Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮
www.elsevier.com/locate/infosys

Specification and validation of process constraints for flexible workflows ☆

Shazia W. Sadiq[a,*], Maria E. Orlowska[a], Wasim Sadiq[b]

[a] School of Information Technology and Electrical Engineering, The University of Queensland, Qld 4072, Australia
[b] Corporate Research, SAP Australia Pty Ltd, Brisbane, Australia

Received 12 May 2003; received in revised form 3 December 2003; accepted 13 May 2004

Abstract

Workflow systems have traditionally focused on the so-called production processes which are characterized by pre-definition, high volume, and repetitiveness. Recently, the deployment of workflow systems in non-traditional domains such as collaborative applications, e-learning and cross-organizational process integration, have put forth new requirements for flexible and dynamic specification. However, this flexibility cannot be offered at the expense of control, a critical requirement of business processes.

In this paper, we will present a foundation set of constraints for flexible workflow specification. These constraints are intended to provide an appropriate balance between flexibility and control. The constraint specification framework is based on the concept of "pockets of flexibility" which allows ad hoc changes and/or building of workflows for highly flexible processes. Basically, our approach is to provide the ability to execute on the basis of a partially specified model, where the full specification of the model is made at runtime, and may be unique to each instance.

The verification of dynamically built models is essential. Where as ensuring that the model conforms to specified constraints does not pose great difficulty, ensuring that the constraint set itself does not carry conflicts and redundancy is an interesting and challenging problem. In this paper, we will provide a discussion on both the static and dynamic verification aspects. We will also briefly present Chameleon, a prototype workflow engine that implements these concepts.

Keywords: Flexible workflows; Process modeling; Workflow verification

1. Introduction

Workflow technology is being applied in quite diverse areas. This diversity has had a strong impact on workflow research. We can find in the literature, several categories of workflow types, such as production, collaborative, ad hoc, etc. To determine the suitability of workflow technology,

☆ Recommended by Prof. Gottfried Vossen.

*Corresponding author.
E-mail addresses: shazia@itee.uq.edu.au (S.W. Sadiq), maria@itee.uq.edu.au (M.E. Orlowska), wasim.sadiq@sap.com (W. Sadiq).

0306-4379/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.is.2004.05.002

process characteristics such as functional complexity, predictability and repetitiveness are often considered, especially in the general class of production workflows. However, the predictability and repetitiveness of production workflows cannot be counted upon, in the dynamic business environments of today, where processes are being continually changed in response to new methods and practices, and changes in laws and policies. Furthermore, these processes are often confronted by exceptional cases, and need to deviate from prescribed procedures without loosing control. These exceptional cases may or may not be foreseen.

At the same time, there is a complementary school of thought, which often speaks of ad hoc workflows, or workflows where the process cannot be completely defined prior to execution [1]. Although we do not advocate ad hocism in workflow processes to the extent of a complete relaxation of coordination constraints, it is obvious that ad hocism is meant to promote flexibility of execution. There is sufficient evidence that process models that are too prescriptive introduce a rigidity that compromises the individualism and competitive edge of the underlying business procedures.

Processes which undergo frequent change cannot be completely pre-defined, or simply have too many instance types, require a specification framework which is in tune with the flexible nature of these processes. Processes which depend on the presence of such flexibility for the satisfactory attainment of process goals can be found in many applications, for example:

- A typical example of flexibility is healthcare, where patient admission procedures are predictable and repetitive, however, in-patient treatments are prescribed uniquely for each case, but nonetheless have to be coordinated and controlled.
- Another application is higher education, where students with diverse learning needs and styles are working towards a common goal (degree). Study paths taken by each student need to remain flexible to a large extent, at the same time providing study guidelines and enforcing course level constraints is necessary to ensure a certain quality of learning.
- Effective customer relationship management (CRM), a critical component in enterprise solutions, also signifies the need to provide a flexible means of composing call center activities according to the available resources and data, by integrating CRM systems with core organizational workflow processes and underlying applications.
- Customizable product manufacturing also requires a flexible means of coordination of production processes, since the requirements of individual customers cannot be pre-determined. Here, made-to-order products are produced against customer requirements which are made available for individual items or batches, and may vary from case to case.

The key issue in flexible workflows is the specification of the partial process, from which a complete workflow specification may be derived. Thus rather than enforcing control through a rigid, or highly prescriptive language that attempts to capture every step and every option within the process, the process is defined in a "flexible" manner, that allows individual instances to determine their own (unique) processes. How to achieve such a means of requirements specification is the main focus of this paper.

In the subsequent sections, we will present a unique, generic and practical approach for the specification of flexible workflows. It is important to clarify that this paper does not present a new language for workflow specification. Instead, we will only make use of simple, generic and well-established constructs, such as sequence, fork, choice, etc. [32]. To establish a common understanding of these constructs and notation, we have given in the Appendix, a short explanation of the semantics. Basically, a Workflow is defined as a directed graph W. There are two types of nodes in W, namely activity nodes (rectangle) and coordinator nodes (ellipse). Fig. 1 gives the graphical representation.

We will use the above notation to demonstrate various examples in this paper. The remaining paper is structured as follows: We first present the

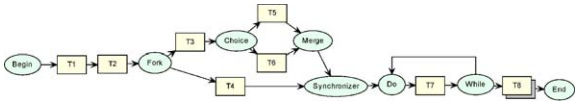


Fig. 1. Workflow modeling constructs.

specification framework, followed by a discussion on achieving the right balance between flexibility and control. This will demonstrate how this specification framework facilitates the attainment of such a balance. We then present in detail our approach to constraint specification, which is a fundamental notion in the framework. Then in the following section, a discussion on constraint validation is presented. We first define various properties of the constraint specification approach, namely transitivity, redundancy and conflicts, and then provide a procedure for achieving a minimal and conflict-free specification. In the next section, the verification of templates built under the given specification framework is discussed. We provide the verification algorithms as well as some discussion on template building support from a system/usability point of view. Finally, we present a short introduction to a flexible workflow engine, Chameleon, which implements the above concepts, before presenting related work and conclusions.

2. Specification framework

It is a long established notion that requirements specification involves the modeling of real-world knowledge and not just functional specification [2]. This representation takes many forms, such as formal specification languages, reactive systems modeling, and conceptual modeling [3]. Several modeling perspectives exist [4] such as notion oriented (e.g. UML) and domain oriented (e.g. ARIS [5]).

Many workflow specification languages have origins in the Petri-net formalism, and process algebra and report different levels of success in validation and verification. Current workflow products and research initiatives have introduced an assortment of workflow specification languages that support several variants of the typical (sequence, choice, fork and iteration) modeling constructs. Introducing flexibility of specification by introducing more constructs within the modeling language has many drawbacks. There is often a semantic overlap in many of these constructs, making it difficult to define and verify. Furthermore, their enactment introduces unnecessary complexity in the workflow engine, limiting its scope and interoperability. However, our approach provides the means for flexible definition without compromising the genericity or simplicity of the modeling language.

We present below a framework for requirements specification for flexible workflows. This framework is based on the concept of *pockets of flexibility* [33].

2.1. Workflow model specification

The specification of the partial, which we shall call the *flexible* workflow, consists of:

- identifiable (pre-defined) workflow activities and control dependencies that form the core *process*;
- *pockets of flexibility* within the process, represented as a special workflow activity called the build activity, and consisting of
  - set of workflow fragments, where a workflow *fragments* may consist of a single activity, or a sub-process,
  - set of *constraints* for concretizing the pocket with a valid composition of workflow fragments.

The assumption is that the control flow between the fragments cannot be completely defined in the core process. The concept of pockets of flexibility aims at compensating for this inability to completely specify the process. The concept as such, is not limited to the workflow modeling language being used. As shown in Fig. 2, the pocket can simply be seen as a special BUILD activity (dotted box) within the workflow model, together with the set of workflow fragments and constraints from which the build activity can form a valid composition.

In Fig. 2, we give an example of a flexible workflow representing a call center response to a user request in a typical CRM environment. This

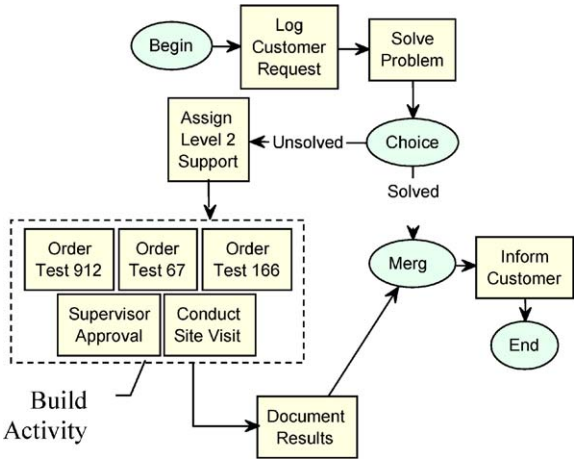


Fig. 2. The flexible workflow.

model is based on the simple language with standard constructs as introduced previously. This language is supported by a process modeling and verification tool FlowMake (www.dstc.edu.au/praxis).

The example above represents a hypothetical and simplified CRM process. Wherein, upon receipt of a customer call, a call center agent will create a request with relevant details in the system, and allocate an engineer based on skills (shown as the activity "Log Customer Request"). The request then appears on the work list of the allocated engineer, who will now have to make decisions on how to resolve the customer request based on his/her expertise (shown as the activity "Solve Problem"). As in typical environments, the problem is addressed and solved in accordance with pre-determined procedures. Temporal/control constraints and quality assurance parameters are associated with these procedures. However, in some circumstances, a greater level of support may be required (so-called level 2 support). It is difficult, if not impossible to determine the exact response warranted for every possible customer request. Thus, some flexibility has to be afforded to the engineers-in-charge. This flexibility is provided within the pocket, but under given control.

In the given example, the level 2 engineers have six different activities (fragments) with which they can compose the pocket, and the following constraints are imposed on building:

- any of the tests may be done, but one at a time;
- the supervisor approval is required to conduct site visit.

One can expect in real-life CRM processes, there will be many more fragments and complex rules, there may be an iterative building of the pocket for difficult cases, and there could be an outcome where the customer request cannot be satisfied. Such complexity is omitted for simplicity. The underlying objective however is to effectively meet the customer request within the given constraints, while making use of individual expertise and experience.

As a side note, it is interesting to point out that the procedures deployed to meet certain requests by different engineers will have different levels of effectiveness. With sufficient volume of operations, the workflow logs can be mined to determine effective practices, and identify successful responses.

2.2. Workflow instance specification

The instance specification initially consists of a copy of the core process. As a particular instance proceeds with execution, the build activities provide the means of customizing the core process for that particular instance. The instance specification prior to building, we call an *open instance*. The instance specification after building we call an *instance template*. Thus, the instance template is a particular composition of the fragments within the flexible workflow. The instance templates in turn have a schema–instance relationship with the underlying execution. In traditional terms, the instance template acts as the process model for a particular instance. Execution takes place with full enforcement of all coordination and temporal constraints, as in a typical production workflow. However, template building is progressive. The core process may contain several pockets of flexibility which will be concretized through the associated build activities as they are encountered.

As such, the template remains open until the last built activity has completed.

In Fig. 3, we give two example instance templates for the flexible workflow given in Fig. 2. These present the two different ways in which customer requests may be handled, however, both templates conform to the build constraints given in the previous section.

Execution will commence with the initial activity of the core process, which establishes the creation of the instance and necessary data. After successful completion of the initial and any subsequent activities, a pocket of flexibility is encountered, and the associated build activity is activated. The execution of the build activity replaces the open instance with the instance template, which in turn serves as a process model for that instance.

The objective is to achieve a workflow specification which effectively captures a business scenario. The process (or build) constraints are the fundamental feature of this approach. The manifestation of flexibility through process constraints is the key to providing configurable process models. We believe that the strength of this approach lies in the separation of the specification of the process constraints which are known, and need to be enforced, and the instance specific requirements, which are difficult to pre-determine and their knowledge lies with actual (end) users. This separation introduces the concept of the open instance and transfers greater control into the hands of actual users, consequently promoting flexibility in work practice.

3. Balancing flexibility and control

In Section 2.1, we have defined the flexible workflow as having the following components: a core process (with normal activities as well as special build activities representing pockets of flexibility), fragments, and build constraints. On a continuum of workflow specification, there is the completely defined model on one end, and the model with no pre-definition on the other. Thus, the former only has strong constraints, e.g. *B* must follow *A*, and the latter no constraints at all. Finding the exact level of specificity along this continuum will mostly be process dependent. Below we present a discussion on how the specification of each component of the flexible workflow can be tuned to strike the right balance between flexibility and control.

