above.

Example 5.15 $WarehouseAllocation$ (Example 5.10), and $Billing$ (Example 5.11) workflows have validation complete control-flows, since intuitively every complete execution of $WarehouseAllocation$ workflow includes the activities in $\cup_{k=1}^{size(destLit)} t_{UML(w_k)}$ if it includes $t_{RM(w_j)}$, and every complete execution of $Billing$ workflow includes either t_{RS} or t_{MC} activity in the case $\mathcal{B}_2$ is falsified by t_{UUB}. □

A workflow environment can be left in an incorrect state due to incorrect interleavings during the execution of activities of the same or different workflows even these individual workflows have validation complete control-flows. Furthermore inter-activity constraints can be invalidated and therefore input conditions of

Table 1. Relations Defined on Time Intervals

Relation	**Condition**
TI_i and TI_j intersect	$\neg(END(TI_i) < START(TI_j)) \wedge$ $\neg(END(TI_j) < START(TI_i))$
TI_i covers TI_j	$(START(TI_i) < START(TI_j)) \wedge$ $(END(TI_j) < END(TI_i))$

some activities may be false when they are executed. Both situations sacrifice the correctness of workflows. Before introducing a correctness notion, we provide a formal definition of concurrent execution of workflows, namely a complete execution history of workflows. To specify interleavings of workflows and their constituting activities clearly in this definition, time intervals are associated with them during execution.

Assuming a model consisting of a fully ordered set of points (instants) of time, a time interval TI is an ordered pair of points which represents its endpoints, i.e., $TI = [START(TI), END(TI)]$, where $START(TI)$ and $END(TI)$ denote the start-point and end-point of TI respectively. Two relations between the time intervals, namely *intersect* and *cover* are presented in Table 1. In this table, TI_i and TI_j represent two arbitrary time intervals. TI_i and TI_j intersect, which is denoted as $TI_i \cap TI_j \neq \emptyset$, if they have at least a common point of time. If TI_i covers TI_j this is denoted as $TI_i \supset TI_j$. These relations are used later in this section. More information about time intervals and relations between them can be found in [2].

After introducing time intervals and required relations among them, the following definition of the complete execution history of workflows is presented.

Definition 5.10 [Complete Execution History of Workflows] A *Complete Execution History* $CH = (T_{CH}, E_{CH}, L_{CH})$ defined over a set of complete workflow executions $CE = \{CE_1, CE_2, ..., CE_n\}$, where $CE_1, CE_2, ..., CE_n$ are generated from control-flows of a set of workflows $W = \{W_1, W_2, ..., W_m\}$, is a labeled split-join hyperNodeDAG, where

- $T_{CH} = \cup_{i=1}^{n} N_{CE_i} \cup \{s_{CH}, j_{CH}\}$, where s_{CH} and j_{CH} denote the split and join nodes of CH respectively, and s_{CH}, j_{CH} are equal to f_{CH} and l_{CH} (first and last nodes of CH) respectively.
- $E_{CH} = (\cup_{i=1}^{n} E_{CE_i}) \cup (\cup_{i=1}^{n} \{\langle s_{CH}, N_{CE_i}\rangle, \langle N_{CE_i}, j_{CH}\rangle\})$.
- L_{CH} is the labels of the nodes, i.e., each node is labeled with its time interval TI. For a simple node S, $TI_S = [start(S), end(S)]$, where $start(S)$ and $end(S)$ denote the time instants when the activity is started and terminated respectively. For a hyperNode S, $TI_S = [min(START(TI_{S_i})), max(END(TI_{S_i}))]$, where S_i is a simple or a hyperNode of S (i.e., $S_i \Subset S$). □

In the following definition, a correctness criterion for a complete execution history of workflows is presented. In this definition, a correct complete execution history is characterized by referring to the properties of the workflow environment state at particular time instants. Intuitively, for an infinite sequence $\tau = 0, 1, 2, ...$ of time

instants there is a corresponding sequence $St_0, St_1, St_2, ...$ of workflow environment states. The notation St_{event} is employed to denote a particular workflow environment state at the time instant with which the *event* is associated. For example, $St_{start(t)}$ denotes the state when activity t is started. If a constraint $\mathcal{F}$ holds at the time instant at which *event* occurs, this situation is represented as $St_{event} \models \mathcal{F}$.

Definition 5.11 [**Correct Complete Execution History**] A *Complete Execution History* $CH = (T_{CH}, E_{CH}, L_{CH})$ is *correct* if the following conditions hold:

(1) $(\forall t \in base(T_{CH})) : (St_{start(t)} \models \mathcal{I}_t)$.

(2) $(St_{start(f_{CH})} \models \mathcal{B}) \Rightarrow (St_{end(l_{CH})} \models \mathcal{B})$, where f_{CH} and l_{CH} are the first and last nodes of CH respectively, and $\mathcal{B} = \wedge_i \mathcal{B}_i$ where $\mathcal{B}_i \in B$, and B is basic constraints of the workflow system. □

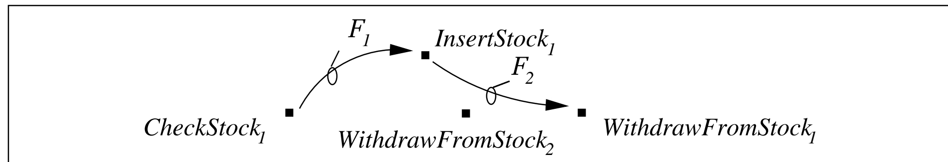
Condition 1 states that when an activity t involved in the history is started its input condition $\mathcal{I}_t$ should hold. Notice that since the individual activities are isolated, validity of their input conditions when they are started is a sufficient condition to execute them correctly. According to Condition 2, if the basic constraints of the workflow system are true when the history is started they should be true after the termination of the history.

After defining a correctness notion for a complete execution history of workflows the ways correctness can be sacrificed are illustrated in the following paragraphs. If the execution of activities of workflows are interleaved, correctness of a complete execution history can be violated in two ways:

- Input condition of an activity t may be false when t is executed (i.e., $St_{start(t)} \not\models \mathcal{I}_t$).
- Although basic constraints are true when the complete execution history is started, they may be false when it is terminated (i.e., $St_{execute(l_{CH})} \not\models \mathcal{B}$).

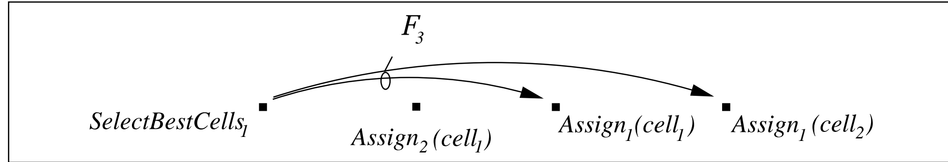
Input condition of an activity (Formula 11) can be violated in three ways: (1) An inter-activity constraint $\mathcal{F} \in C_{in}(t)$, or (2) a basic constraint $\mathcal{B} \in B(t)$, or (3) an extensional constraint $\mathcal{G} \in G(t)$ may not be true when t is executed. The following two examples demonstrate the first case.

