Process Flexibility Patterns
Nataliya Mulyar, Wil M.P. van der Aalst, and Nick Russell

Department of Technology Management, Eindhoven University of Technology
GPO Box 513, NL5600 MB Eindhoven, The Netherlands
{n.mulyar, w.m.p.v.d.aalst,n.c.russell}@tue.nl

Abstract. The topic of process flexibility has become a center of attention from both commercial and research institutions as understanding of requirements for business processes capable to adapt their behavior to changes in an external environment is crucial for supporting organizations operating in a dynamic context. In this paper, a set of requirements addressing the aspect of the process flexibility are identified and presented in the form of the Process Flexibility Patterns. Each of the patterns facilitates one of the five process flexibility types: flexibility by design, flexibility by deviation, flexibility by underspecification, flexibility by momentary and permanent change. The semantics of the patterns is illustrated by means of Colored Petri Nets as functionality augmenting the behavior of flexible process engine. The patterns are used to evaluate the flexibility of a selection of offerings.

Keywords: Process Flexibility Patterns, Colored Petri Nets, change, deviation, underspecification, design.

1 Introduction

The manner in which desired and/or possible behavior is captured in the process definition, greatly influences the flexibility given to a user in selecting a suitable execution path during process execution. As a consequence of changes in the operating environment, the execution of a process may need to be steered in the right direction in order to achieve the desired execution order of tasks. This becomes possible when multiple alternative paths have been foreseen during design-time. In situations, where the required execution path cannot be found, the need to deviate from the prescribed execution path may arise, necessitating changes to be made to process definition on the spot. In this paper, we focus on the aspect of process flexibility. We distinguish different types of process flexibility and present them in the form of a taxonomy. For each of the flexibility types identified, we systematically analyze requirements for process flexibility and describe them in a language-independent and precise manner using a pattern-based approach. We then utilize the process flexibility patterns identified in order to evaluate the functionality of selected PAISs.

This paper is organized as follows. Section 2 presents a taxonomy for process flexibility. In Section 3 we describe 34 process flexibility patterns that have been identified. In Section 4, we use these patterns to evaluate the support for process flexibility in a selection of contemporary PAISs. In Section 5 we discuss related work. Finally, Section 6 concludes this paper.
2 Taxonomy of process flexibility

In this section, we present a comprehensive description of five distinct approaches that can be taken to facilitate flexibility within a process. This is a variant of our earlier taxonomy presented in [58]. These approaches are presented in the form of a taxonomy which defines each of them in detail. Figure 1 shows five types of flexibility: flexibility by design, flexibility by deviation, flexibility by underspecification, and two types of flexibility by change, i.e. flexibility by momentary change and flexibility by permanent change. The flexibility types identified represent orthogonal dimensions and are intended to operate independently of each other. The taxonomy is applicable to both classical (imperative) and constraint-based (declarative) process specifications. Note however that it mainly concentrates on the control-flow perspective, and does not address issues related to other perspectives such as data handling and resource assignment.

![Fig. 1. Flexibility types](image)

Each of the five flexibility types aim at improving the ability of business processes to respond to changes in their operating environment without necessitating a complete redesign of the underlying process definition, however they differ in the timing and manner in which they are applied. Figure 2 shows how the flexibility types identified can be classified in terms of the moment at which the need for a specific flexibility type is recognized, the moment at which steps for facilitating anticipated flexibility are taken, and their relation to actual execution. The scope of impact associated with each of the flexibility types is defined as follows. When a decision related to the realization of the specific flexibility type affects only a particular process instance, we map it at the process instance level; when all existing process instances are be affected, we map the flexibility type at the type level.