3.1. Core process

Within a core process, one or more pockets are built from fragments and constraints. Once built, a pocket is basically a sub-process within the parent core process. The only difference is that the pocket

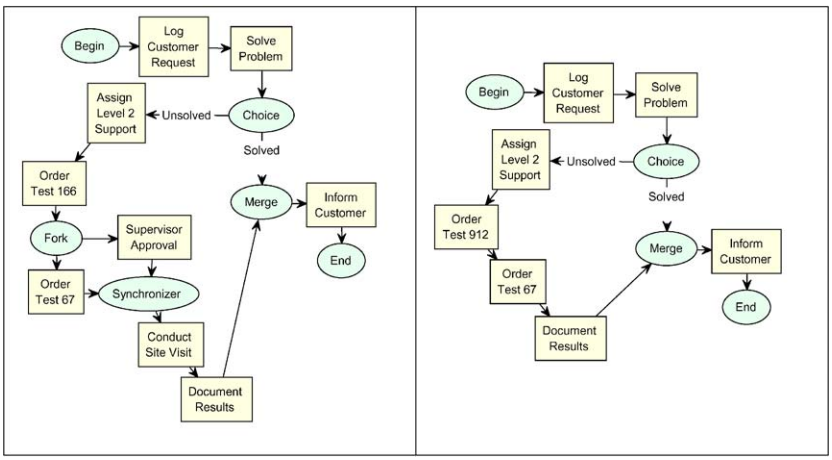


Fig. 3. Instance templates.

may have a more restricted version (explained later in Section 4) of the prescriptive language than the one used to specify the core process. However, building a pocket from given fragments and constraints, or building a process from given fragments and constraints, are conceptually similar. In other words, the entire process model may be defined as a pocket.

The previous examples and discussion assumed that the only components to build a pocket (or process) are fragments and constraints. A generalization of this concept is to allow a core template to be defined as well. This template represents the pre-defined part of the pocket (process). The given fragments and constraints are used to build upon the template, rather than an empty space.

The subsequent discussion will consider this generalized definition of a core process. That is, a core process is a workflow model which may represent the process itself or a pocket within a process, and this model may be of any complexity, including an empty graph. It is henceforth called a core template.

3.2. Fragments

There is no reason why building should be allowed from existing fragments only. There are several examples of processes, where entirely new (unprecedented and hence not pre-defined) activities have to be performed for a given instance, and perhaps subsequently used for other instances. Building from a pre-defined set of fragments, however, will generally be more realistic since it provides some degree of control on what is included in the instance.

3.3. Build constraints

In all of the above cases, the result is basically an instance template, which is dynamically built. The specification of fragments, which are merely workflow activities or sub-processes, as well as the specification of the core process, does not present any new challenges, over and above process modeling. Specification of the constraints imposed on the composition of fragments is novel and specific to the proposed framework. Building may be constrained by several factors, including the data relating to that instance, the stage of execution of the instance, temporal constraints, fragments available for building, and the business rules of the particular application for which the template is being defined. For example, in education, it will be constrained by the progress of the student, such that at any given stage of the study process, the student can build the template from a specified set of study activities under given rules. A student, for example, cannot take up a study activity for which pre-requisites have not been met. Similarly, a student may not be permitted to take more than a given number of study activities at one time.

These constraints are distinct from strong constraints. Whereas strong constraints exhibit the same semantic behavior in all instances, these weak or flexible constraints map to several variants.

Although a build environment may be provided without build constraints, that is, the composition of fragments is not controlled through constraints, the challenge lies more in providing a small set of constraints which will allow the building of a large number of instance templates in a controlled environment. What that set of constraints consists of, is a challenging issue. Another critical question is, how can a template be validated for conformity to these constraints. For example, why is the instance template in Fig. 3 a valid composition. The subsequent sections will address these questions.

From the above discussion, we identify the following options, which define the continuum of process specification. The 'X' in Table 1 indicates the existence of the component in process specification.

Options 2–5 represent various options for flexible definition. In the previous sections, Figs. 2 and 3 provided an illustration of option 4, where the pocket contained fragments and constraints, but no core process (within the pocket). Which option provides the right balance will largely be dictated by the nature of the application. In the following sections, we will continue the discussion based on options 2–5.

Table 1
Options for process specification

	CP	F	BC	
1	X			Complete definition
2	X	X	X	Flexible definition: with given core template
3	X	X		Flexible definition: with given core template, but no constraints on fragments composition
4		X	X	Flexible definition: without core template
5		X		Flexible definition: without core template, or constraints on fragments composition
6	X		X	N/A
7			X	N/A
8				No definition

CP: core process; F: fragments; BC: build constraints.

4. Constraint specification

We identify two main classes of constraints, namely structural class and containment class. The constraints belonging to the structural class impose restriction on how fragments can be composed in the templates. We will discuss three constraint types under the structural class, namely serial, order and fork. The constraints belonging to the containment class identify conditions under which fragments can and cannot be contained in the resulting templates. We will discuss two constraint types within the containment class, namely inclusion and exclusion.

Below we give the meta-definitions for these constraint types. The actual definitions will be made in accordance with given domain requirements. It is important to note that, although these definitions allow for a large number of arbitrary structures to be constructed in the given language, it is not the intention, nor it may be possible, to give a set of constraint types that can satisfy any workflow structure (against any business logic). The underlying motivation is to provide a greater *degree* of flexibility by dynamic building, while still maintaining a desirable level of control through constraint specification. A discussion on possible extensions of the basic set of five constraint types will be given later in Section 4.6. We first introduce basic terminology.

Let F be a set of fragments, and $f \in F$ be a fragment, where a fragment is a sub-process (or a single activity).

Further let $|F| = n$, be the number of fragments in F.

$F_m \subseteq F$ where $m = 1, 2, \ldots$.

C is a set of constraints.

ConstraintType: $C \rightarrow$ {Serial, Order, Fork, Inclusion, Exclusion}, where $\forall c \in C$, ConstraintType (c) represents the type of c.

W is a workflow graph as defined previously. W represents the core process.

P is a pocket (process) given as $P = < W, F, C >$ where W and/or C may be empty. (Definition in accordance with options 2–5 given in Section 3.)

T is a workflow, as well as the target template which is intended to be dynamically built given P.

For the following consideration, the coordinator nodes in T are restricted to begin, end, fork, and synchronizer. While choice and merge coordinators may be present within workflow fragments, we propose that these coordinators not be used to build the instance template. Since an instance template represents a particular occurrence of the workflow process, the decision of what to choose should be made as part of the build activity. For example, the constraint "do any one of A, B or C" will be built as either A or B or C, and not all within a choice-merge construct. The elimination of the choice-merge construct from the template, further has the advantage of removing the chance of structural errors such as deadlocks or lack of synchronization [6].

As a final point, note that there will be a mapping from $P \rightarrow T$, where one pocket may map to several templates. In order for T to be a valid template, all constraints $c1, c2, \ldots cn \in C$ as given in $P = < W, F, C >$ must hold in T, that is $c1 \wedge c2 \wedge \cdots \wedge cn$ must be true.

4.1. Serial constraint

A serial constraint is defined when fragments are to be arranged serially. However, the choice of order remains flexible and is determined by the user during the build. Fig. 4 gives example valid compositions.

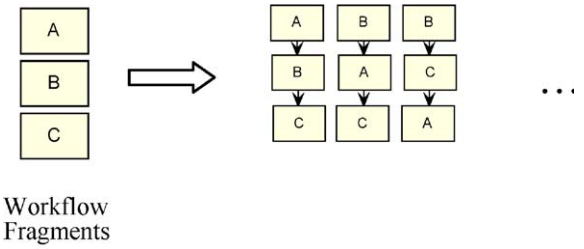


Fig. 4. Serial constructs.

It is interesting to note that this flexibility of composition contradicts conventional sequential constraints, where "A is followed by B" indicates a control flow dependency between A and B. This dependency cannot be established in the above scenario. However, such a flexible composition is required to fulfill constraints such as "do all of A, B and C, but one at a time". A further expansion of compositions can be illustrated by a constraint such as "do any k from a given set of fragments, but one at a time". The serial constraint is basically defined to capture this flexibility. Note that, the serial constraint does not imply consecutive placement, that is, the fragments may have other workflow activities placed between them. Also, the serial constraint does not put any restriction on the number of fragments that must be included in the resulting template.

Scenario: A number of tests have been prescribed for a given patient, e.g. blood test, X-ray and ECG. These tests can be done in any order but only one at a time.

Constraint notation: $S(F_m)$ where F_m is a non-empty subset of F. Thus $S(F_m)$ is represented as $S(\{f1, f2, \ldots, fn\})$; $fi \in F_m$ for $i = 1, \ldots, n$.
Constraint definition: Let $S(F_m)$ be given, and F_k be a subset of F_m, such that all fragments in F_k are present in T. Then $\forall\, fi, fj \in F_k$, there exists a unique path between them in T. The choice of fragments in F_k is user-driven.

4.2. Order constraint

The order constraint is a special case of the serial constraint, where fragments must be executed in serial as well as in a specified order. However, like the serial constraint, the order constraint does not imply consecutive placement, that is, the fragments may have other workflow activities placed between them.

Scenario: Applications for admission in a university program are processed by the admission section as well as the faculty. The admission section receives the applications, screens them for missing documents and in most cases also determines eligibility. In certain cases, the recommendation of the faculty is required to determine academic eligibility. This recommendation may be requested at any stage, provided the application has been screened for missing documents by the admission section before being received by the faculty.

Constraint notation: $O(F_m)$ where F_m is a non-empty subset of F. Thus $O(F_m)$ is represented as $O(\{f1, f2, \ldots, fn\})$; $fi \in F_m$ for $i = 1, \ldots, n$. $O(F_m)$ provides an order on the elements of F_m.
Constraint definition: Let $O(F_m)$ be given, and F_k be a subset of F_m, such that all fragments in F_k are present in T. Then $\forall\, fi, fi + j \in F_k$ where $i = 1, \ldots, n-1$ and $j = 1, \ldots, n-i$, there exists a path from fi to $fi + j$ in T. The choice of fragments in F_k is user-driven.

4.3. Fork constraint

The fork constraint basically allows selected fragments to be executed in a fork. Flexibility of definition is especially required when the fork can be built from any number of fragments. A typical constraint for the above can be "do no more than three of A, B, C or D". Fig. 5 shows the example valid compositions.

Note that execution in fork does not necessarily imply parallel execution, since activities present on

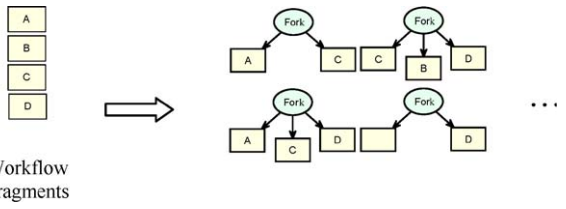


Fig. 5. Fork constructs.

the multiple outgoing branches of a fork may be activated in any order. Hence, this constraint rather implies not sequence. Furthermore, the fragments may appear in fork structures together with other workflow activities.

Scenario: A number of study activities are designed within a course. A student is expected to undertake at least three of these for a given period of time.

Constraint notation: $F(F_m)$ where F_m is a non-empty subset of F. Thus $F(F_m)$ is represented as $F(\{f1, f2, \ldots, fn\})$; $fi \in F_m$ for $i = 1, \ldots, n$.
Constraint definition: Let $F(F_m)$ be given, and F_k be a subset of F_m, such that all fragments in F_k are present in T. Then $\forall fi, fj \in F_k$, there does not exist a path between them in T. The choice of fragments in F_k is user-driven.

4.4. Inclusion constraint

The inclusion constraint identifies a dependency between two sets of fragments. Essentially, the presence (or absence) of fragments from one set imposes a restriction on the fragments of the second set. Inclusion constraint can be supplemented with a serial/order or fork constraint, that imposes an additional restriction on how the included fragments must be composed.

Scenario: In travel booking, a customer making a booking for both flight and accommodation will be provided by free transport from/to the airport.

Constraint notation: $I(F_p, F_m)$ where F_p and F_m are two non-empty subsets of F. Thus $I(F_p, F_m)$ is represented as $I(\{f1, \ldots, fq\}, \{f1, \ldots, f_n\})$; $fi \in F_p$ for $i = 1, \ldots, q$; $fj \in F_m$ for $j = 1, \ldots, n$.
Constraint definition: Two cases may occur:
If all fragments from F_p are present in T then all fragments from F_m must appear in T.
If not all the fragments from F_p are present in T then no rule is enforced on F_m.

4.5. Exclusion constraint

The exclusion constraint, similar to the inclusion constraint, also identifies a dependency between two sets of fragments. However, this prohibits certain fragments from being included in the resulting template.