Example 5.16 Consider the *CheckStock* (t_{CS}), *InsertStock* (t_{IS}), and *WithdrawFromStock* (t_{WFS}) activities, and the inter-activity constraints $\mathcal{F}_1 \equiv (quantity(m_i) \geq n)$, and $\mathcal{F}_2 \equiv (quantity(m_i) \geq required(m_i))$ given in Example 5.6. Remember that $\mathcal{F}_1 \in C_{\{t_{CS}, t_{IS}\}}$, and $\mathcal{F}_2 \in C_{\{t_{IS}, t_{WFS}\}}$. Since raw materials of type m_i may be withdrawn from the stock by the concurrently executing t_{WFS} activity of some other workflows, $\mathcal{F}_1$, and $\mathcal{F}_2$ may be invalidated between the t_{CS}, and t_{IS} activities, and corresponding t_{WFS} activity. This situation is depicted in the following:



Suppose that t_{CS_1} sees $n = 75$ raw materials in the stock and $required(m_i) = 125$; therefore 50 raw materials are ordered from vendors and inserted into stock through t_{IS_1} activity. After this, if a t_{WFS_2} activity of another instance of $OrderProcessing$ workflow withdraws 30 raw materials of same type, input condition of t_{WFS_1} (i.e., $quantity(m_i) \geq 125$) is invalidated. □

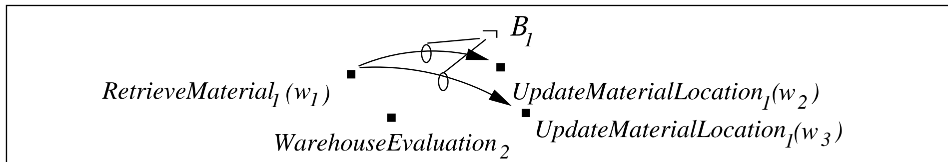
Example 5.17 Consider $SelectBestCells$ (t_{SBC}) and $Assign$ (t_A) activities, and the inter-activity constraint $\mathcal{F}_3 \equiv ((\forall cell_j \in (qualifiedCells)) : (rank(cell_i) \geq rank(cell_j)))$ defined in Example 5.7. Recall that $\mathcal{F}_3 \in C_{\{t_{SBC}, t_{A(cell_i)}\}}$. Since other t_A activities might concurrently assign a work to a preselected cells they can invalidate $\mathcal{F}_3$. This situation is depicted as follows:



Suppose that available cells are evaluated in t_{SBC_1}, and $cell_1$ and $cell_2$ are selected. If $t_{A_2(cell_1)}$ assigns a heavy work to $cell_1$, and degrades its previously assessed $rank$, $cell_1$ may become a worse selection for the assignment of the work in $t_{A_1(cell_1)}$. Thus input condition of $t_{A_1(cell_1)}$ may be invalid when it is executed. □

The following example demonstrates a situation in which a basic constraint involved in the input condition of an activity is falsified.

Example 5.18 Consider Examples 5.5 and 5.10, and note that basic constraint $\mathcal{B}_1$ is false between $RetrieveMaterial(w_j)$ (shortly $t_{RM(w_j)}$), and corresponding $UpdateMaterialLocation(w_k)$ ($t_{UML(w_k)}$) activities for every $w_k \in destList$. If a $WarehouseEvaluation$ (t_{WE}) activity is executed between these activities it executes incorrectly, since its input condition includes $\mathcal{B}_1$. This situation is demonstrated in the following:



Suppose that $t_{RM_1(w_1)}$ retrieves 1200 raw materials of type m_i from the stock of warehouse w_1 and these materials are distributed to stocks of warehouses w_2, and w_3 through $t_{UML_1(w_k)}$ activities. If t_{WE_2} activity is executed between them it misses the raw materials being transferred and an incorrect amount of raw material m_i is reported. □

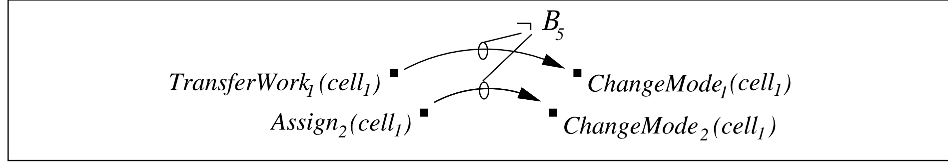
The preceding examples demonstrate the possible violations of input conditions. Now, we discuss the cases in which basic constraints may remain false after the termination of a complete execution history.

Note that validation completeness (Definition 5.9) is an essential requirement to preserve basic constraints in a complete execution history, thus if a basic constraint is invalidated by an activity it is revalidated by the execution of activities in its validating set. Yet to achieve this, the input conditions of activities in the validating set must hold when they are executed (Example 5.12). If input conditions of activities in a validating set are falsified, revalidation of a basic constraint fails. Thus, although workflows having validation complete control-flows are involved in a complete execution history, a workflow environment can be left in a state where basic constraints do not hold. The following example demonstrates this situation.

Example 5.19 Suppose that a basic constraint $\mathcal{B}_5$ is defined as follows:

$$\mathcal{B}_5 \equiv ((\forall cell_i \in cells) : (((capacityMode(cell_i) = Normal) \Rightarrow (workload(cell_i) \leq \mathsf{C}_i)) \vee ((capacityMode(cell_i) = Max) \Rightarrow (\mathsf{C}_i < workload(cell_i) \leq \mathsf{MAX}_i)))) \quad (20)$$

The intuition behind this constraint is as follows: A manufacturing cell ($cell_i$) can work in normal ($Normal$) or maximum (Max) capacity modes. If $cell_i$ works in $Normal$ mode, its workload should be equal or less than a predetermined upper limit C_i. In Max mode, its workload should be between C_i and MAX_i. Employing cells in $Normal$ load is more desirable, and transferring a part of a workload to other available cells is possible. Consider the following executions of related activities:



Assume that $\mathsf{MAX}_1 = 500$, $\mathsf{C}_1 = 300$, and current workload of $cell_1$ is 400. $TransferWork_1(cell_1)$ (shortly $t_{TW_1(cell_1)}$) transfers a part of $cell_1$'s workload (i.e., 150) to other available cells. In this case, $\mathcal{B}_5$ is invalidated and $ChangeMode_1(cell_1)$ ($t_{CM_1(cell_1)}$) should be executed to change mode of $cell_1$ from Max to $Normal$. Notice that to guarantee validation of $\mathcal{B}_5$, inter-activity constraint

$$\mathcal{F}_4 \equiv (workload(cell_1) \leq \mathsf{C}_1) \quad (21)$$

must hold when $t_{CM_1(cell_1)}$ is executed. Thus, $cell_1$ works in $Normal$ capacity mode with $workload = 250$, and therefore $\mathcal{B}_5$ is revalidated after the termination of $t_{CM_1(cell_1)}$. This situation is similar to one presented in Example 5.12. Consider the executions of activities which belong to another workflow instance. Suppose that $Assign_2(cell_1)$ ($t_{A_2(cell_1)}$) assigns a work to $cell_1$ in amount of 200, and therefore the resulting $workload$ is 450. Since this workload requires Max capacity mode $t_{CM_2(cell_1)}$ is executed to validate $\mathcal{B}_5$, and $capacityMode(cell_1)$ is made Max. Note that the activities presented belong to workflows having validation complete control-flows. At the end of these executions, the resulting $capacityMode$ is $Normal$ and current $workload$ is equal to 450. Thus $\mathcal{B}_5$ is still invalid. This is due to $t_{A_2(cell_1)}$

is invalidated $\mathcal{F}_4$ which is required for the correct execution of $t_{CM_1(cell_1)}$. □

As discussed through the preceding examples, although individual activities of a workflow are executed in isolation, workflow correctness may be violated due to improper interleavings. Thus, proper concurrency control mechanisms are required to ensure correctness of a complete execution history. A concurrency control mechanism can guarantee that when t_j is executed $\mathcal{I}_{t_j}$ is true if it does not permit any activity that falsifies constraints in $C_{\{t_i,t_j\}}$ to be executed between t_i and t_j for different t_is. Furthermore, if a basic constraint involved in $\mathcal{I}_{t_j}$ is invalidated by a previously executed activity, execution of t_j should be delayed until this basic constraint is satisfied again by the activities of corresponding validating set. Revalidation of a basic constraint can be ensured by the validation completeness property, and guaranteeing correctness of input conditions of activities in a validating set.