As Figure 2 shows, the need for flexibility by design is recognized on the type level at design-time. All decisions related to the incorporation of this type of flexibility into a process definition are both taken and realized before process initiation. Thus there is a direct relation between the flexibility incorporated into a process definition at design-time and the way it influences process execution. Flexibility by underspecification is recognized at design-time when a process definition is being created, however its complete realization is postponed until runtime. For this flexibility type, the desired behavior is set for a specific process
instance, during the course of its execution. Note that different instances of the same process may have distinct realizations. Flexibility by deviation is recognized at run-time at the moment of (or shortly after) process initiation (for a specific process instance). The same classification applies to the flexibility by momentary change. The main difference between these two flexibility types is that flexibility by momentary change results in the modification of the process definition associated with a given process instance before it can be executed, whilst realization of flexibility by deviation leaves the process definition unaffected. Finally, flexibility by permanent change can be seen as redesign of the process definition at the type level at run-time. This flexibility type is facilitated by performing changes to the process definition, after which it can be executed (potentially impacting all existing process instances).

Each of the individual flexibility types introduced above are now described in detail using a standard format including: a motivation, definition, scope and realization options. We start with flexibility by design.

2.1 Flexibility by design

Motivation When a process operates in a varying operational environment it is desirable to incorporate support for the various execution alternatives that may arise during execution within the process definition. At runtime, the most appropriate execution path can then be selected from those encoded in the design time process definition.

Definition *Flexibility by Design* is the ability to incorporate alternative execution paths within a process definition at design time allowing for selection of the most appropriate execution path to be made at runtime for each process instance depending on the circumstances encountered.

Scope Flexibility by design applies to any process which has multiple predefined execution trace and the choice of these traces can be influenced at run-time.

Realization options The most common options for realization of flexibility by design, such as parallelism, choice, iteration, etc. are thoroughly described by the control-flow patterns in [55] and have been widely observed in a variety of imperative languages. We argue that these concepts are equally applicable in a declarative
setting which has a much broader repertoire of constraints that allow for flexibility by design. Note that both approaches really differ with respect to flexibility. To increase flexibility in an imperative process, more execution paths have to be modeled explicitly, whereas increasing flexibility in declarative processes is accomplished by reducing the number of constraints, or weakening existing constraints. A declarative model is most flexible when it does not have any constraints [45]. In this case, all of its constituent tasks can be executed in any order, any number of times.

Describing all possible execution paths in a process definition completely at design-time may be either undesirable from the standpoint of model complexity or impossible due to an unknown or unlimited number of possible execution paths. The following flexibility types provide alternative mechanisms for process flexibility.

2.2 Flexibility by deviation

Motivation Some process instances need to temporarily deviate from the execution sequence prescribed by their associated process definition in order to accommodate changes in the operating environment encountered at runtime. The overall process definition and its constituent tasks remain unchanged.

Definition Flexibility by Deviation is the ability for a process instance to deviate at runtime from the execution path prescribed by the original process without altering its associated process definition. The deviation can only encompass changes to the execution sequence of tasks in the process for a specific process instance, it does not allow for changes in the process definition or the tasks that it comprises.

Scope The concept of deviation is particularly suited to the specification of process definitions which are intended to guide possible sequences of execution rather than restrict the options that are available (i.e., they are descriptive rather than prescriptive). These specifications contain the preferred execution of the process, but other scenarios are also possible.

Realization options The manner in which deviation is achieved depends on the specification approach utilized. Deviation can be seen as varying the actual tasks that will be executed next, from those that are implied by the current set of enabled tasks in the process instance. In imperative languages this can be achieved by applying deviation operations such as redo, task skip, etc. For declarative approaches, deviation occurs through the violation of (optional) constraints.

2.3 Flexibility by underspecification

Motivation When specifying a process definition it might be foreseen that during run-time execution more execution paths are needed which must be incorporated within an existing process definition. Furthermore, often only during the execution of a process instance does it become clear what needs to be done at a specific point in a process. When all execution paths cannot easily be defined in advance in the
standard way, it is useful to be able to execute an incomplete process definition and dynamically add process fragments expressing missing scenarios to it.

**Definition** Flexibility by Underspecification is the ability to execute an incomplete process definition at run-time, i.e., one which does not contain sufficient information to allow it to be executed to completion. Note that this type of flexibility does not require the definition to be changed at run-time, instead the definition is completed by providing a concrete realization for the undefined parts as they are encountered at run-time.