Scenario: Company travel reimbursements can be made in two ways: payment by cheque or payment by direct deposit. In a particular instance, inclusion of one, should exclude the other.

Constraint notation: $E(F_p, F_m)$ where F_p and F_m are two non-empty subsets of F. Thus $E(F_p, F_m)$ is represented as $E(\{f1, \ldots, fq\}, \{f1, \ldots, fn\})$; $fi \in F_p$ for $i = 1, \ldots, q$; $fj \in F_m$ for $j = 1, \ldots, n$.
Constraint definition: Again, two cases may occur:
If all fragments from F_p are present in T then all fragments from F_m must not appear in T.
If not all the fragments from F_p are present in T then no rule is enforced on F_m.

4.6. Extending the constraint set

The five constraints introduced in this paper provide a basic set, but still allow for a substantial number of templates to be built. However, as explained before, there can always be particular requirements of business processes which these constraint types may not directly satisfy. One can envisage extensions to the basic set of constraint types, which capture additional semantics, for example Min/Max constraints that impose restrictions on how many fragments can be included; or constraints on multiple executions such as "do *A k* number of times in parallel" where k is instance dependent. Introduction of new constraint types will impact on the verification algorithms (Section 6), as well as the existing rules for conflicts (Section 5). For example, fragments that can be executed multiple times should not be part of the argument for an order constraint. Extending the constraint types is similar to extending the constructs in a language. Extensions will allow further diversity in the building of templates, but can also be expected to have an impact on the verification.

5. Constraint validation

An important aspect of the above framework is the validation of the set of constraints. Since these

constraints are the basis for controlling subsequent dynamic building, it is vital that the constraints themselves do not impose a restriction which cannot be met. In other words, can we say that possible structures allowable under a given constraint set are acceptable? Below we give a few examples of such violations:

- Suppose that a fork constraint is given as $F(\{A, B, C\})$, however, there is a data dependency between B and C, i.e. C requires data generated by B. Constructing a fork between B and C would result in a violation of this dependency.
- Suppose an order constraint is given as $O(\{A, B, C, D\})$. However, C has a temporal constraint specified as an absolute deadline say, 1 July 2002. At the same time, the initiation of A is constrained by a given date as well, say 1 August 2002. Thus, the specified order of execution would violate these temporal constraints.
- Suppose an inclusion constraint is given by $I(\{A\}, \{B\})$, however A and B are incompatible activities both of which should not be done together for a given instance. For example, they could represent an "Offer discount" and an "Offer on VIP member price" activity, respectively, within a sales workflow.

Thus, constraints may be invalid for a number of reasons including missing or lost data, temporal violations, and business rule contradictions. However, specifying incorrect constraints as above is equivalent to constructing a process model that does not meet business constraints and/or process goals. Obviously, the greater the knowledge of the designer in terms of the business domain and properties of underlying activities and applications, the less chance of incorrect constraint specification.

Incorrect or missing constraints in a prescriptive process model would also result in the process goals being compromised. That is, a process model may be structurally correct with respect to the workflow modeling language, but may be incorrect with respect to the business logic. Similarly, in our framework, templates may be built which are in conformance with given constraints, but do not represent a sensible process due to incorrect or missing constraints. This is not a new problem and substantial work exists, especially in the context of software validation [7].

Constraint validation in the context of our approach primarily deals with the identification of conflicts within the constraint set. Since individual constraints could hold, but the conjunction of constraints may fail due to conflicts. It is essential to ensure that template verification (Section 6) is conducted on a conflict-free and minimal constraint set. Furthermore, constraint specification may have transitive and/or redundant elements. In the following sections, we will discuss further the properties of these constraints and demonstrate how the above may be achieved.

5.1. Transitivity in constraints

It is important to note that three of the above constraint types, specifically inclusion, exclusion and order, have a transitivity property. For example, $O(\{A, B\})$ and $O(\{B, C\})$ implies that there is an order constraint on $\{A, B, C\}$, that is $O(\{A, B, C\})$ can replace the above two. Similarly, $I(\{A\}, \{B, C\})$ and $I(\{B\}, \{D, E\})$ implies that B, C, D, and E must all be present when A is present. Although, unlike the order constraint, $I(\{A\}, \{B, C, D, E\})$ does not imply $I(\{B\}, \{D, E\})$, and as such cannot replace the above two inclusion constraints.

However, as the number of constraints increase, identifying the transitivity will not be so trivial. In addition, these constraints may have redundancy (Section 5.2) in their specification, and order constraints may also contain conflicts (Section 5.3).

Fork and serial do not possess the transitivity property. That is, given $S(\{A, B\})$ and $S(\{B, C\})$ does not imply $S(\{A, C\})$ since the two constraints can be satisfied by the construct given in the Fig. 6(a). Similarly, given $F(\{A, B\})$ and $F(\{B, C\})$ does not imply $F(\{A, C\})$, as illustrated in the Fig. 6(b).

5.2. Redundant constraints

Redundant constraints will cause a non-minimal specification and may introduce inefficiency in the

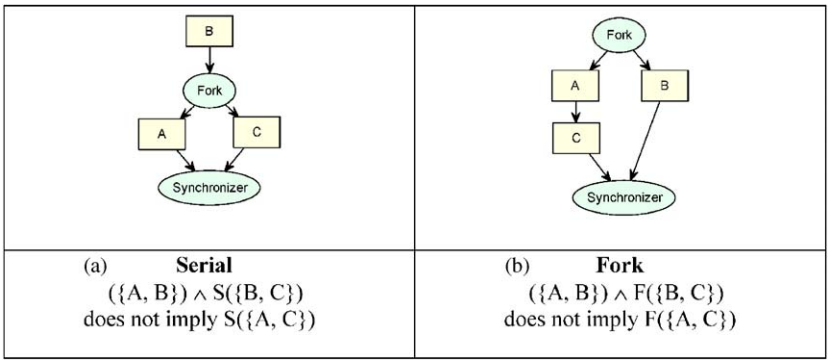


Fig. 6. Non-transitivity in constraints.

Table 2
Redundancy between constraint types

	Order	Serial	Fork	Inclusion	Exclusion
Order	✓	✓			
Serial	✓	✓			
Fork			✓		
Inclusion				✓	
Exclusion					✓

template verification procedure. For example, two constraints $O(\{A, B, C\})$ and $O(\{A, B\})$ are given. However, $O(\{A, B, C\})$ subsumes $O(\{A, B\})$ making it redundant. Table 2 identifies where potential redundancy may exist. Redundancy mostly exists within constraint types, except in the case of order and serial, redundancy can also exist across constraint types.

Redundancy between order and serial constraints: We say that if $O(F_m)$ and $S(F_n)$ are given and $|F_m \cap F_n| > 1$, then there is potential redundancy in $S(F_n)$. A trivial case is when $F_n \subseteq F_m$, in which case the entire constraint, $S(F_n)$, is redundant. Thus, the stronger constraint of order will subsume the serial constraint.

Redundancy between order (serial, fork) constraints: If $O(F_m)$ and $O(F_n)$ are given and $|F_n \cap F_m| > 1$, then there is potential redundancy within these constraints. A trivial case is when $F_n \subseteq F_m$, in which case $O(F_n)$ is redundant.

Similarly, there can be redundancy between serial and fork constraints.

Redundancy between inclusion (exclusion) constraints: If $I(F_p, F_m)$ and $I(F_q, F_n)$ are given and $F_p = F_q$, then there is redundancy within these constraints. Thus, $I(F_p, F_m)$ can be changed to $I(F_p, F_m \cup F_n)$, making $I(F_q, F_n)$ redundant.

Another case of redundancy also exists when $F_p \cap F_m \neq \varnothing$. Thus, $I(F_p, F_m)$ can be changed to $I(F_p, F_m - (F_p \cap F_m))$, without any loss.

Similarly, there can be redundancy between exclusion constraints.

5.3. Conflicting constraints

Conflicts in constraints can potentially cause a scenario where no template built can be verified for conformance to the given constraints. For example $O(\{A, B\})$ and $O(\{B, A\})$ are conflicting constraints, and any given graph containing A and B will not be verified.

Similarly consider the constraints $S(\{A, B\})$ and $F(\{A, B\})$. These are conflicting since the serial constraint requires that a path be present between A and B, and the fork constraint requires that a path not be present. However, there may be a graph constructed that does not contain A or B. Such a graph can be verified in spite of the conflict. Although, if an additional constraint $I(\{A\}, \{B\})$ is included, then together the three constraints will not allow any graph containing A to be built. We therefore define a conflict between any two constraint, whether of the same type or of different types, as the following:

> A conflict exists between two constraints when they prohibit one or more fragments from being included in any valid template.

Hence, it is critical to identify and eliminate conflicts in the specified constraints. Table 3 identifies the conflicting constraint types. Thus, conflicts can arise within a constraint type (order, inclusion, exclusion) as well as between constraint types (fork/serial, fork/order, inclusion/exclusion).

The following identify potential conflicts within and between constraint types. However, the resolution of the constraints is semantically driven. That is, if conflicting constraints $O(\{A, B\})$ and $O(\{B, A\})$ are given, then only a user with domain knowledge can resolve which one is to be taken.

Conflict between order constraints: A conflict will exist between two order constraints if any pair of fragments is present in both, but in conflicting order.

Given $O(F_m)$ and $O(F_n)$, there will be a conflict when

$\exists fi, fj \in F_m$ such that fi precedes fj in F_m and
$\exists f_s, f_t \in F_n$ such that f_s precedes f_t in F_n and
$fi = f_t$ and $fj = f_s$

However, due to the presence of transitivity and redundancy, conflicts between order constraints may not be so straightforward to detect. The next section will discuss this issue in greater detail.

Conflict between fork and serial constraints: A conflict will exist between a fork and serial constraint if they have any two or more fragments in common.

Given $S(F_m)$ and $F(F_n)$, there will be a conflict when $|F_m \cap F_n| > 1$

Conflict between fork and order constraints: Similarly, a conflict will exist between a fork and order constraint if they have any two or more fragments in common.

Given $O(F_m)$ and $F(F_n)$, there will be a conflict when $|F_m \cap F_n| > 1$.

Table 3
Conflicting constraint types

	Order	Serial	Fork	Inclusion	Exclusion
Order	✓		✓		
Serial			✓		
Fork	✓	✓			
Inclusion					✓
Exclusion				✓	

Conflict between inclusion and exclusion constraints: A constraint set should not have inclusion and exclusion constraints on the same fragment, hence conflicts may exist between an inclusion and exclusion constraint.

Given $I(F_p, F_m)$, $E(F_q, F_n)$, there will be an obvious conflict when $F_p = F_q$ and $F_m \cap F_n \neq \varnothing$.

However, there can be less obvious cases of conflicts. For example, given $I(F_p, F_m)$, $E(F_q, F_n)$, and $F_q \cup F_n \subseteq F_m$ or $F_p \cup F_m \subseteq F_n$. As a simple illustration consider $I(\{A\}, \{B, C\})$ and $E(\{B\}, \{C\})$. Any template containing A will not be verified against the above constraints, hence representing a conflict.

We will present in the next section, a means of combined specification for inclusion and exclusion constraints, which will identify conflicts in a generic manner, and provide a minimal and conflict free specification.

5.4. Minimal specification

Due to the inherent transitivity and redundancy which may exist in constraints, it is not possible to identify conflicts from the original set of constraints. For example, given a set of order constraints:

$$O_1(\{A, B, C\}), \quad O_2(\{A, C, D, G\})$$

and

$$O_3(\{A, G, B\}).$$

There is potential redundancy since $|\{A, B, C\} \cap \{A, C, D, G\}| > 1$. There is also obvious transitivity in constraints O_1 and O_2 since A must precede B, and B must precede C according to O_1 and, in turn C must precede D (and D must precede G) according to O_2, indicating an ordering constraint on $\{A, B, C, D, G\}$. Although not obvious from the original set of constraints, it is obvious that $\{A, B, C, D, G\}$ and $\{A, G, B\}$ are in fact in conflict due to B and G. A resolution of this conflict rests on the user since the ordering represents a semantic dependency between B and G. However, the identification of this conflict definitely must be system supported.

What the above example illustrates is that it may not be always possible to provide a rigorous

analysis of the constraints from the original set of constraints as given by the user, due to the flexible nature of constraint specification. This set may contain transitivity, redundancy, as well conflicts. However, rather than restricting constraint specification, which defeats the whole purpose of flexibility, we will attempt to provide a means of validating this highly flexible approach to constraint specification.