Extensional constraints (i.e., $G(t_j)$) involved in the input condition of an activity may be falsified by the activities which are terminated even before the beginning of workflow in which t_j participates, and remain invalid for an uncertain time. Therefore, ensuring their validity like inter-activity or basic constraints through a concurrency control mechanism is not possible. A possible way to achieve this is that, a workflow designer places preceding activities in the control-flow to check these constraints, and if they evaluate to false either they are validated by proper activities or t_j is excluded from the execution history through conditional branches. Placing *CheckStock* and *InsertStock* activities before the *WithdrawFromStock* is an example to the first case. In this way, extensional constraints can be transformed to inter-activity constraints and their validity can be ensured like other constraints. If this design requirement is not taken into consideration by workflow designers, activity itself should verify extensional constraints, and if they evaluate to false, the activity should be removed from the execution history (e.g., by aborting it).

The essential design requirements which provide for the correctness of a complete execution history of workflows and hence must be ensured by the workflow designers can be summarized as follows: (1) Control-flow of workflows must be validation-complete; (2) proper inter-activity constraints must be introduced between the activities which invalidate and later revalidate a basic constraint; (3) extensional constraints must be transformed to inter-activity constraints, thus $G(t_j) = \emptyset$.

Theorem 5.1 provides the concurrency control requirements explained above in a formal manner. To specify the intervals where the basic constraints are (or may be) invalid, and where inter-activity constraints should be preserved at run-time in the theorem, time intervals (TI_E) are associated with the edges of a basic constraints graph (BC), and inter-activity constraints graph (IC) in the following:

- If E is an edge of an IC then, $TI_E = [START(TI_{source(E)}), END(TI_{sink(E)})]$, i.e., TI_E is denoted by the start of time interval associated with the source node and end of time interval associated with the sink node of E.
- If E is an edge of a BC, and $VL_{BC} = and$ then, $TI_E = [START(TI_{source(E)}), END(TI_{sink(E)})]$.
- If E is an edge of a BC, and $VL_{BC} = or$ then, $TI_E = [START(TI_{source(E)}), min(END(TI_{S_i}))]$, and $S_i \in sink(E)$, i.e., TI_E is denoted by the start of

time interval associated with the source node, and minimum end-point of time intervals associated with the elements of the sink node of E. This is due to the fact that once an activity in $sink(E)$ is terminated, validity of a basic constraint is ensured.

Theorem 5.1 [Correctness of a Complete Execution History] Let $CH = (T_{CH}, E_{CH}, L_{CH})$ be a complete execution history defined over a set of complete executions $CE = \{CE_1, CE_2, ..., CE_n\}$, where $CE_1, CE_2, ..., CE_n$ are generated from a set of workflows $W = \{W_1, W_2, ..., W_m\}$ having validation complete control-flows. $W_i \in W$ is represented as $W_i = (N_i, CF_i, DF_i, IC_i, BC_i)$, where $IC_i = (V_{IC_i}, E_{IC_i}, L_{IC_i})$, and $BC_i = (V_{BC_i}, E_{BC_i}, CL_{BC_i}, VL_{BC_i})$. CH is *correct* if the following conditions hold:

(1) $St_{start(f_{CH})} \models \mathcal{B}$.

(2) $(\forall W_i \in W, \forall E \in E_{BC_i}, \forall t_x \in base(T_{CH})) : (TI_E \cap (\cup_x \{TI_{t_x} \mid \neg CL_{BC_i}(E) \in I_{t_x}\}) = \emptyset)$.

(3.a) $(\forall W_i \in W, \forall E \in E_{IC_i}, \forall t_x \in base(T_{CH})) : (TI_E \cap (\cup_x \{TI_{t_x} \mid Preserve(t_x, L_{IC_i}(E)) = 0\}) = \emptyset)$.

(3.b) $(\forall W_i \in W, \forall E \in E_{IC_i}, \forall t_x \in base(T_{CH})) : (((Preserve(t_x, L_{IC_i}(E)) = 1/2) \wedge (TI_E \cap TI_{t_x} \neq \emptyset)) \Rightarrow (St_{end(t_x)} \models L_{IC_i}(E)))$. □

In the following, these conditions are explained to clarify them.

(1) Basic constraints (i.e., $\mathcal{B} \equiv \wedge_i \mathcal{B}_i$, where $\mathcal{B}_i \in B$) should hold when complete execution history (CH) is started (i.e., when its first activity, f_{CH}, is started).

(2) If $E = \langle t_j, VS = \{t_k, t_l, ...\}, CL_{BC_i}(E) = \neg\mathcal{B}_n, VL_{BC_i}(E) = and/or\rangle$ is an edge in BC_i (where BC_i a basic constraints graph of a workflow $W_i \in W$), and if $\neg CL_{BC_i}(E) = \mathcal{B}_n$ is involved in the input condition of another activity t_x (i.e., $\mathcal{B}_n \in I_{t_x}$), time intervals associated with E (TI_E) and t_x (TI_{t_x}) should not intersect.

(3.a) If $E = \langle t_j, \{t_k, t_l, ...\}, L_{IC_i}(E) = \mathcal{F}\rangle$ is an edge in IC_i (where IC_i is an inter-activity constraints graph of a workflow $W_i \in W$), and if another activity t_x falsifies $\mathcal{F}$ (i.e., $Preserve(t_x, \mathcal{F}) = 0$), TI_E and TI_{t_x} should not intersect.

(3.b) If $E = \langle t_j, \{t_k, t_l, ...\}, L_{IC_i}(E) = \mathcal{F}\rangle$ is an edge in IC_i, and t_x *may* falsify $\mathcal{F}$ (i.e., $Preserve(t_x, \mathcal{F}) = 1/2$), $\mathcal{F}$ should be still valid when t_x is terminated. Notice that, if t_x does not participate in CH (e.g., by removing it from CH), this condition automatically holds.

Proof: To prove this theorem, we show that if the conditions stated in Theorem 5.1 are true, the conditions in the definition of a correct complete execution history (i.e., Definition 5.11) hold.

(1) As a first step, it is proved that $(\forall t \in base(T_{CH})) : (St_{start(t)} \models \mathcal{I}_t)$ is true. Assume that $(\exists t_x \in base(T_{CH})) : (St_{start(t_x)} \not\models \mathcal{I}_{t_x})$. To achive this, at least one of the conditions below should hold:

- $St_{start(t_x)} \not\models \mathcal{B}_i$, where $\mathcal{B}_i \in B(t_x)$.
- $St_{start(t_x)} \not\models \mathcal{F}_j$, where $\mathcal{F}_j \in C_{in}(t_x)$.
- $St_{start(t_x)} \not\models \mathcal{G}_k$, where $\mathcal{G}_k \in G(t_x)$.

Remember that the constraints constituting an input condition are the elements of $B(t_x) \cup C_{in}(t_x) \cup G(t_x)$ (Formula 11). Trivially, Condition 2 of Theorem 5.1 prevents first case; second case is not possible due to Conditions 3.a and 3.b. It is guaranteed that the last case does not occur by workflow design.

(2) In this step, it is proved that $(St_{start(f_{CH})} \models \mathcal{B}) \Rightarrow (St_{end(l_{CH})} \models \mathcal{B})$ holds. First part of the formula is true by assumption (i.e., Condition 1 of Theorem 5.1). Assume that $St_{end(l_{CH})} \not\models \mathcal{B}$; to achieve this $\mathcal{O}_{t_x} \not\Rightarrow \mathcal{B}$ should hold for a $t_x \in base(T_{CH})$. In this case, however, activities of an and/or-validating set are present in CH due to validation completeness property (Definition 5.9). It has been already proved that validity of input conditions of activities in a validating set are guaranteed. Thus, $\mathcal{B}$ is certainly validated prior to the termination of CH by these activities.