**Scope** The concept of underspecification is mostly suited to processes where it is clearly known in advance that the process definition will have to be adjusted or has high variability at specific points, although the exact content at this point is not yet known (and may not be known until the time that an instance of the process is executed). This approach to process design and enactment is particularly useful where distinct parts of an overall process are designed and controlled by different work groups, but the overall structure of the process is fixed. In this situation, it allows each of the individual groups to retain some degree of autonomy in regard to the tasks that are actually executed at runtime in their respective parts of the process, whilst still complying with the overall process definition.

**Realization options** An underspecified process definition is deemed to be one which is well-formed but does not have a detailed definition of the ultimate realization of every task. An incomplete process definition contains one or more so-called placeholders. Placeholders are nodes which are marked as underspecified (i.e., their content is unknown or is not fixed) and whose content is specified during the execution of these placeholders. We distinguish two types of placeholder enactment:

- **Late binding**: where the realization of a placeholder is selected from a set of available process fragments. Note that to realize a placeholder one process fragment has to be selected from an existing set of predefined process fragments or defined via ripple-down rules (RPR)\(^1\). This approach is limited to selection, and does not allow a new process fragment to be constructed.
- **Late modeling**: where a new process fragment is constructed in order to realize a given placeholder. Not only can a process fragment be constructed from a set of currently available process fragments, but also a new process fragment can be developed from scratch. Therefore late binding is complemented by late modeling. Some approaches [57] limit the construction of new definitions by (declarative) constraints.

For both approaches, the realization of a placeholder can occur at a number of distinct times during process execution. Here, two distinct moments of realization are recognized:

---
\(^1\) If late binding is used based on rules (see RDR in Worklets [10]), then one could argue that this is just an XOR-split. In principle, this is correct, however we still consider this flexibility by underspecification.
- **before placeholder execution**: the placeholder is realized at the commencement of a process instance or during execution before the placeholder has been executed for the first time.
- **at placeholder execution**: the placeholder is realized when it is executed.

Placeholders can be either realized once, or for every execution of the placeholder. We distinguish two distinct realization types:

- **static realization**, where the process fragment chosen to realize the placeholder during the first execution is used to realize the placeholder for every subsequent execution.
- **dynamic realization**, where the realization of a placeholder can be chosen for each execution of the placeholder.

The following two types of flexibility facilitate the incorporation of execution behaviors at run-time that have not been anticipated at design-time. Depending on the extent to which the realization of flexibility by change applies, we can distinguish flexibility by momentary change and flexibility by permanent change. These two types of flexibility by change are described in detail below.

### 2.4 Flexibility by momentary change

**Motivation** In some situations, when executing a given process instance, the need to achieve specific behavior that has not been foreseen during design-time may arise. Sometimes such behavior cannot be facilitated by temporary deviations from the existing process definition, but requires the process definition for a particular process instance to be modified.

**Definition** *Flexibility by Momentary Change* is the ability to modify a process definition at run-time such that the process definition associated with a given process instance is temporarily amended in order to realize previously not foreseen behavior.

**Scope** Flexibility by change allows processes to adapt to transient changes that are identified in the operating environment. Changes are introduced at the level of the process instance and do not affect other process instances.

**Realization options**

- **Moment of allowed change** specifies the moment at which changes can be introduced in a given process instance.
  
  - **Entry time**: changes can be performed at only the moment the process instance is created. After the process instance has been created, no further changes can be introduced to the given process instance. Momentary changes performed at entry time affect only a single process instance.
  - **On-the-fly**: changes can be performed at any time during process execution. Momentary changes performed on-the-fly correspond to customization of a given process instance during its execution.
2.5 Flexibility by permanent change

**Motivation** In some cases, events may occur during process execution that were not foreseen during process design. Sometimes these events cannot be addressed by temporary deviations from the existing process definition, but require the addition or removal of tasks or links from the process definition on a permanent basis. This may necessitate changing the process definition for all currently executing instances.

**Definition** *Flexibility by Permanent Change* is the ability to modify a process definition at run-time such that all currently executing process instances are migrated to a new process definition and all new process instances are routed to the new process definition. The process definition constructed at design time is modified and all process instances need to be transferred from the old to the new process definition.