We will present below a means of finding minimal representation for each of the above constraint types. This will provide an equivalent but minimal (redundancy and conflict free) specification required for a set of constraints of a given type. It will then be used in the constraint validation procedure given in the next section.

5.4.1. Order

Let $\mathscr{C}(\mathscr{O})$ be a set of order constraints defined on $F_1, F_2, \ldots, F_n$ where $Fi \subseteq F$ for $i = 1, 2, \ldots, n$.

Further let $F^{\text{Order}} = \{F_1, F_2, \ldots, Fn\}$

For each $\mathscr{C}(\mathscr{O})$ we associate $\mathscr{O}\check{\mathscr{G}}$, a directed graph.[1]

We say that $\mathscr{C}(\mathscr{O})$ is conflict free if the corresponding $\mathscr{O}\check{\mathscr{G}}$ is a directed acyclic graph (DAG). A cycle in the graph $\mathscr{O}\check{\mathscr{G}}$ represents a conflict within the ordering constraints. The elimination of the conflict, which as mentioned before, is semantically driven, will result in a different set of order constraints and consequently a different $\mathscr{O}\check{\mathscr{G}}$.

Let $\mathscr{O}\mathscr{G}$ be the DAG representing the conflict free set of order constraints $\mathscr{C}(\mathscr{O})$.

We say that $\mathscr{C}(\mathscr{O})_1$ and $\mathscr{C}(\mathscr{O})_2$ are equivalent if $\mathscr{O}\mathscr{G}_1 = \mathscr{O}\mathscr{G}_2$, and call them covers. Clearly, the same $\mathscr{O}\mathscr{G}$ may be generated by many different $\mathscr{C}(\mathscr{O})_i$'s, but all are equivalent covers.

For a given $\mathscr{C}(\mathscr{O})$, we define a minimum cover $\mathscr{C}(\mathscr{O})^{min}$, as the smallest set $\mathscr{C}(\mathscr{O})_i \approx \mathscr{C}(\mathscr{O})$. There may be several minimum covers, all with the same number of elements (order constraints), but with different arguments. The minimal cover $\mathscr{C}(\mathscr{O})^{min}$ will be used in the constraint validation procedure given in the next section.

Finding a minimum cover for a given $\mathscr{C}(\mathscr{O})$ constitutes the following steps:

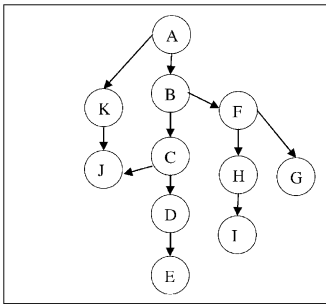
1. For efficiency, we can first eliminate all trivially redundant order constraints, that is for any F_m, $F_n \in F^{\text{Order}}$, if $F_n \subseteq F_m$, then $\mathscr{C}(\mathscr{O}). = \mathscr{C}(\mathscr{O}). - \{O(F_n)\}$.
2. We then map $\mathscr{C}(\mathscr{O})$ onto $\mathscr{O}\check{\mathscr{G}}$. The set of nodes for $\mathscr{O}\check{\mathscr{G}}$ is given by the union of all subsets contained in F^{Order}, that is $F_1 \cup F_2 \cup \cdots \cup F_n$. Thus, each node represents a fragment. An edge between any two nodes *fi* and *fj* is defined if *fi* and *fj* are elements of the same subset F_n, that is an order constraint $\mathscr{O}(F_n)$ exists in $\mathscr{C}(\mathscr{O})$.
3. Remove all *superfluous* edges, where a superfluous edge between two nodes is identified when there exists another path in the same direction, between the same two nodes. This eliminates any redundant specification within the order constraints.
4. Identify cycles (conflicts) in $\mathscr{O}\check{\mathscr{G}}$ if any, and resolve conflicts through user input.
5. Repeat steps 1–4 until $\mathscr{C}(\mathscr{O})$ maps to a DAG $\mathscr{O}\mathscr{G}$.
6. Traverse $\mathscr{O}\mathscr{G}$ to determine $\mathscr{C}(\mathscr{O})^{min}$. The determination of $\mathscr{C}(\mathscr{O})^{min}$ from $\mathscr{O}\mathscr{G}$, is simply a matter of finding all maximal non-branching paths of $\mathscr{O}\mathscr{G}$, such that no two paths have an edge in common.

The above is now illustrated with the following example. Order constraints are given as

$$\begin{aligned}\mathscr{C}(\mathscr{O}) = \{\, & O(\{A,\ B,\ C\}),\\ & O(\{A,\ K,\ J\}),\\ & O(\{A,\ C,\ J\}),\\ & O(A,\ B,\ F,\ G),\\ & O(\{F,\ H,\ I\}),\\ & O(\{B,\ F,\ H\})\}.\end{aligned}$$

We can construct the graph in Fig. 7 from these constraints. The superfluous edges are already removed. It is clear that the graph constructed is a DAG and hence $\mathscr{C}(\mathscr{O})$ is conflict free. From the DAG we can extract a $\mathscr{C}(\mathscr{O})^{min}$, which represents a

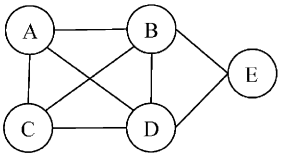
[1] Note that for a given set of order constraints, there can be more than one $\mathscr{O}\check{\mathscr{G}}$ since not all constraints will have fragment subsets with common elements. However, the principle remains the same. This applies to the characterization of the other constraints as well.



$\mathscr{C}(\mathscr{O})^{min}$ = {O({A, B, C, D, E}),
O({A, K, J}),
O({C, J}),
O({B, F, G}),
O({F, H, I})}

$\mathcal{OG}$ representing $\mathscr{C}(\mathscr{O})$	$\mathscr{C}(\mathscr{O})^{min}$ a minimal cover for $\mathscr{C}(\mathscr{O})$

Fig. 7. Minimal cover for order constraint.



$\mathscr{C}(\mathscr{S})^{min}$ = {S({A, B, C, D}),
S({B, D, E}) }

$\mathcal{SG}$ representing $\mathscr{C}(\mathscr{S})$	$\mathscr{C}(\mathscr{S})^{min}$, a minimal cover for $\mathscr{C}(\mathscr{S})$

Fig. 8. Minimal cover for serial constraint.

minimal cover for $\mathscr{C}(\mathscr{O})$. In $\mathscr{C}(\mathscr{O})^{min}$ all transitivity has been captured and redundancies eliminated, hence leading to a more concise specification for the same set of constraints.

5.4.2. Serial

Let $\mathscr{C}(\mathscr{S})$ be a set of serial constraints defined on $F_1, F_2, , \ldots, F_n$ where $Fi \subseteq F$ for $i = 1, 2, \ldots, n$.

Further let $F^{\text{Serial}} = \{F_1, F_2, \ldots, F_n\}$

For each $\mathscr{C}(\mathscr{S})$ we associate $\mathcal{SG}$, a fully connected graph.

We say that $\mathscr{C}(\mathscr{S})_1$ and $\mathscr{C}(\mathscr{S})_2$ are equivalent if $\mathcal{SG}_1 = \mathcal{SG}_2$, and call them covers. Similar to minimal cover for order constraints, for a given $\mathscr{C}(\mathscr{S})$, we define a minimum cover $\mathscr{C}(\mathscr{S})^{min}$, as the smallest set $\mathscr{C}(\mathscr{S})_i \approx \mathscr{C}(\mathscr{S})$. Again, there may be several minimum covers, all with the same number of elements (serial constraints), but with different arguments.

We first illustrate with an example. Suppose the following serial constraints are given:

$\mathscr{C}(\mathscr{S}) = \{ S(\{A, B, D\}).$
$S(\{B, D, C\}),$
$S(\{A, C\}),$
$S(\{D, C\}),$
$S(\{B, D, E\})\}.$

We can construct the graph in Fig. 8 from these constraints, and then extract $\mathscr{C}(\mathscr{S})^{min}$, representing the minimal cover for $\mathscr{C}(\mathscr{S})$. Again $\mathscr{C}(\mathscr{S})^{min}$ eliminates redundancies found within the original specification, leading to a minimal specification.

Finding a minimum cover for a given $\mathscr{C}(\mathscr{S})$ constitutes the following steps:

1. We first eliminate all redundant serial constraints, that is, for any F_m, $F_n \in F^{\text{Serial}}$, if $F_n \subseteq F_m$, then $\mathscr{C}(\mathscr{S}) = \mathscr{C}(\mathscr{S}) - \{S(Fn)\}$.

2. The next step is to map $\mathscr{C}(\mathscr{S})$ onto $\mathscr{SG}$. The set of nodes for $\mathscr{SG}$ is given by the union of all subsets contained in F^{Serial}, that is $F_1 \cup F_2 \cup \cdots \cup F_n$, where each node represents a fragment. An edge between any two nodes *fi* and *fj* is defined if *fi* and *fj* are elements of the same subset F_n, that is a serial constraint $S(F_n)$ exists in $\mathscr{C}(\mathscr{S})$.
3. Finally, we traverse $\mathscr{SG}$ to determine $\mathscr{C}(\mathscr{S})^{min}$. The determination of $\mathscr{C}(\mathscr{S})^{min}$ from $\mathscr{SG}$ can be related to a well-known problem, which is the determination of all (maximal) cliques within a graph. Although the clique problem is known to be NP-complete, due to the nature of these constraints, the number of fragments and consequently the number of nodes in $\mathscr{SG}$ will always be a small number.

5.4.3. Fork

Let $\mathscr{C}$ ($\mathscr{F}$) be a set of fork constraints defined on $F_1, F_2, \ldots, F_n$ where $Fi \subseteq F$ for $i = 1, 2, \ldots, n$.

Further let $F^{\text{Fork}} = \{F_1, F_2, \ldots, Fn\}$.

The characterization of fork constraints can be done similar to serial. Although the semantics of the two constraints are different, since they have the same properties (no transitivity, no conflict, but potential redundancy), the minimal specification can be determined in a similar way.

For each $\mathscr{C}(\mathscr{F})$ we associate $\mathscr{FG}$, a fully connected graph. However, the semantics of an edge in $\mathscr{FG}$ is different from $\mathscr{SG}$. In $\mathscr{FG}$, an edge represents the absence of a path between the corresponding fragments in any valid template.

We say that $\mathscr{C}(\mathscr{F})_1$ and $\mathscr{C}$ $(\mathscr{F})_2$ are equivalent if $\mathscr{FG}_1 = \mathscr{FG}_2$, and call them covers. Similar to minimal cover for order constraints, for a given $\mathscr{C}(\mathscr{F})$, we define a minimum cover $\mathscr{C}(\mathscr{F})^{min}$, as the smallest set $\mathscr{C}(\mathscr{F})_i \approx \mathscr{C}(\mathscr{F})$. Again, there may be several minimum covers, all with the same number of elements (fork constraints), but with different arguments.

The procedure for finding a minimum cover $\mathscr{C}(\mathscr{F})^{min}$ for a given $\mathscr{C}(\mathscr{F})$, is the same as for serial constraints:

1. We again first eliminate all redundant serial constraints, that is, for any F_m, $F_n \in F^{\text{Serial}}$, if $F_n \subseteq F_m$, then $\mathscr{C}(\mathscr{F}) = \mathscr{C}(\mathscr{F}) - \{F(F_n)\}$.
2. Map $\mathscr{C}(\mathscr{F})$ onto $\mathscr{FG}$.
3. Traverse $\mathscr{FG}$ to determine $\mathscr{C}(\mathscr{F})^{min}$.

5.4.4. Inclusion and exclusion

The minimal specification for inclusion and exclusion constraints can be determined in a simple and uniform manner through the use of truth tables. For every inclusion and exclusion constraint, we associate $\mathscr{TT}$, a truth table. The truth table is merely a representation of the logical relation between constraint arguments. For example, for an inclusion constraint $I(\{A\}, \{B\})$, the truth table would be constructed for $\neg A \vee B$.

We say that two containment constraints are in conflict when they have common columns in their truth table, and they have a different collection of rows within the common columns.

We illustrate this with examples. Suppose $I(\{A\}, \{B\})$ and $E(\{A\}, \{B\})$ are given representing an obvious conflict. The truth tables appear as below, and indicate different rows among common columns.

A	B
1	1
0	1
0	0

$\mathscr{TT}$ for I({A}, {B})

Conflicting rows

A	B
1	0
0	1
0	0

$\mathscr{TT}$ for E({A}, {B})

Similarly, suppose $I(\{A\}, \{B, C\})$ and $E(\{B\}, \{C\})$ are given:

Common columns

A	B	C
1	1	1
0	1	1
0	0	0
0	0	1
0	1	0

$\mathscr{TT}$ for I({A}, {B, C})

Conflicting rows

B	C
1	0
0	1
0	0

$\mathscr{TT}$ for E({B}, {C})

As with other constraints, resolution of conflicts will need user input. In order to find a minimal and conflict free specification for a set of inclusion and exclusion constraints, the following simple procedure can be applied:

1. remove trivial redundancies as identified in Section 5.2;
2. for every constraint, generate a truth table;
3. identify and resolve conflicting constraints through corresponding truth tables.