Thus, if the conditions of Theorem 5.1 are true, correctness of CH is guaranteed. □

6. Constraint Based Concurrency Control (CBCC) Mechanism

In this section, a *Constraint Based Concurrency Control (CBCC)* mechanism for workflows based on the correctness notion developed in Section 5 is proposed.

In Section 5 it is shown that, if the conditions of Theorem 5.1 hold, correctness of a complete execution history of workflows is guaranteed. Validity of these conditions can indeed be guaranteed through a *Constraint Based Concurrency Control* mechanism to control activity interleavings in such a way that inter-activity constraints are preserved and accesses to workflow environment on which the basic constraints do not hold are prevented. In this mechanism, activities acquire and release locks on inter-activity and basic constraints in two different modes, and certain inter-activity constraints are evaluated within an activity. To achieve this, CBCC mechanism employees three stages for the execution of an activity: (1) Locking stage before the actual execution of an activity; (2) Certification (evaluation) stage before the actual termination of an activity; (3) Lock releasing stage after an activity terminates. Activities acquire locks on the relevant constraints in the locking stage by issuing lock requests to CBCC mechanism. The lock compatibility table for inter-activity and basic constraints is given in Table 2. "Y" means that the locks do not conflict and "N" means the locks conflict.

An inter-activity constraint $\mathcal{F}$ can be locked by an activity t_x in one of the following modes:

- **Shared:** This mode of lock is acquired when t_x intends to preserve $\mathcal{F}$ until a set of other activities terminate, i.e., $\mathcal{F} \in C_{out}(t_x)$.
- **Exclusive:** This mode is used when t_x falsifies $\mathcal{F}$, i.e., $Preserve(t_x, \mathcal{F}) = 0$. All inter-activity constraints in a workflow management system which are falsified

Table 2. The Lock Compatibility Table for Inter-activity and Basic Constraints

Mode	Existing	
Requested	Shared	Exclusive
Shared	Y	N
Exclusive	N	Y

by t_x constitute the set $F(t_x)$. Note that not only inter-activity constraints within a workflow in which t_x resides, but also all inter-activity constraints of other workflows are considered for this set.

If $\mathcal{F}$ is to be preserved in the interval between activity t_j and a set of activities $\{t_k, t_l, ...\}$, and if another activity t_x that falls in this interval falsifies $\mathcal{F}$, t_x should be delayed until $\mathcal{F}$ is unlocked by the every activity in $\{t_k, t_l, ...\}$. Therefore, the shared lock taken by t_j conflicts with the exclusive lock taken by t_x, as indicated in Table 2. Furthermore if $\mathcal{F}$ is to be preserved in the interval between activities t_j and $\{t_k, t_l, ...\}$, and again $\mathcal{F}$ is to be preserved in another interval between t_m and $\{t_n, t_o, ...\}$, both t_j and t_m lock $\mathcal{F}$ in shared mode and clearly there is no need for these shared locks to be in conflict, as indicated in Table 2. Note that we use the term "exclusive lock" differently than its conventional meaning in that, two exclusive locks on the same constraint do not conflict with each other in our approach as opposed to traditional exclusive locks.

It should be noted that some of the inter-activity constraints *may be* falsified by t_x, i.e., $Preserve(t_x, \mathcal{F}) = 1/2$, which constitute the set $LF(t_x)$. For the activities that may falsify inter-activity constraints, we prefer to use an optimistic scheme rather than locking with the intention of increasing the performance, since there is a probability that the activity will not falsify these constraints. If a constraint in this set is already locked in shared mode to be maintained when t_x is executed, this constraint is evaluated in the certification stage and if it evaluates to false, t_x is rolled back and resubmitted to workflow management system.

A basic constraint $\mathcal{B}$ can be locked by t_x in one of the following modes:

- **Shared:** If t_x requires the correctness of $\mathcal{B}$, i.e., $\mathcal{B} \in B(t_x)$, a shared lock is acquired.
- **Exclusive:** If t_x invalidates (or may invalidate) $\mathcal{B}$, i.e., $\mathcal{B} \in (\cup_{VS} SB_{\{t_x, VS, and/or\}})$, an exclusive lock is required.

An activity t_x (may) falsify a basic constraint $\mathcal{B}$ to be revalidated by the activities of and/or-validating sets as explained in Section 5. Therefore the activities that require the correctness of $\mathcal{B}$ in this interval should not be allowed to execute. For this reason, t_x obtains an exclusive lock on $\mathcal{B}$. On the other hand the activity that requires the correctness of $\mathcal{B}$ acquires a shared lock. The shared lock conflicts with the exclusive lock as indicated in Table 2. It is clear that the activities that require correctness of $\mathcal{B}$ do not conflict with each other.

6.1. CBCC Algorithms

In this section, the algorithms employed by CBCC mechanism are described. In these algorithms, data structures IC, BC for every workflow, and $B(t_x)$, $F(t_x)$, $LF(t_x)$ for every activity are required. A *Constraint Editor* in conjunction with a first-order constraint specification language [11, 14] can be used by an administrator and/or workflow designers to define these data structures.

6.1.1. Algorithm for Activity Start

Any activity t_x needs an exclusive lock for every inter-activity constraint it falsifies to start (*Steps 1-2* of *Algorithm 6.1*). This is possible only when there is no other activity that has a shared lock on $\mathcal{F}$; in other words no other activity wants to preserve $\mathcal{F}$. Furthermore, t_x also needs to acquire shared locks for all the basic constraints involved in its input condition (i.e., $B(t_x)$) (*Steps 3-4*). A lock for a constraint $\mathcal{B}$ in $B(t_x)$ is granted to t_x if there is no invalidating activity that has an exclusive lock on $\mathcal{B}$. After this step, every inter-activity constraint emanating from t_x in the inter-activity constraints graph (IC) (i.e., elements of $C_{out}(t_x)$) are locked in the shared mode in *Steps 5-6*. t_x can acquire a shared lock on $\mathcal{F} \in C_{out}(t_x)$ if no other invalidating activity for $\mathcal{F}$ has an exclusive lock on $\mathcal{F}$. Recall that $\mathcal{F}$ may be incident to more than one activity, and these activities are grouped into a hyperSet $S_{(t_x,\mathcal{F})}$. This is represented by the edge $\langle t_x, S_{(t_x,\mathcal{F})}, \mathcal{F}\rangle$ in IC. Since $\mathcal{F}$ should be preserved until the termination of all the activities in the hyperSet $S_{(t_x,\mathcal{F})}$, it is necessary to obtain a shared lock for each of the activities in this set, i.e., $size(S_{(t_x,\mathcal{F})})$ locks are acquired. A conflicting lock can then only be allowed when all these locks are released. In *Steps 7-10*, exclusive locks are acquired on the basic constraints which are invalidated by t_x which is only possible if there are no shared locks on $\mathcal{B}$. That is, since t_x is invalidating $\mathcal{B}$, there should not exist any activity that requires the correctness of $\mathcal{B}$. If VS is an and-validating set for $\mathcal{B}$ and if it contains more than one activity, t_x acquires an exclusive lock on $\mathcal{B}$ for each activity of VS, that is the number of locks acquired is $size(VS)$. If VS is an or-validating set, t_x acquires a single lock since the termination of the first activity of VS guarantees validity of $\mathcal{B}$.