Furthermore, truth tables with common columns but no conflicting rows can be combined, such that a single truth table can be constructed for all constraints with common elements. For example, given $I(\{A\}, \{B\})$, $I(\{B\}, \{C\})$ and $E(\{C\}, \{D\})$, we have

A	B
1	1
0	1
0	0

$\mathcal{TT}$ for I({A}, {B})

B	C
1	1
0	1
0	0

$\mathcal{TT}$ for I({B}, {C})

C	D
1	0
0	1
0	0

$\mathcal{TT}$ for E({C}, {D})

Giving

A	B	C	D
1	1	1	0
0	1	1	0
0	0	1	0
0	0	0	1
0	0	0	0

The resulting truth tables provide a list of allowable combinations of fragments. This, in turn can be used to verify templates, greatly simplifying the verification procedure. The truth table characterization not only provides a means of achieving a conflict-free specification, but it can be extended to any constraint type under the containment class of constraints.

5.5. Constraint validation procedure

The constraint validation procedure has the following basic steps:

1. generation of minimal and conflict-free specification for *each* constraint type;
2. conflict resolution *between* constraint types.

The procedures for the individual steps are given below. Suppose, we are given a set of user specified constraints C.

The algorithms for the generation for minimal covers have been given in the previous section. Thus, for each set of constraints of a given constraint type, we find the minimal cover. Collectively $C \rightarrow C^{\min}$, where $C^{\min}$ consists of the collection of minimal and conflict free specifications for each constraint type.

We have identified four conflicts between constraint types in Section 5.3, namely order/order, fork/order, fork/serial and inclusion/exclusion. Generation of minimal specification (Section 5.4) eliminates conflicts between order constraints (order/order) and between inclusion and exclusion constraints (inclusion/exclusion). The remaining conflicting constraints that is, fork/order, and fork/serial are determined by applying the conflict rules as given in Section 5.3. Conflict resolution however, as mentioned before, will be dependent on the user.

For example, two conflicting constraints $O(\{A, B, C\})$ and $F(\{A, B, D\})$ are given. The conflict is identified since $|\{A, B, C\} \cap \{A, B, D\}|$ is not less than 2. A resolution of this conflict may result in the fork constraint being changed to $F(\{A, D\})$ and $F(\{B, D\})$, order constraint remaining the same. Similarly, the order constraint may be changed to $O(\{A, C\})$ and $O(\{B, C\})$, with the fork constraint remaining the same. Another possible change can be the dropping of the order constraint altogether, and so on.

As a result of choices made during conflict resolution, the constraint set will be changed. The changes made will need to be validated by iterating through the above steps, eventually generating a minimal and conflict-free constraint set.

Thus $C^{\min} \rightarrow C^{\text{valid}}$, where C^{valid} represents the minimal and conflict free set of constraints.

Implementation of the procedures given above in a validation tool will allow a complete analysis of the constraint set prior to deployment. Although such a validation tool can only *identify* the potential conflicts, that is resolution of the conflicts requires the expertise of a domain expert with knowledge of the underlying semantics of the business process. However, it is essential to provide complete automated support for the identification of all conflicts, as proposed by the procedures above.

6. Template verification

The set of five constraints discussed above provide a minimum set of meta-definitions for constraints, which once populated according to given domain requirements, will be used to verify an arbitrary construct built within a pocket in a given language. This takes verification to a new dimension, i.e. template verification under (3). Typically verification in workflow specification provides (1) and (2) only.

1. *Syntactic verification*: The model constructed is correct with respect to the grammar of the language, e.g. a transition cannot have more than one from-node, or the graph must have a single initial and final node, etc.
2. *Structural verification*: Two structural errors for the given language have been identified: deadlocks and lack of synchronization [8]. However, as explained previously, under the restricted language of the template (allowable coordinator nodes are begin, end, fork, synchronizer), the problem of these errors does not arise.
3. *Template verification*: The constraints given above basically provide a means by which requirements for highly flexible processes can be captured. It is obvious that typically a large number of models can be built in conformity with these constraints. Hence, the need to verify that the model built conforms to the given constraints is critical.

We illustrate this with the help of a simple example.

Let $F = \{f1, f2, f3, f4, f5, f6, f7, f8, f9, f10\}$. Let the following constraints specify process requirements. Underlying assumption is that these constraints are minimal and conflict free: $O(\{f1, f3, f5\})$; $F(\{f2, f3, f4, f7\})$; $I(\{f7\}, \{f2, f3\})$. We assume that building is done on an empty pocket (process) as given in option 4 in Section 3.

In Fig. 9, several workflow graphs are given, which represent a valid template against the specified fragments and constraints. It is important to note that these do not represent *all* possible valid graphs that can be constructed against the given constraints. These are merely some examples to demonstrate the explosion of possibilities, and consequently the flexibility of workflow specification afforded by this approach.

We can similarly find a set of templates that violate one or more of the given constraints. Examples of invalid templates are given in Fig. 10.

In all the above cases it can be easily verified when the constraints are being met, or violated. It is also clear that once the number of constraints increase and/or the template built is large and complex, this visual verification will not be sufficient. Thus, algorithms for verifying these constraints are required. We present below the algorithms for verification of each constraint type. These algorithms are based on the set of validated constraints C^{valid}, as determined by the constraint validation procedure given in the previous section.

6.1. Verification of order constraint $O(F_m)$

```
GraphNodes : Set of Nodes in the workflow
graph
OrderFrags: Set of fragments in F_m
Check: Set of Nodes

//Where the sets GraphNodes, OrderFrags, and
  Check have elements of the
//same type, i.e. a fragment will be a node in
  the graph

//InDownPath (X, Y) returns true when a path
  from X to Y can be found in the
//workflow graph

//Count (S) returns the number of elements in
  the set S
```

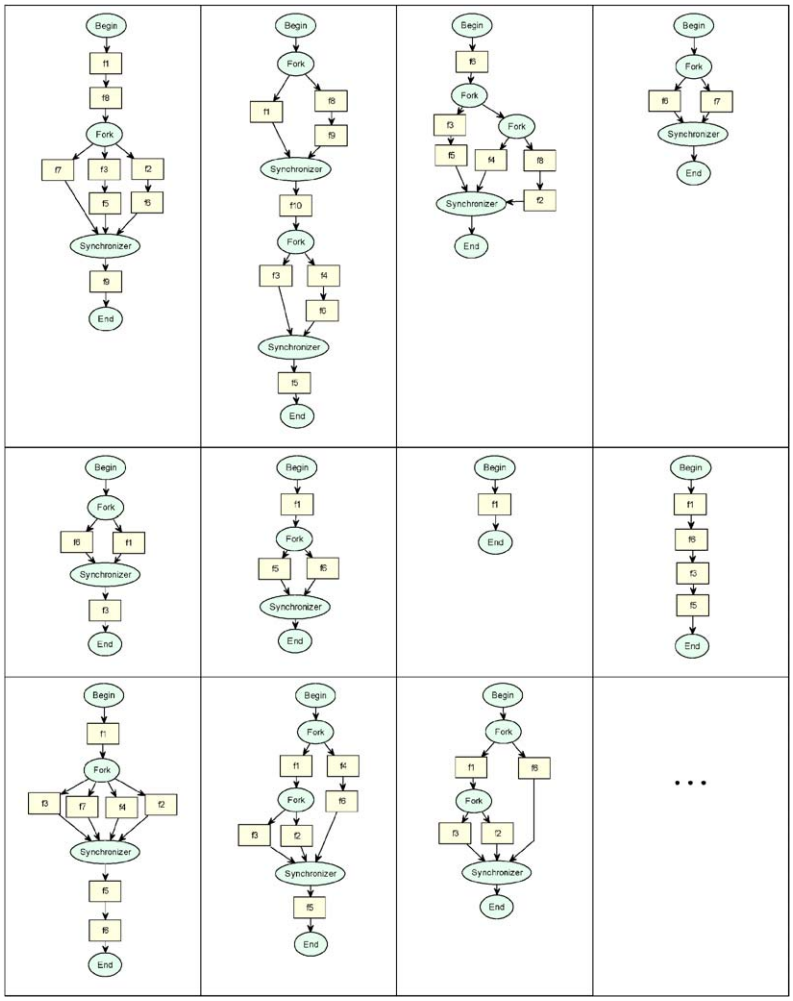


Fig. 9. Valid templates.

```
Begin
    Check : = OrderFrags Intersect Graph-
Nodes
    If Check = {} then Return (''No Con-
    straint Violation'')
    For i = 1 to Count (Check) −1
        If      InDownPath      (Check[i],
        Check[i+1]) <> TRUE
        Then
      Return-Error (''Order Constraint Vio-
      lated'')
        End-If
    End-For
  Return (''No Constraint Violation'')
End
```

6.2. Verification of serial constraint $S(F_m)$

```
GraphNodes : Set of Nodes in the workflow
graph
SerialFrags: Set of fragments in Fm
Check: Set of Nodes

//Where the sets GraphNodes, SerialFrags,
  and Check have elements of the
//same type, i.e. a fragment will be a node in
  the graph

//InPath (X, Y) returns true when a path
  between X and Y can be found in the
//workflow graph. InPath (X, X) returns true.
```

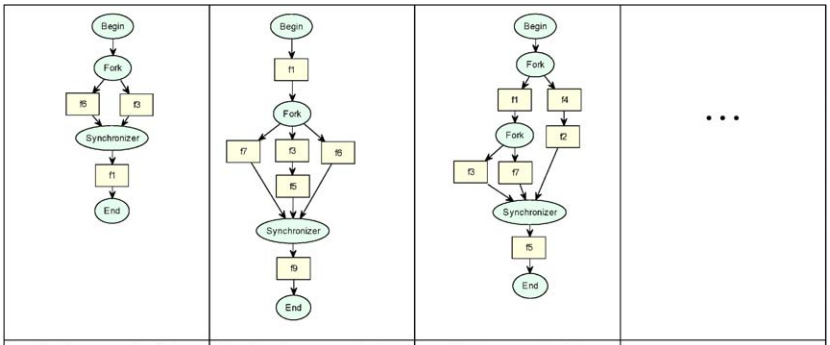


Fig. 10. Invalid templates.

```
//Count (S) returns the number of elements in
  the set S

Begin
    Check := SerialFrags Intersect Graph-
    Nodes
    If Check = {} then Return (''No Con-
    straint Violation'')
    For i = 1 to Count (Check)
      For j = 1 to Count (Check)
        If InPath(Check[i], Check[j]) <>
        TRUE
        Return-Error (''Serial Constraint
        Violated'')
        End-If
      End-For
  End-For
    Return (''No Constraint Violation'')
End
```

6.3. Verification of fork constraint $F(F_m)$

```
GraphNodes : Set of Nodes in the workflow
graph
ForkFrags: Set of fragments in Fm
Check: Set of Nodes

//Where the sets GraphNodes, ForkFrags, and
  Check have elements of the
//same type, i.e. a fragment will be a node in
  the graph

//InPath (X, Y) returns true when a path
  between X and Y can be found in the
//workflow graph. InPath (X, X) returns
  true.

//Count (S) returns the number of elements in
  the set S

Begin
    Check : = ForkFrags Intersect GraphNodes
    If Check = {} then Return (''No Con-
    straint Violation'')
    For i = 1 to Count (Check)
      For j = 1 to Count (Check)
        If InPath(Check[i], Check[j]) =
        TRUE
        Return-Error (''Fork Constraint
        Violated'')
        End-If
      End-For
  End-For
    Return (''No Constraint Violation'')
End
```

6.4. Verification of inclusion constraint $I(F_p, F_m)$

```
GraphNodes : Set of Nodes in the workflow
graph
Present: Set of fragments in Fp
```

```
Include: Set of fragments in Fm

//Where the sets GraphNodes, Present, In-
  clude have elements of the same
//type, i.e. a fragment will be a node in the
  graph

Begin
    If Present Subset GraphNode
    Then
      If Include Subset GraphNodes
      Then
        Return (''No Constraint Viola-
        tion'')
      Else
        Return-Error (''Inclusion Con-
        straint Violated'')
    Else
        Return (''No Constraint Enforced'')
    End-If
End
```

6.5. Verification of exclusion constraint $E(F_p, F_m)$

```
GraphNodes : Set of Nodes in the workflow
graph
Present: Set of fragments in Fp
Exclude: Set of fragments in Fm

//Where the sets GraphNodes, Present, Ex-
  clude have elements of the same
//type, i.e. a fragment will be a node in the
  graph

Begin
    If Present Subset GraphNode
    Then
      If Exclude Intersect GraphNodes = {}
      Then
        Return (''No Constraint Viola-
        tion'')
      Else
        Return-Error (''Exclusion Con-
        straint Violated'')
    Else
      Return (''No Constraint Enforced'')
    End-If
End
```

7. Chameleon: flexible workflow engine

The concept of pockets of flexibility has been implemented in a prototype flexible workflow engine: Chameleon (www.dstc.edu.au/praxis). Chameleon is supported by a process editing and verification tool FlowMake (www.dstc.edu.au/praxis).

7.1. Deployment of the flexible workflow

Below we explain the functions of the flexible workflow management system based on the pocket of flexibility concept. The discussion is presented as a series of steps in the specification and deployment of an example process. Fig. 11 provides an overview diagram of these steps and associated functions of the flexible workflow engine.

Step 1: The definition of the (flexible) workflow model takes place. The process with or without embedded pockets, and where applicable, core templates for pockets, fragments and constraints are defined. That is, any of the options 2–5 as given in Table 1 may be utilized.

The pocket is implemented through a BUILD activity which is followed by a *sub-process object*. The sub-process may or may not contain a core template, fragments and constraints — in this example it consists of a set of fragments, but no core template or build constraints are given (option 5 of Table 1). The sub-process object acts as a place holder for what is going to be built dynamically.

Fig. 12 shows another example of a customer support process model in CRM domain modeled with pocket of flexibility concept. A customer problem is logged by a front end customer interaction center consultant in "Log Customer Problem" activity. The next activity "level 1 Support" is assigned to a help desk staff with level 1 expertise. If he is able to solve the problem, the "Reply to Customer" activity is assigned to a front end customer consultant. In case level 1 customer is not able to resolve the customer problem, the problem is passed to a level 2 support staff. If the level 2 staff member is able to resolve the problem, the process control is transferred to

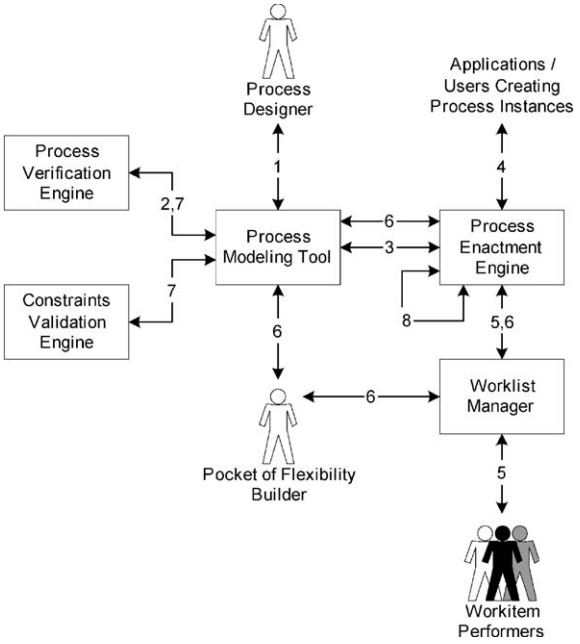


Fig. 11. Deployment of a flexible workflow.

"Reply to Customer" activity. However, it is possible that level 2 staff is also not able to resolve the problem. In this case, the level 2 staff is also our "expert user" who is able to review the problem and then build a support pocket with relevant sub-process definition to resolve the customer problem. This pocket of flexibility is modeled in the process model as "configurable support pocket" sub-process.

Fig. 13 shows a snapshot of the modeling tool that is used to model the process definition and associated pocket of flexibility fragments. These fragments will be used to build a pocket of flexibility sub process to replace "configurable process model."

Step 2: The process is verified for structural errors (FlowMake consists of a verification engine that implements two verification approaches [6]). Typically, the validation of the given constraint set takes place at this time. However, in the example under consideration, this step in not required.

Step 3: The process definition created above is uploaded to the workflow engine. This process model is now ready for deployment.

Step 4: For each case of the process model, the user or application would create an instance of the process model. On instantiation, the engine creates a copy of the process definition and stores it as an *instance template*. This process instance is now ready for execution.

Step 5: The scheduled process activities of the newly created instance are assigned to performers (workflow users) through work lists and activity execution takes place as usual, until the special purpose BUILD activity is encountered.

Fig. 14 shows a screen snapshot of the simulation prototype that we have developed as a proof of concept to build and test process models with pockets of flexibility. It is showing the status graph of the process instance. The green color of the "Level 2 Support" shows that it has been commenced by a level 2 staff.

Step 6: On triggering, the build activity is assigned to a special workflow user, who may be defined as an "expert" user, since he/she is responsible for defining (or revising) the sub-process object.

The strength of this approach lies in the user's ability to dynamically build instance templates, and still have the confidence that the built template is in accordance with prescribed business process constraints. This not only integrates the process of change into the workflow process itself, but also allows domain experts (who would typically have a greater understanding of the business requirements) to define a process for a given business case/workflow instance, which is reflective of the particular requirements of that case; thereby introducing a process improvement not always possible in generic, pre-defined approaches.

Fig. 15 shows one possible composition of a "configurable support pocket." In this example, we use the process modeling tool to build the pocket and export to the workflow enactment engine for execution. One may envisage, a more friendly and/or domain-specific interface for the support of template building since it is undertaken by domain users rather than workflow designers. A wizards-based approach for example may be utilized to progressively build and reduce the build options based on given constraints and what the user chooses.

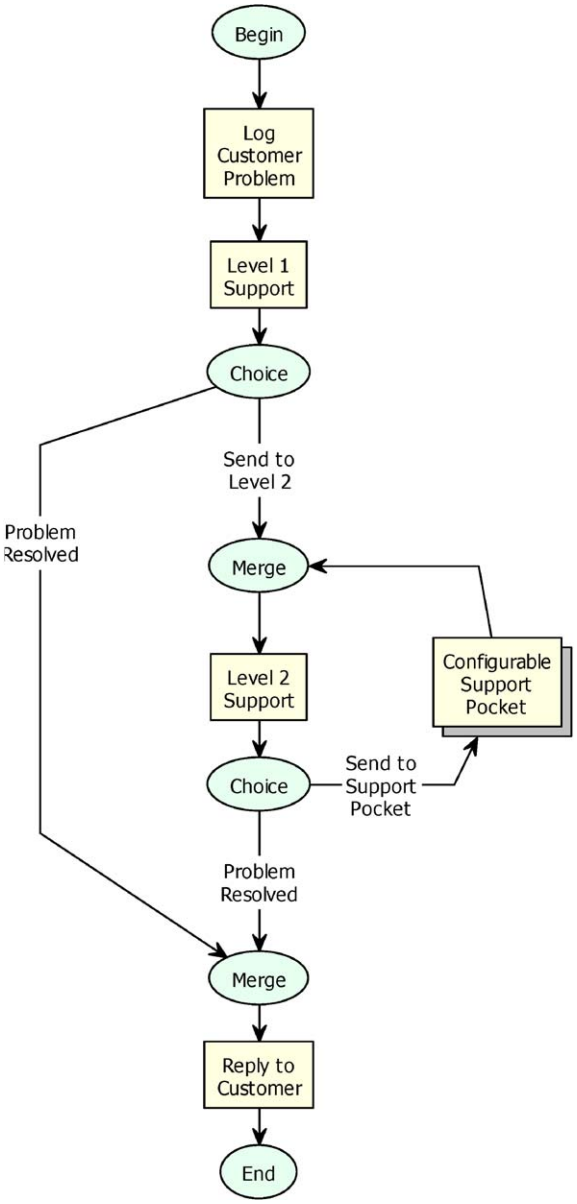


Fig. 12. Specification of the flexible workflow.

Step 7: The next step is to verify the new template, to ensure that it conforms to the correctness properties of the language as well as the given constraints (where applicable). Template verification algorithms have also been implemented [9] for the five classes of constraints discussed in Section 4. The defined and verified sub-process is incorporated into the instance template.

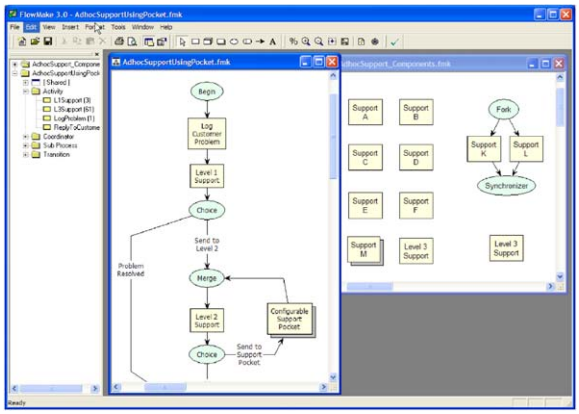


Fig. 13. Modelling tool to model process models and pockets.

Step 8: When the execution of build activity is completed, the newly defined (or revised) sub-process will be invoked.

Fig. 16 shows the work item details for level 2 staff. In this form, level 2 staff will identify the correct sub-process for the pocket (ConfigurableSupportPocket_090_), that had been modeled using the modeling tool.

Execution will continue in typical mode, until the next build activity is encountered or the instance is completed.

7.2. Generalization to ad hoc modification

An interesting and beneficial outcome of the above approach is that ad hoc modifications can also be provided through essentially the same functionality. Ad hoc modification means that any unexecuted part of the instance template may be modified at runtime. This is possible since the workflow engine provides the facility to modify instance templates even in the absence of a dedicated pocket.

The main difference is that in ad hoc modification, the change takes place after instantiation, whereas, in case of a pocket, the change takes place on the sub-process which has not been instantiated as yet. However, it is important to point out that we advocate the approach of a dedicated pocket because it provides greater control over allowable changes at runtime.

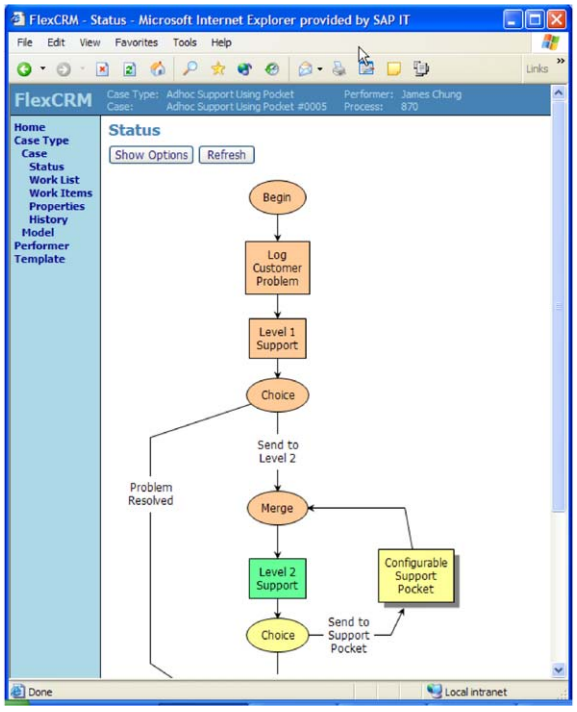


Fig. 14. Execution of the process instance and build activity.

8. Related work

Workflow technology has typically been deployed in domains with processes that have obvious coordinative requirements. As such, the flow of control and data offered by this technology is easily mapped to process effectiveness. However, there is growing evidence that workflow technology must deliver in domains where coordinative requirements are less obvious, dynamic, prone to frequent change or exceptions. As such, research and development in this area has been diverse, and spanned several dimensions of consideration [34]. We identify below three dimensions of change in workflow management, and report on the literature in each area.