Inter-activity constraints which *may* be falsified by t_x, i.e., $LF(t_x)$ are handled in an optimistic manner. Note that all the constraints in $LF(t_x)$ may not be active, that is, it may be the case that for some constraints in $LF(t_x)$, there is no activity requiring these constraints to hold. We include all the active constraints in $ActiveICS$ set and all the constraints in this set are already locked in the shared mode. The intersection of $LF(t_x)$ and $ActiveICS$ sets gives us the set of constraints denoted as $ALF(t_x)$, that are both active when t_x has started and also has to be validated when t_x terminates (*Step 11*). Since new shared locks can be acquired on the elements of $LF(t_x) - ActiveICS$ by other activities before the activity terminates, constraints in $PLF(t_x) = LF(t_x) - ActiveICS$ (i.e., non-active constraints which are in $LF(t_x)$) are locked in exclusive mode. Furthermore, operations in

Step 11 are executed atomically (i.e., in a critical section). In this way, further constraints that may be falsified by t_x are prevented from becoming active after the set of constraints that will be validated are determined.

Algorithm 6.1 [Algorithm for Activity Start]

---

begin
1. **for** *every* $\mathcal{F} \in F(t_x)$ **do**
2. $\quad$ *ExclusiveLock*($\mathcal{F}$)*;*
3. **for** *every* $\mathcal{B} \in B(t_x)$ **do**
4. $\quad$ *SharedLock*($\mathcal{B}$)*;*
5. **for** *every* $\mathcal{F} \in C_{out}(t_x)$ **do**
6. $\quad$ *SharedLock*($\mathcal{F}$) *with* $Counter = size(S_{(t_x,\mathcal{F})})$*;*
7. **for** *every* $\mathcal{B} \in (\cup_{VS} SB_{\{t_x,VS,and\}})$ **do**
8. $\quad$ *ExclusiveLock*($\mathcal{B}$) *with* $Counter = size(VS)$*;*
9. **for** *every* $\mathcal{B} \in (\cup_{VS} SB_{\{t_x,VS,or\}})$ **do**
10. $\quad$ *ExclusiveLock*($\mathcal{B}$)*;*
11. $\left[\begin{array}{l} ALF(t_x) \leftarrow (LF(t_x) \cap ActiveICS); \\ PLF(t_x) \leftarrow (LF(t_x) - ActiveICS); \\ \textbf{for } every\ \mathcal{F} \in PLF(t_x)\ \textbf{do} \\ \quad ExclusiveLock(\mathcal{F}) \end{array}\right]$

end

After successfully acquiring all the necessary locks as indicated in the *Algorithm 6.1*, an activity can be scheduled for execution.

6.1.2. *Algorithm for Activity End*

An activity terminates when all of its operations are complete. But prior to termination, an evaluation algorithm (*Algorithm 6.2*) is executed to check whether an active inter-activity constraint is falsified by the execution of this activity. This is achieved in *Step 1* by evaluating the constraints in $ALF(t_x)$ in parallel by the routine $EvalInParallel$; once a constraint evaluates to $false$, $EvalInParallel$ terminates immediately and returns $false$. In this case, the activity t_x is rolled backed and resubmitted to workflow management system. Note that all the locks acquired by t_x should be released. If $ALF(t_x)$ is empty, *Algorithm 6.2* is not executed.

Algorithm 6.2 [Algorithm for Activity End]

---

begin
1. **if** (*EvalInParallel*($ALF(t_x)$) $= false$) **then**
2. $\quad$ *Rollback*(t_x), *Resubmit*(t_x)

end

6.1.3. *Algorithm For Activity Post-Processing*

After an activity t_x is terminated, all locks acquired by t_x on the constraints in $PLF(t_x)$, $F(t_x)$, and $B(t_x)$ are released in *Steps 1-2, 3-4*, and *5-6* of *Algorithm 6.3* respectively. Inter-activity constraints incident to t_x (i.e., $C_{in}(t_x)$) which are locked by other activities are released in *Steps 7-8*. If t_x is in an and-validating set (VS) of a basic constraint $\mathcal{B}$, one of the previously acquired exclusive locks by the invalidating activity of $\mathcal{B}$ is released in *Steps 9-10*. If t_x is the first terminating activity of an or-validating set, a corresponding lock is released *Steps 11-12*.

Algorithm 6.3 [Algorithm for Activity Post-Processing]

---

begin
1. **for** *every* $\mathcal{F} \in PLF(t_x)$ **do**
2. *Unlock*($\mathcal{F}$);
3. **for** *every* $\mathcal{F} \in F(t_x)$ **do**
4. *Unlock*($\mathcal{F}$);
5. **for** *every* $\mathcal{B} \in B(t_x)$ **do**
6. *Unlock*($\mathcal{B}$);
7. **for** *every* $\mathcal{F} \in C_{in}(t_x)$ **do**
8. *Unlock*($\mathcal{F}$);
9. **for** *every* $\mathcal{B} \in (\cup_{t_i} SB_{\{t_i, VS, and\}})$ *where* $t_x \in VS$ *or* $t_x = VS$ **do**
10. *Unlock*($\mathcal{B}$);
11. **for** *every* $\mathcal{B} \in (\cup_{t_i} SB_{\{t_i, VS, or\}})$ *where* $t_x = first(VS)$ **do**
12. *Unlock*($\mathcal{B}$)
end

6.2. Correctness of the CBCC Mechanism

To prove that a complete execution history (CH) generated by CBCC mechanism is correct we show that the conditions of Theorem 5.1 hold for CH. The following properties about time intervals are used in the proof. Note that $\supset$ and $\cap$ denote cover and intersect relations between the time intervals respectively.

- $((TI_i \supset TI_j) \wedge (TI_j \cap TI_k \neq \emptyset)) \Rightarrow (TI_i \cap TI_k \neq \emptyset)$.
- $((TI_i \supset TI_j) \wedge (TI_i \cap TI_k = \emptyset)) \Rightarrow (TI_j \cap TI_k = \emptyset)$.

Theorem 6.1 Any complete execution history (CH) generated by CBCC mechanism is correct.

Proof:

(1) Condition 1 of Theorem 5.1 holds due to the assumption.

(2) Assume that Condition 2 of Theorem 5.1 does not hold; hence $TI_E \cap TI_{t_x} \neq \emptyset$ in CH for an edge $E = \langle t_j, VS = \{t_k, t_l, ...\}, \neg\mathcal{B}_n, and/or\rangle$ in BC_i, and an activity t_x where $\mathcal{B}_n \in B(t_x) \subseteq I_{t_x}$. The interval between the time when an

exclusive lock on $\mathcal{B}_n$ is acquired with counter by t_j and the time when the last of these locks are released is denoted as $TI_E^{XL(\mathcal{B}_n)}$ in the case where VS is an and-validating set. Same notation is used to denote the interval between the time instances where a single lock is acquired by t_j and released by the first activity of an or-validating set VS. Similarly, the interval between the time when a shared lock is acquired and released on $\mathcal{B}_n$ by t_x is denoted as $TI_{t_x}^{SL(\mathcal{B}_n)}$. Since activities acquire locks before they start and release after they complete, $TI_E^{XL(\mathcal{B}_n)} \supset TI_E$ and $TI_{t_x}^{SL(\mathcal{B}_n)} \supset TI_{t_x}$. Since exclusive and shared locks on a basic constraint conflict, it is guaranteed that $TI_E^{XL(\mathcal{B}_n)} \cap TI_{t_x}^{SL(\mathcal{B}_n)} = \emptyset$. Yet, due to first property above $((TI_E^{XL(\mathcal{B}_n)} \supset TI_E) \wedge (TI_E \cap TI_{t_x} \neq \emptyset)) \Rightarrow (TI_E^{XL(\mathcal{B}_n)} \cap TI_{t_x} \neq \emptyset)$. Furthermore, according to second property, $((TI_{t_x}^{SL(\mathcal{B}_n)} \supset TI_{t_x}) \wedge (TI_{t_x}^{SL(\mathcal{B}_n)} \cap TI_E^{XL(\mathcal{B}_n)} = \emptyset)) \Rightarrow (TI_{t_x} \cap TI_E^{XL(\mathcal{B}_n)} = \emptyset)$. Observe that the right hand sides of two formulas contradict each other; hence our presumption is false and Condition 2 of Theorem 5.1 holds.