8.1. Dynamism

The first dimension represents dynamism — which is the ability of the workflow process to change when the business process evolves. This evolution may be slight as for process

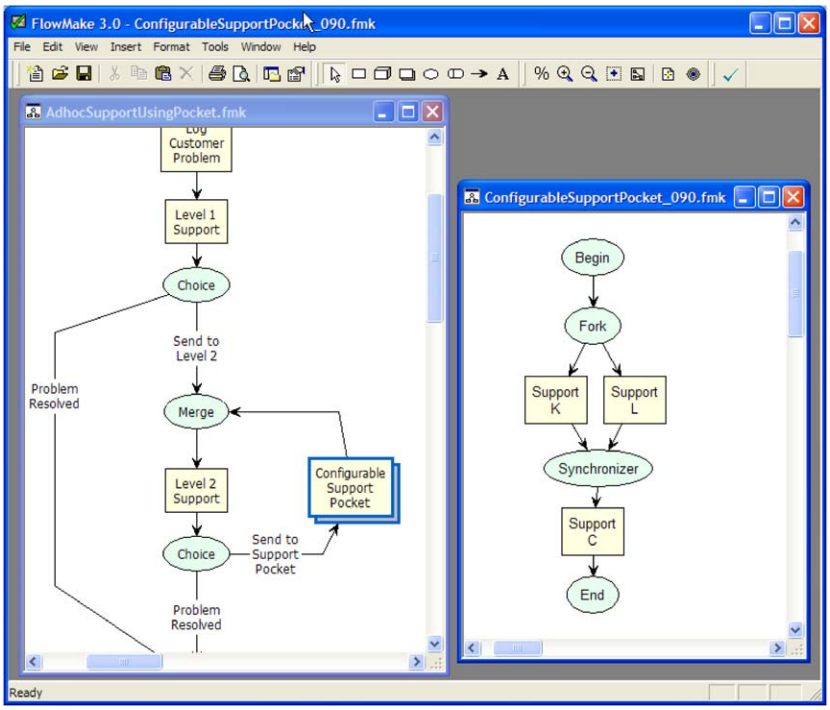


Fig. 15. Building configurable support pocket.

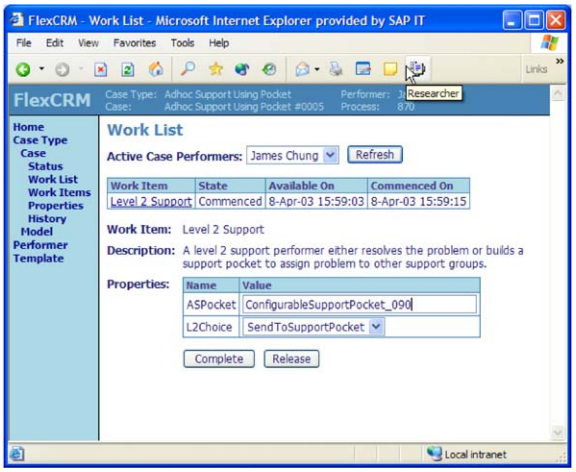


Fig. 16. Work item for selecting built pocket.

improvements, or drastic as for process innovation or process reengineering. In any case, the assumption is that the workflow processes have predefined models, and business process change, causes these models to be changed. The biggest problem here is the handling of active workflow instances, which were initiated in the old model, but need to comply now with the new specification. The issue of compliance is rather a serious issue, since potentially thousands of active instances may be affected by a given process change. Achieving compliance for these affected instances may involve loss of work and therefore has to be carefully planned [10,11].

A typical example of dynamic workflows can be found in university admissions. Consider a scenario of a large tertiary institute that processes thousands of admission applications every year. The procedure for application, review and acceptance is generally well defined and understood. Suppose that the office of postgraduate studies revises the admission procedure for postgraduate students, requiring all applicants to submit a statement of purpose together with their application for admission. To implement this change, there can be two options available; one is to flush all existing applications, and apply the change to new applications only. Thus, all existing applications will continue to be processed according to the old process model. This requires the underlying workflow system to at least provide some version management support [12]. The second option to implement the change is to migrate to the new process. It may be decided that all applicants, existing and new, will be affected by the change. Thus all admission applications, which were initiated under the old rules, now have to migrate to the new process. This migration may involve addition of some transition workflow activities as well as rollback activities. Defining the migration strategy is a complex problem and has been the target of extensive research in this area [10,13–17].

8.2. Adaptability

The second dimension of change is adaptability — which is the ability of the workflow processes to react to exceptional circumstances. These exceptional circumstances may or may not be foreseen, and generally would affect one or a few instances. Of course the handling of exceptions, which cannot be anticipated, is more complex. However, a large majority of exceptions can be anticipated [18,19], and by capturing these exceptions, the adaptability of the workflow is promoted. In fact, unless these exceptions are captured within the workflow model, their handling will continue to be done outside of the system, in the form of "system workarounds", the consequences of which may come in conflict with process goals. However, the complete set of exceptions for a given process can never be captured, thus dealing with unanticipated (true) exceptions will always be an issue [7,20,21].

Using the same example as before, we can consider dealing with an admission application for a student with a multi-disciplinary background. For example, a student with a background in microbiology may be applying for a degree in IT. The review of this application may require the services of an academic outside the offering department. This may be rare but, if captured within the process model, could be handled within the workflow system.

Another example, which represents a true (unanticipated) exception, can be found in the employment workflow. An employment instance may have reached a stage where even the letter of intent has been issued. If at that time the

organization issues a spending freeze, the active instances of the employment workflow will have to be dealt with, requiring rollback and/or compensation activities, even though the original employment workflow remains unchanged.

8.3. Flexibility

The third dimension is flexibility — which is the ability of the workflow process to execute on the basis of a loosely, or partially specified model, where the full specification of the model is made at runtime, and may be unique to each instance. This dimension of change is the focus of this paper.

Essentially, flexible processes are characterized by a lack of ability to completely predict and hence define coordinative requirements. This could be due to the limitations of a prescriptive, or an explosive number of instance types. Greater support for flexibility using the prescriptive model may be possible through extensions to the workflow definition language. Possibly as a response to commercial concerns, work has been undertaken to extend the modeling language to support more complex routing and control primitives not possible with the existing basic constructs [17]. For example, the basic OR-Join (merge) construct could be extended to allow the subsequent activity to activate once N paths of the M paths converging into the merge have completed. Completion of the remaining paths would be ignored. The various contemporary workflow management systems support different levels of expressive power in the workflow definition languages developed for them to capture the more complex requirements that recur frequently in business processes. Given the fundamental differences this introduces to a workflow management system, use of these language extensions will have implementation consequences resulting in the loss of genericity and limited verification support. The work presented in this paper differs from the work done on advanced patterns, where the goal is to increase the expressability of the language. However, this paper proposes an approach that advocates flexibility through runtime modification.

Another related area of research is the automated composition of process models. This can be specifically found in the context of automated composition of web services (see e.g. [22–25]. To position this work appropriately, we would like to point out that automatically composing process models is not the focus of this work since we do not make any underlying assumptions on the availability of contextual information required for such automated activity requiring no or minimal user intervention. Instead, in our approach, the knowledge worker drives the process of dynamically composing the process to meet specific needs of individual process instances. Although, the work of the knowledge worker is supported by establishing constraints which prevent the building of disallowed compositions, the approach is still end-user (human) centric. This approach is thus distinct from related work reported for web service, where the focus is on the automated composition.

Perhaps the most relevant area of application exists in CSCW, where significant work already exists, and workflow concepts have been used to support non-coordinative (collaborative) processes. Ref. [1] speaks of ad hoc workflows, where the process cannot be completely defined prior to execution. Ref. [26] also discusses the coordination of collaboration intensive processes. Although, the complete relaxation of coordination, to support ad hoc processes is not conducive to the processes targeted by our work, structured ad hoc workflows, which are identified as ad hoc workflows where patterns can be derived from the activities in the process as a result of underlying rules to achieve certain goals, are the foundation for a related workflow paradigm shift. The paradigm shift allows the workflow system to derive the workflow incrementally from workflow components and avoids the need to pre-define the process prior to enactment [27]. The completion of a structured ad hoc workflow instance allows flows to be derived for other instances of workflows that share the same process rules. Although this work is concerned with a connector facility to interconnect workflows from manual and automated departments the concept of defining parts of a process incrementally rather than enacting on a pre-defining process is worth mentioning. However, the development of this concept to address the

modeling of processes that cannot be eloquently pre-defined contains significant differences as a result of the rules being more explicit and consistent across instances and the fragments having a much smaller granularity.

Within well-defined fixed processes the need to support dynamic change as the business process evolves in response to competitive and regulatory influence, still exists and so does the need to handle exceptions to cater for instances that cannot be anticipated. Another aspect of work in this area is based on the concept that an exception may result in a change in the level of specificity of the process. For example, a highly specified process of processing customer orders may move to a highly unspecified collaborative process involving emails and consultations as a result of strike conditions at a particular manufacturing plant [28]. The developments supporting these dimensions of change are based on the underlying assumption that the change is exceptional. In the processes under discussion here the change is inherent in the process. This is a fundamental difference. Consequently, the overheads of these developments are too large to implement on a process that is to be dominated by change that cannot be pre-defined. Below we present a summary of some work reported along these lines.

The concept of using a rule-based method for modeling workflows is described in [29]. The approach takes a rule-based description of a business process and transforms it, by applying several refinement steps, to a set of structured rules which represent the business process at different levels of abstraction that is a rule-based workflow specification. This approach is primarily directed towards the development of coordinated processes that span enterprise boundaries by providing a layered approach that separates the transformation of business (sub-) processes and the derivation of workflow specifications and does not address the issue of catering for processes that cannot be eloquently pre-defined.

To accommodate change in organizational procedures, tighter integration between the workflow specification and the enactment modules is recommended [30]. Under such a framework the change itself can be treated as a process that can be modeled and enacted. The modeling associated with this work represents the elements of a workflow such as the process and its activities and rules as object classes. It also supports a variety of pre-defined change schemes, which replace sections of the old version of the procedure affected by the change. In particular, an ad hoc scheme to support changes where the change specification is precisely completed after the change process is enacted. However, the tighter integration between specification and enactment, and the enactment on a partial specification still relies on a prescriptive definition of the processes rather than a rule-based generation of valid models.

The object-orientated paradigm also appears in enterprise modeling where companies are restructured around a number of business operations and processes to accommodate process design change and facilitate the synthesize of business processes out of components [31].

9. Conclusions

Inspite of many good results from the research community, the issue of flexibility in workflows remains a problem in commercial solutions, and a limiting factor in the wider scale applicability of this technology.

It is apparent that change is an inherent characteristic of today's business processes. In this paper, we present an approach that recognizes the presence of change, and attempts to integrate the process of defining a change into the workflow process itself. Our basic idea is to provide a powerful means of capturing the logic of highly flexible processes without compromising the simplicity and genericity of the workflow specification language. This we accomplish through pockets of flexibility in workflow specifications, which allow workflow processes to be tailored to individual instances at runtime. This approach is human centric and empowers the knowledge worker to dynamically adapt a workflow instance in accordance with its requirements but under some degree of control as dictated by the build constraints.

A fundamental feature of the pockets framework is specification of build constraints which essentially control dynamic building while maintaining a desirable degree of flexibility. One can observe that the design of an appropriate means to facilitate the specification of build constraints is an interesting and challenging requirements engineering issue. This paper presents foundation concepts on constraint specification, conflicts, redundancy and verification. The set of five basic constraint types identified can provide a flexible means of definition for a large number of business processes. This set, by no means, claims to provide a means to capture any business logic. Extensions to the set may be envisaged, although it is arguable if such a complete and generic set can be found, and hence achieving flexibility still remains a matter of degree.

Although, this paper presents and applies the concepts of constraints specification and pockets of flexibility in the area of workflows involving human and automated activities within an organization, the underlying principles are applicable to a much wider domain of business process management. Now, even more than before, there is a need to provide configurable and flexible solutions to business process management in light of emerging concepts such as dynamic service composition, and process management in e-business where outsourcing and changing market conditions is the norm. The challenge is to come up with approaches that can satisfy static and pre-defined as well as dynamic and ad hoc business process requirements within a unified process management environment. We believe the framework presented in this paper can provide such a foundation to build next generation flexible business process management solutions.