(3.a) We start with proving that if $Preserve(t_x, \mathcal{F}) = 0$ then $TI_E \cap TI_{t_x} = \emptyset$ is guaranteed in CH for an edge $E = \langle t_j, \{t_k, t_l, ...\}, \mathcal{F}\rangle$ in IC_i. We denote the interval between the time when a shared lock on $\mathcal{F}$ is acquired with counter by t_j and the time when the last of these locks are released as $TI_E^{SL(\mathcal{F})}$. Similarly, the interval between the time when an exclusive lock is acquired and released on $\mathcal{F}$ by t_x is denoted as $TI_{t_x}^{XL(\mathcal{F})}$. Again, $TI_E^{SL(\mathcal{F})} \supset TI_E$ and $TI_{t_x}^{XL(\mathcal{F})} \supset TI_{t_x}$. Since exclusive and shared locks on an inter-activity constraint conflict, it is ensured that $TI_E^{SL(\mathcal{F})} \cap TI_{t_x}^{XL(\mathcal{F})} = \emptyset$. With the similar observations as in Condition 2 of this proof, Condition 3.a of Theorem 5.1 holds.

(3.b) We conclude with proving that if $Preserve(t_x, \mathcal{F}) = 1/2$, $TI_E \cap TI_{t_x} \neq \emptyset$ implies $\mathcal{F}$ holds after t_x is terminated. Depending on the execution sequences of t_j and t_x two possibilities can occur:

- t_j acquires a shared lock on $\mathcal{F}$ before t_x acquires an exclusive lock on $\mathcal{F}$: $\mathcal{F}$ is certainly logged into $ALF(t_x)$ and if t_x falsifies $\mathcal{F}$, $EvalInParallel(ALF(t_x))$ returns $false$ and t_x is removed from CH (i.e., rolled backed); hence $TI_E \cap TI_{t_x} = \emptyset$.
- t_x acquires an exclusive lock on $\mathcal{F}$ before t_j acquires a shared lock on $\mathcal{F}$: t_j can not lock $\mathcal{F}$ in shared mode after *Step 11* of *Algoritm 6.1* and before t_x terminates, since t_x already locked $\mathcal{F}$ in exclusive mode in *Step 11*. Hence $TI_E \cap TI_{t_x} = \emptyset$.

Thus, a complete execution history generated by CBCC mechanism is correct. □

6.3. Discussion

There are several alternatives to implement a constraint based concurrency control mechanism. In the following, some of these alternatives are discussed:

- *Conservative*: In this approach, activities that are certainly or likely to falsify basic and inter-activity constraints are determined in advance (i.e., in design-time), and possible invalidations of inter-activity constraints and accesses to states on which the basic constraints do not (or may not) hold are prevented conservatively. For example, proposed CBCC mechanism can be classified into this category if activities try to acquire locks on the inter-activity constraints which they may falsify in addition to constraints which they certainly falsify in *Steps 1-2* of *Algorithm 6.1.* Also *Step 11* of *Algorithm 6.1*, and *Algorithm 6.2* become unnecessary in this case. Since this conservative technique is based solely on locking, we call it as the *Constraint Locking Concurrency Control* (*CLCC*) mechanism. In CLCC mechanism, constraints themselves are no longer necessary, but can be represented through some simple data items just for locking purposes. It should also be noted that, if such a technique is not implemented in a workflow system, it is possible to acquire locks manually on virtual data items using the same principles.
- *Optimistic*: In this approach, activities validate their input conditions. This requires additional operations for the verification of these conditions. Optimistic technique is very similar to concurrency control mechanism of ConTract model [60]; however the input conditions we check are well-defined interms of inter-activity and basic constraints. If input condition of an activity evaluates to false, a conflict resolution algorithm can be executed to correct the input condition violation or to relax the requirements in the input condition. An inevitable result may be abortion of the activity and compensation of some previously terminated activities.
- *Dynamic-conservative*: The approach employed by the CBCC mechanism can be classified into this category.

In the optimistic technique, if conflict resolution algorithm requires rollback of the activity this may cause (possibly cascading) compensation of previously terminated activities which may be a very costly process [44, 52]. In addition, overhead of validation of every input condition should not be ignored. CLCC and CBCC techniques guarantee that input condition of an activity is true when it is executed; thus neither input condition validation nor compensation of other activities to resolve conflicts are required in these techniques. In addition, CBCC mechanism provides some activities to be executed and terminated if they pass certification process although these activities and consequently successor activities would be blocked by the CLCC mechanism. Furthermore, in the optimistic technique it is necessary to check the constraints themselves; however in CLCC mechanism these constraints can be represented by some simple data items just for locking purposes. In CBCC mechanism on the other hand, only the inter-activity constraints which may be falsified by the activities are needed in the validation phase. In Section 6.4, a comparison of the performance characteristics of these techniques is provided.

It should be noted that, proposed CBCC and CLCC mechanisms may result in deadlocks like any other locking-based concurrency control mechanism, since activities may be blocked indefinitely. Therefore, special algorithms are required to

handle deadlocks. There are three well known types of methods for handling deadlocks: prevention, avoidance, and detection and resolution [51]. We have developed a deadlock avoidance technique for CBCC and CLCC mechanisms in which potential deadlock situations are detected in advance (i.e., in design-time) and it is ensured that they will not occur at run-time by imposing additional restrictions on the interleavings of activities. Since concurrency control dependencies among activities are known in advance, possible deadlock situations can be detected in design-time in CBCC and CLCC mechanisms. Detailed explanation and formal foundation of this approach are presented in [6] due to space limitations.

6.4. Performance Analysis

In this section, a performance comparison of the CBCC, CLCC mechanisms and optimistic technique which is similar to concurrency control mechanism of ConTract model [60] is given. The simulation is realized in GPSS [57]. In the experiments, average response time of a workflow instance ($avgResTime$) is measured by averaging response times of 10 workflow instances. Response time is defined as the time between the generation and termination of a workflow instance.

In the simulation, there are a total of 10 different basic and inter-activity constraints in the system. It should be noted that, the total number of constraints are kept small so that the possibility of conflicts among activities is high. In this way, the performances of the methods can be observed in a very high conflict case. For each activity, the number of constraints that should be considered (i.e., locked or evaluated) is randomly chosen from the interval $[0 - maxCons]$ where $maxCons$ denotes the maximum number of constraints per activity and is given a priori. In the CLCC mechanism, each activity tries to obtain a lock on all of its constraints. Note that, some of the constraints which *may* be falsified by an activity are evaluated at the activity end instead of being locked in the CBCC mechanism. The evaluation cost per constraint is taken as constant for simplicity (i.e., 5 simulation time units). If a constraint evaluates to false the activity is aborted and restarted later. In the optimistic technique, the constraints are evaluated when the activity starts and once a constraint evaluates to false the activity is aborted and preceding activities are compensated. The result of the evaluation is randomly determined as true or false with the probability of 70% and 30% respectively. It should be noted that this fraction favors the optimistic technique rather than the CBCC mechanism, because in the CBCC mechanism a small fraction of constraints goes through the validation as opposed to all constraints in the optimistic method. Also in favor of the optimistic technique, the compensation cost is chosen as close to the maximum duration of just one activity, i.e., 50 simulation time units, although in reality this cost is much higher since compensation of more than one activity is more probable.

The graph in Figure 14 shows the average workflow instance response times ($avgResTime$) of three techniques for different maximum number of constraints per activity ($maxCons$). The experiment results can be summarized as follows:

All techniques provide their best $avgResTime$s when $maxCons$ is small, i.e., in

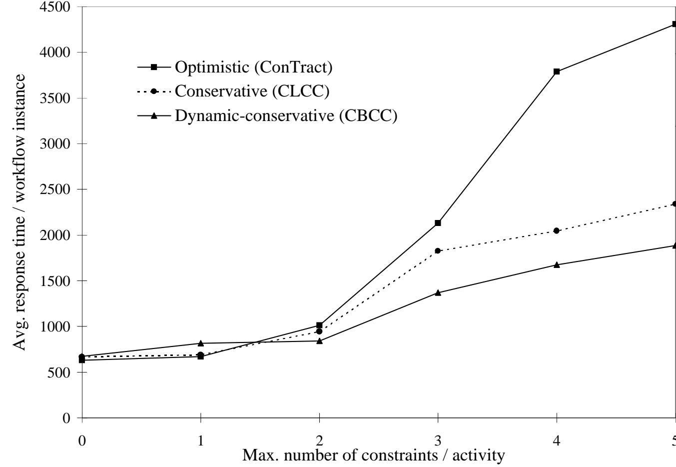


Figure 14. Average Response Times for Different Maximum Number of Constraints per Activity

$[0-2]$. This is expected since when $maxCons < 2$ the probability of conflicts among activities is low, and consequently the number of blocked or compensated activities is small.

When $maxCons \geq 2$, CBCC and CLCC techniques provide better $avgResTime$s than optimistic technique. For example, when $maxCons$ is equal to half of the total number of constraints in the system (e.g., around 5), $avgResTime$ provided by the optimistic technique becomes worse than two times of $avgResTime$ provided by CBCC mechanism, i.e., 1884 vs. 4306 simulation time units. This is due to fact that, the number of compensated activities increases in the optimistic technique with the increasing number of constraints ($maxCons$) which implies higher rate of conflicts. In CBCC mechanism, however, abortion of an activity does not lead to compensation of previous activities, only the activity itself is retried later.

When $maxCons = 2$, CBCC mechanism starts to perform better than CLCC mechanism. For example, when $maxCons \geq 3$, CBCC mechanism provides approximately 25% faster $avgResTime$ than CLCC technique. Since not all the constraints are locked in the CBCC mechanism, the probability of delays due to locking is lower than that of CLCC mechanism. This difference becomes more visible when $maxCons$ is larger.

Performance results presented indicate that the CBCC mechanism results in lower average workflow instance response times in almost all cases except when maximum number of constraints that should be considered per activity ($maxCons$) is very small (e.g., 1) or such a constraint does not exist. If $maxCons$ is small, $avgResTime$s provided by the compared techniques are almost the same.

After observing that the performance of the optimistic technique is not good in a high conflict case, additional experiments are conducted to compare the performances of CBCC and CLCC techniques for different evaluation costs. These experiment results are presented in [6].

7. Conclusions

Concurrency control aspects of workflow systems is addressed in this work, which is very important for some workflow applications where mission critical operations require the consistent view of the execution environment [22].

The fundamental issue of correctness criterion specific to workflow systems is defined through inter-activity constraints and basic constraints by using the semantic workflow information available at design-time. A concurrency control technique, namely Constraint Based Concurrency Control (CBCC) mechanism, based on this criterion is defined which uses the concept of locking in conjunction with validation with a fundamental difference from the database locking: the constraints rather than data items are locked. We have shown that, with a proper constraint locking and validation mechanism, the inter-activity constraints that should remain valid are preserved, and the activities that need basic constraints to hold are prevented from executing in the intervals where these constraints do not hold. It is also possible to use a more conservative approach in which the activities acquire locks instead of going through a validation phase. We call this technique as Constraint Locking Concurrency Control (CLCC) mechanism. These techniques are simple to implement, and the performance analysis indicate that the suggested techniques have better performance than an optimistic approach based on the constraints (similar to ConTract [60]). Note that when a workflow designer does not require the correctness to be preserved, some of the constraints may not be enforced. In this respect, it is possible to apply an isolation mechanism similar to isolation levels in databases [33] by allowing the workflow designer to customize the constraints graphs according to the correctness requirements of workflow application. For these reasons, we believe that the CBCC and CLCC techniques have practical importance.

References

1. D. Agrawal, A. E. Abbadi, and A. K. Singh, *Consistency and Orderability: Semantics-Based Correctness Criteria for Databases*, ACM TODS, Vol. 18, No. 3, September 1993.
2. J. F. Allen, *Maintaining Knowledge about Temporal Intervals*, Comm. of the ACM, 26, 11, November 1983.
3. G. Alonso, D. Agrawal, and A. E. Abbadi, *Process Synchronization in Workflow Management Systems*, 8th IEEE Symposium on Parallel and Distributed Processing, 1996.
4. G. Alonso, and H.-J. Schek, *Research Issues in Large Workflow Management Systems*, In. Proc. of NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions, Edited by A. Sheth, Athens, Georgia, May 1996.
5. P. Ammann, S. Jajodia, and I. Ray, *Applying Formal Methods to Semantic-Based Decomposition of Transactions*, ACM TODS, Vol. 22, No. 2, June 1997.
6. I. B. Arpinar, *Concurrency Control and Transaction Management in Workflow Management Systems*, Ph. D. Thesis, in preparation, Dept. of Computer Engineering, Middle East Technical University, 1998.
7. I. B. Arpinar, S. (Nural) Arpinar, U. Halici, and A. Dogac, *Correctness of Workflows in the Presence of Concurrency*, NGITS' 97, Next Generation Information Technologies and Systems, Israel, July 1997.
8. P. A. Attie, M. P. Singh, A. Sheth, and M. Rusinkiewicz, *Specifying and Enforcing Intertask Dependencies*, In Proc. of the 19th Intl. Conf on VLDB, September 1993.