Acknowledgements

The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

Appendix A

Let $G = <N, F>$ be a graph where N: finite set of nodes, F: flow relation $F \subseteq N \times N$

$\forall n \in N$, NodeType: $n \rightarrow$ {Coordinator, Task}

$N = C \cup T, C \cap T = \phi$ where C: set of coordinator nodes, T: set of task nodes

$\forall n \in C$, CoordType: $n \rightarrow$ {Fork, Synchronize, Choice, Merge, Begin, End, Do, While}

Let P be a directed path in G, such that

$P = \{n1, n2, \ldots, nk\}, \quad (ni, ni+1) \in F \quad for \quad i = 1, 2, \ldots, k-1$

Let Nat: set of natural numbers, NId: set of node identifiers

$\forall n \in N$, $I : N \rightarrow Nat$, $I(n)$: number of incoming flows for node n

$\forall n \in N, O : N \rightarrow Nat, O(n)$: number of outgoing flows for node n

$\forall n \in N, ID : N \rightarrow NId$, ID($n$): identifier for node n

$\forall f \in F$, FromNode: $F \rightarrow N$, FromNode(f): from node of f

$\forall f \in F$, ToNode: $F \rightarrow N$, ToNode(f): to node of f

$\forall n \in N$, Inflow(n) $= \{f$ s.t. $f \in F$ and ToNode(f) $= n\}$

$\forall n \in N$, Outflow(n) $= \{f$ s.t. $f \in F$ and FromNode(f) $= n\}$

A workflow is a directed acyclic graph (DAG) $\mathbf{W} = <\mathbf{N}, \mathbf{F}>$ such that

$\exists n \in C$, s.t. $I(n) = 0 \wedge (\neg \exists m \in N, \; s.t. \; I(m) = 0 \wedge m \neq n)$, we call this Begin Node $n0$ and CoordType ($n0$) = Begin

$\exists n \in C$, s.t. $O(n) = 0 \wedge (\neg \exists m \in N, \; s.t. \; O(m) = 0 \wedge m \neq n)$, we call this End Node nf and CoordType (nf) = End

$\forall n \in N$, $\exists$ P, s.t. $P = \{n0, \ldots n, \ldots, nf\}$

$\forall n \in C$, $I(n) \geqslant 2 \vee O(n) \geqslant 2$ where $n \neq n0$ and $n \neq nf$

$\forall n \in T$, $I(n) + O(n) > 1$ where $n \neq n0$ and $n \neq nf$

$\forall n \in T$, TaskType: $T \rightarrow$ {Activity, SubProcess}, Activity represents a single task and SubProcess represents nesting.

Fig. 17. gives the graphical representation of the modeling constructs used in the above language. The modeling constructs are further defined in the subsequent sections.

Sequence: Sequence execution takes place when a task or coordinator in the workflow is triggered

after the completion of another node. That is, $\forall n \in N$, $I(n) \leq 1 \vee O(n) \leq 1$, n is part of a sequential order. Tasks $T1$ and $T2$ show a sequential configuration in the Fig. 17.

Fork: The fork coordinator or *AND-SPLIT* facilitates the concurrent triggering of all nodes on its outgoing flows. A fork coordinator n will have $O(n) \geq 2$. In Fig. 17, the fork triggers tasks $T3$ and $T4$. Note that, the concurrent triggering does not necessarily indicate the concurrent execution or completion of these tasks. In fact, the tasks that reside on the multiple branches of a fork's outgoing flows, represent a lack of control dependency. That is, the tasks $T3$ and $T4$ have no control flow dependency on each other. In other words, these tasks will be initiated at one time, but the execution and completion of one, will not impact on the execution and completion of the other.

Synchronization: The synchronizer pairs with the fork, and represents an *AND-JOIN*. The synchronizer coordinator waits for all incoming flows to be triggered, before allowing the control flow to progress further. Thus, it synchronized multiple parallel branches of control into a single flow. The synchronizer n will have $I(n) \geq 2$. In Fig. 17, the synchronizer waits for the task T4 on one branch, and the tasks $T3$ followed by $T5$ *or* (due to the choice-merge construct, see next section) $T3$ followed by $T6$ on the other branch.

Choice: The choice coordinator or *OR-SPLIT* represents alternative execution paths within the process. For example in Fig. 17, only one of the tasks $T5$ or $T6$ will be executed in a given instance of the workflow. Like fork, a choice coordinator n will also have $O(n) \geq 2$. Generically speaking, the choice construct represents a point in the workflow process where one of several branches is chosen based upon the results of a condition. Each condition is a Boolean expression based on workflow relevant instance data. Exclusivity of the conditions must be ensured by the workflow designer.

Merge: The merge construct, also called *OR-JOIN*, is the opposite of the choice construct, and merges the multiple branches of a choice construct. A merge thus allows the process to proceed when any one of its incoming flows is triggered. In other words, it allow alternate branches to come together without synchronization. Like a synchronizer, a merge will have $I(n) \geq 2$.

Nesting: Nesting is used to simplify the workflow specifications through abstraction. For each execution of the nested task $T8$, the underlying workflow is executed. Nesting, or the use of subprocesses, also promotes reusability. Thus $\forall n \in T$, TaskType(n) = SubProcess, n represents nesting.

Iteration: A set workflow tasks ($T7$) may be executed repeatedly. Iterative structures are generally represented through a do–while or repeat–until structure, and introduce cycles in the workflow graph.

Initiation: A workflow model in this language will have a single begin. The begin node n0 is represented by the following: CoordType ($n0$)= Begin $\wedge$ I(n0) = 0 $\wedge$ ($\neg \exists$m $\in$ N, $s.t. I(m) = 0 \wedge m \neq n0$). Typically, the execution of the task(s) immediately following the begin node (for example task $T1$ in Fig. 17.) will represent the first activity in the process. The begin node is simply a demarcation for the beginning of the process.

Termination: A process may have multiple termination tasks due to the presence of choice and/or fork. There are two approaches in this regard: the first one is where all multiple branches of a process are joined (merge/synchronize) before the end, thus resulting in a single termination node. Another approach is to not merge/synchronize to

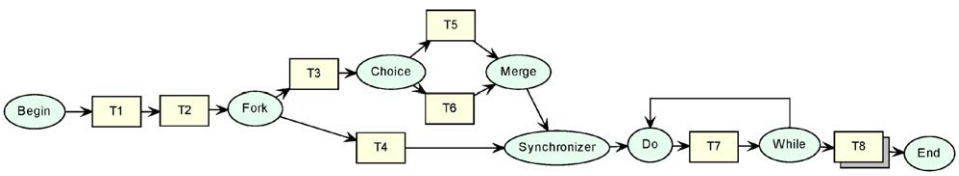


Fig. 17. Modelling constructs.

these branches, thus resulting in multiple ending tasks. Each approach has its advantages, but the single end is preferred due to its simplicity and ease of verification, and will be used in most examples in this thesis. Hence, the termination is represented by a coordinator node nf, such that CoordType $(nf) = End \wedge O(nf) = 0 \wedge (\neg \exists m \in N, s.t.\ O(m) = 0 \wedge m \neq nf)$.

References

[1] M. Klein, C. Dellarocas, A knowledge-based approach to handling exceptions in workflow systems, Workshop on Adaptive Workflow Systems, Conference on Computer Supported Cooperative Work (CSCW), Seattle, USA, November 1998.

[2] S.J. Greenspan, J. Mylopoulos, Capturing more world knowledge in the requirements specification, Proceedings of the sixth International Conference on Software Engineering, Tokyo, Japan, 1982.

[3] S.M. Easterbrook, An Introduction to Formal Modeling in Requirements Engineering Tutorial at the 10th Joint International Requirements Engineering Conference, Essen, Germany, September 2002.

[4] H.W. Nissen, M. Jarke, Repository support for multi-perspective requirements engineering, Inform. Systems 24 (2) (1999) 131–158.

[5] A.W. Scheer, ARIS—Business Process Modeling, 3rd Edition, Springer, Berlin, 2000.

[6] W. Sadiq, M.E. Orlowska, Analysing process models using graph reduction techniques, Inform. Systems 25 (2) (2000) 117–134.

[7] S.R. Rakitin, Software Verification and Validation: a Practitioner's Guide, Artech House, Boston, 1997.

[8] W. Sadiq, M.E. Orlowska, On correctness issues in conceptual modeling of workflows, in: Proceedings of the Fifth European Conference on Information Systems (ECIS '97), Cork, Ireland, June 19–21, 1997.

[9] O. Thomas, Constraint specification and verification in flexible workflows, Honors Thesis, The University of Queensland, October 2002.

[10] F. Casati, S. Ceri, B. Pernici, G. Pozzi, Workflow evolution, in: Proceedings of the 15th International Conference on Conceptual Modeling, ER'96, Cottbus, Germany, Lecture Notes in Computer Science, Springer, Berlin, 1996.

[11] S. Sadiq, O. Marjanovic, M. Orlowska, Managing change and time in dynamic workflow processes, Int. J. Coop. Inform. Systems 9, (1,2), (2000).

[12] G. Joeris, O. Herzog, Managing evolving workflow specifications, Proceedings of the Third IFCIS International Conference on Cooperative Information Systems (CoopIS 98), NewYork, USA, August 1998.

[13] W.M.P. van der Aalst, T. Basten, Inheritance of workflows: an approach to tackling problems related to change, in: Theoretical Computer Science, Vol. 270, Issue 1–2, January 2002.

[14] M. Kradolfer, A. Geppert, Dynamic workflow schema evolution based on workflow type versioning and workflow migration, Proccedings of the Fourth IFCIS International Conference on Cooperative Information Systems (CoopIS99), Edinburgh, Scotland, September 2–4, 1999.

[15] C. Liu, M. Orlowska, Hui Li, Automating handover in dynamic workflow environments, Proceedings of the 10th International Conference on Advances in Information System Engineering (CAiSE 98), Pisa, Italy, June 1998.

[16] S. Sadiq, Handling dynamic schema changes in workflow processes, 11th Australian Database Conference (ADC), Canberra, Australia, January 30–February 3, 2000.

[17] W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, B. Kiepuszewski, Advanced workflow patterns, in: O. Etzion, P. Scheuremann (Eds.), Proceedings of the Seventh IFCIS International Conference on Cooperative Information Systems, CoopIS 2000, Vol. Lecture Notes in Computer Science, 1901, Springer, Eilat, Israel, September 2000, pp. 18–29.

[18] F. Casati, G. Pozzi, Modeling exception behaviors in commercial workflow management systems, Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems (CoopIS99), Edinburgh, Scotland, September 2–4, 1999.

[19] D.M. Strong, S.M. Miller, Exceptions and exception handling in computerized information processes, ACM Trans. on Inform. Systems, 13 (2) (1995) 206–233.

[20] M. Reichert, P. Dadam, ADEPTflex—supporting dynamic changes of workflow without loosing control, J. Intell. Inform. Systems (1998) (special issue on workflow and process management).

[21] S. Sadiq, On capturing exceptions in workflow process models, Proceedings of the Fourth International Conference on Business Information Systems, Poznan, Poland, April 12–13, 2000.

[22] M. Benatallah, M.C. Fauvet Dumas, F. Rabhi. Towards patterns of web services composition. in: S. Gorlatch, F. Rabhi (Eds.), Patterns and Skeletons for Parallel and Distributed Computing, Springer,Berlin, November 2002, (ISBN 1-85233-506-8).

[23] F. Casati, Ming-Chien.Shan, Dynamic and adaptive composition of e-services, Inform. Systems 26 (2001) 143–163.

[24] J. Peer, Towards Automatic Web Service Composition using AI Planning Techniques, August 2003.

[25] M. Thandar Tut, D. Edmond, The use of patterns in service composition, International Workshop on Web Services, E-Business, and the Semantic Web, held in conjunction with CAiSE02, Toronto, Canada, May 27–28, Lecture Notes in Computer Science Vol. 2512, Springer, Berlin, 2002.

[26] D.P. Bogia, S.M. Kaplan, Flexibility and control for dynamic workflows in the worlds environment, Proceedings of ACM Conference on Organizational Computing Systems (COOCS 95), Milpitas, CA. USA, November 1995.

[27] D. Han, J. Shim, Connector-oriented workflow system for the support of structured ad hoc workflow, Proceedings of the 33rd Hawaii International Conference on System Sciences, 2000.

[28] A. Bernstein, How can cooperative work tools support dynamic group processes? Bridging the Specificity Frontier, CSCW'00, Philadelphia, USA, December 2–6, 2000.

[29] G. Knolmayer, R. Endl, M. Pfahrer, Modeling processes workflows by business rules, in: W. van der Aalst, et al. (Eds.), Business Process Management, Lecture Notes in Computer Science, Vol. 1806, Springer, Berlin, 2000, pp. 16–29.

[30] S. Ellis, K. Keddara, G. Rozenberg, Dynamic changes within workflow systems. Proceedings of ACM Conference on Organizational Computing Systems COOCS 95, 1995.

[31] M.P. Papazoglou, W.J. Heuvel Van Den, Configurable business objects for building evolving enterprise models and applications, in: W. van der Aalst, et al. (Eds.), Business Process Management, Vol. 1806, Springer, Berlin, 2000, pp. 16–29.

[32] W. Sadiq, M. Orlowska, On capturing process requirements of workflow based information systems, in: Proceedings of the Third international Conference on Business Information Systems (BIS '99), Poznan, Poland, April 14–16, 1999, Springer, Berlin, Verlag, 1999, pp. 195–209.

[33] S. Sadiq, W. Sadiq, M. Orlowska, Pockets of flexibility in workflow specifications, 20th International Conference on Conceptual Modeling, ER'2001, Yokohama, Japan, 2001.

[34] S. Sadiq, On dynamically changing workflow processes, Ph.D.Thesis, The University of Queensland, December 2002.