9. B. Badrinath, and K. Ramamritham, *Semantics-based Concurrency Control: Beyond Commutativity*, In Proc. of Intl. Conf. on Data Engineering, February 1987.
10. C. Beeri, P. A. Bernstein, and N. Goodman, *A Model for Concurrency in Nested Transaction Systems*, Journal of the ACM, 36(2), 1989.
11. M. Benedikt, T, Griffin, and L. Libkin, *Verifiable Properties of Database Transactions*, ACM PODS 1996, Montreal, Canada.
12. A. J. Bernstein, and P. M. Lewis, *Transaction Decomposition Using Transaction Semantics*, Distributed and Parallel Databases, 4, 25-47, 1996.
13. Y. Breitbart, A. Deacon, H. J. Schek, A. Sheth, and G. Weikum, *Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows*, ACM SIGMOD Record, 22(3), Sept. 1993.
14. S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca, *Automatic Generation of Production Rules for Integrity Maintenance*, ACM TODS, Vol. 19, No. 3, September 1994.
15. U. Dayal, and M.-C. Shan, *Issues in Operation Flow Management for Long-Running Activities*, Data Eng. Bulletin, June 1993.
16. U. Dayal, H. Garcia-Molina, M. Hsu, B. Kao, and M.-C. Shan, *Third Generation TP Monitors: A Database Challenge*, SIGMOD, 1993.
17. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
18. A. Dogac, E. Gokkoca, S. Arpinar, P. Koksal, I. Cingil, I. B. Arpinar, N. Tatbul, P. Karagoz, U. Halici, M. Altinel, *Design and Implementation of a Distributed Workflow Management System: METUflow*, In: [19].
19. A. Dogac, L. Kalinichenko, M. T. Ozsu, and A. Sheth (eds.), *Advances in Workflow Management Systems and Interoperability*, Springer Verlag, 1998.
20. J. A. Ellis, and G. J. Nutt, *Modeling and Enactment of Workflow Systems*, 14th Intl. Conf. on Application and Theory of Petri Nets, 1993.
21. A. K. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, San Mateo, 1992.
22. A. Elmagarmid, and W. Du, *Workflow Management: State of the Art vs. State of the Market*, In: [19].
23. E. A. Emerson, *Temporal and Modal Logic*, In: J. van Leeuwen (ed.), Handbook of Theoretical Computer Science, Elseiver 1990.
24. A. Farrag, and M. T. Ozsu, *Using Semantic Knowledge of Transactions to Increase Concurrency*, ACM TODS, Vol. 14, No. 4, December 1989.
25. M. Fitting, *First Order Logic and Automated Theorem Proving*, Springer Verlag, NY, 1990.
26. H. Garcia-Molina, *Using Semantic Knowledge for Transaction Processing in a Distributed Database*, ACM TODS, Vol. 8, No. 2, June 1983.
27. H. J. Genrich, *Predicate/Transition Nets*, In Advances in Petri Nets, 1986, Springer, 254.
28. D. Georgakopoulos, M. Hornick, and F. Manola, *Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation*, IEEE Trans. on Knowledge and Data Eng., 1995.
29. D. Georgakopoulos, M. Hornick, and A. P. Sheth, *An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure*, Distributed and Parallel Databases, 3, pp. 119-153, 1995.
30. D. Georgakopoulos, M. Rusinkiewicz, and A. P. Sheth, *Using Tickets to Enforce the Serializability of Multidatabase Transactions*, IEEE TKDE, 6(1), 1994.
31. E. Gokkoca, M. Altinel, I. Cingil, N. Tatbul, P. Koksal, A. Dogac, *Design and Implementation of a Distributed Workflow Enactment Service*, in Proc. of Intl. Conf on Cooperative Information Systems, Charleston, USA, June 1997.
32. J. Gray, *The Transaction Concept: Virtues and Limitations*, In Proc. of 7th Intl. Conf. on VLDB, Cannes, France, September 1981.
33. J. Gray, and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.
34. U. Halici, I. B. Arpinar, and A. Dogac, *Serializability of Nested Transactions in Multidatabases*, Intl. Conf. on Database Theory (ICDT '97), Greece, January 1997.
35. T. Harder, *Handling Hot Spot Data in DB-sharing Systems*, Info. Sys. , 13, 2, 1988.

36. D. Harel, et al., *Statemate: A Working Environment for the Development of Complex Reactive Systems*, IEEE Transactions on Software Engineering, Vol. 16, No. 4, April 1990.
37. M. P. Herlihy, and W. E. Weihl, *Hybrid Concurrency Control for Abstract Data Types*, J. Comput. Syst. Sci., Vol. 43, No.1, August 1991.
38. C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, Commun. ACM, 12, 10, October 1969.
39. D. Hollinsworth, *The Workflow Reference Model*, Technical Report TC00-1003, Workflow Management Coalition, December 1994. Accessible via: http://www.aiai.ed.ac.uk/WfMC/.
40. M. Hsu, *Special Issue on Workflow Systems*, Bulletin of the Technical Committee On Data Engineering, IEEE, 18(1), March 1995.
41. G. Kappel, P. Lang, S. Rausch-Schott, and W. Retschitzegger, *Workflow Management Based on Objects, Rules, and Roles*, In: [40].
42. P. Karagoz, S. Arpinar, P. Koksal, N. Tatbul, E. Gokkoca, and A. Dogac, *Task Handling in Workflow Management Systems*, In Proc. of Intl. Workshop on Issues and Applications of Database Technology, IADT'98, Berlin, June 1998.
43. P. Koksal, S. Arpinar, and A. Dogac, *Workflow History Management*, ACM Sigmod Record, Vol. 27, No. 1, March 1998.
44. H. F. Korth, E. Levy, and A. Siberschatz, *A Formal Approach to Recovery by Compensating Transactions*, In Proc. of the 16th VLDB Conf., Brisbane, Australia, 1990.
45. H. F. Korth, and G. Speegle, *Formal Aspects of Concurrency Control in Long-Duration Transaction Systems Using the NT/PV Model*, ACM TODS, Vol. 19, No. 3, 1994.
46. N. Krishnakumar, and A. Sheth, *Managing Heterogeneous Multi-System Tasks to Support Enterprise-Wide Operations*, Distributed and Parallel Databases, 3(2):155-186, April 1995.
47. N. A. Lynch, *Multilevel Atomicity: A New Correctness for Database Concurrency Control*, ACM TODS, Vol. 8, No. 4, pp. 484-502, Dec. 1983.
48. P. Muth, D. Wodtke, J. WeiBenfels, G. Weikum, and A. K. Dittrich, *Enterprise-wide Workflow Management based on State and Activity Charts*, In: [19].
49. P. E. O'Neil, *The Escrow Transaction Method*, ACM TODS, Vol. 11, No. 4, December 1986.
50. *Object Transaction Service*, OMG Document, 1994.
51. M. T. Ozsu, and P. Valduriez, *Principles of Distributed Database Systems*, 2nd edition, Prentice Hall, Englewood Cliffs, New Jersey, 1998.
52. M. Rusinkiewicz, A. Cichocki, A. Sheth, and G. Thomas, *Bounding the Effects of Compensation under Multi-level Serializability*, Dist. and Parallel Databases, 4(4), Oct. 1996.
53. M. Rusinkiewicz, and A. P. Sheth, *Transactional Workflow Management Systems*, In Proc. of Advances in Database and Information Systems, ADBIS'94, Moscow, May 1994.
54. M. Rusinkiewicz, and A. P. Sheth, *Specification and Execution of Transactional Workflows*, W. Kim, editor, Modern Database Systems: The Object Model, Interoperability and Beyond, pp. 592-620, ACM Press, New York, NY, 1995.
55. F. Schwenkreis, *A Formal Approach to Synchronize Long-Lived Computations*, In Proc. of the 5th Australasian Conf. in Information Systems, Melbourne 1994.
56. A. Sheth, D. Georgakopoulos, S. M.M. Joosten, M. Rusinkiewicz, W. Scacchi, J. Wileden, and A. Wolf, *Report from the NSF Workshop on Workflow and Process Automation in Information Systems*, Accessible via: http://lsdis.cs.uga.edu/activities/.
57. I. Stahl, *Introduction to Simulation with GPSS*, Prentice Hall, 1990.
58. J. Tang, and J. Veijalainen, *Transaction-oriented Workflow Concepts in Inter-organization Environments*, Intl. Conf. on Information and Knowledge Management, Baltimore, 1995.
59. J. Tang, and S.-Y. Hwang, *Handling Uncertainty in Workflow Applications*, In Proc. of 5th Intl. Conf on Info. and Knowledge Engineering, CIKM'96, Maryland, November, 1996.
60. H. Waechter, and A. Reuter, *The ConTract Model*, In: [21], Chapter 7.
61. G. Weikum, *Principles and Realization Strategies of Multilevel Transaction Management*, ACM TODS, Vol. 16, No. 1, 1991.
62. D. Wodtke, and G. Weikum, *A Formal Foundation for Distributed Workflow Management Based on State Charts*, In Proc. of 6th Intl. Conf on Database Theory, Greece, Jan. 1997.
63. D. Worah, and A. Sheth, *What do Advanced Transaction Models Have to Offer for Workflows?*, In Proc. of Intl. Workshop on Advanced Transaction Models and Architectures (ATMA), Goa, India, August 1996.