**Scope** Flexibility by change allows processes to adapt to changes that are identified in the operating environment. Changes may be introduced at type level resulting in the modification of the process definition.

**Realization options**

*Moment of allowed change* specifies the moment at which changes can be introduced in a given process instance or a process definition.

- **Entry time**: changes can be performed only at the moment the process instance is created. After the process instance has been created, no further changes can be made to the given process instance. The result of permanent change performed at entry time is that all new process instances have to be started after the change to the process definition has been performed, and no existing process instances are affected (they continue execution according to the process definition with which they are associated).

- **On-the-fly**: changes can be performed at any time during process execution. Permanent changes performed on-the-fly impact both existing and new process instances. The new process instances are created according to the new process definition, while the existing process instances may need to be migrated from the existing process definition to the new process definition.

*Migration strategy* defines what to do with running process instances that are impacted by a permanent change.

- **Forward recovery**: affected process instances are aborted.
- **Backward recovery**: affected process instances are aborted (compensated if necessary) and restarted.
- **Proceed**: changes introduced are ignored by the existing process instances. Existing process instances are handled the old way, and new process instances are handled the new way.
- **Transfer**: the existing process instances are transferred to a corresponding state in the new process definition.

We have described the five flexibility types by illustrating the motivation for each of them, giving their definition and defining the scope that each of them impact.
in a process. However, when it comes to the realization of these flexibility types in practice, it is unclear what the typical constructs required for realizing particular types of flexibility, what the semantics of these constructs and in which situations and domains they are applicable. In the next section, we address this issue by defining a set of process flexibility patterns.

3 Process Flexibility Patterns

In this section, we present the set of flexibility patterns that have been identified. The patterns are divided into eight groups (see Figure 3). Each group addresses a common problem and is intended to describe a specific aspect of process flexibility such as flexible initiation, flexible termination, flexible reordering, flexible selection, flexible elimination, flexible extension, flexible concurrency and flexible repetition. For instance, flexible elimination patterns aim to facilitate flexibility by avoiding the execution of a particular task. This can be done already during design-time by incorporating a by-path path, at run-time by skipping the execution of currently enabled task, or by removing this task from the process definition.

Each of the pattern groups consist of at most five patterns, and each pattern corresponds to one of the five flexibility types: flexibility by design, flexibility by deviation, flexibility by underspecification, flexibility by momentary change or flexibility by permanent change. This provides an overall structure of $8 \times 5$ possible patterns. However, not all desired aspects of flexibility can be mapped to the notions associated with the five flexibility types, therefore some pattern groups contain less than five patterns. When describing each of the pattern groups, we will indicate which configurations are meaningful and which are not. Note that the goal of the process flexibility patterns not to fill in the process flexibility matrix, but rather to use it as a guide to meaningful types of flexible execution extensions.

<table>
<thead>
<tr>
<th>Flexible initiation</th>
<th>Flexible termination</th>
<th>Flexible selection</th>
<th>Flexible reordering</th>
<th>Flexible elimination</th>
<th>Flexible extension</th>
<th>Flexible concurrency</th>
<th>Flexible repetition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>Deviation</td>
<td>Underspecification</td>
<td>Momentary Change</td>
<td>Permanent Change</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3. The Process Flexibility Matrix used to position patterns in this paper

3.1 Context assumptions

A process definition consists of a set of tasks and a set of constraints specifying the order in which these tasks have to be executed. Figure 4 shows the graphical no-
tation we will use to visualize a process definition. Typically, a process definition contains a single start task, i.e., a task from which process execution begins, and a single end task, i.e., a task at completion of which process execution terminates. Process elements that are left underspecified at design-time are marked as placeholders. We will use a square with a solid and dashed line to denote placeholders (cf. Figure 4). For some patterns we also need to differentiate individual execution instances. We will use a symbol in the form of a shaded triangle, circle or square to denote (partial) states associated with different process instances (i.e., tokens referring to particular instances). Such a symbol preceding a task in the process definition indicates that this task is enabled for execution.

To show the effect of applying a flexibility pattern to a particular process instance at run-time, both the process definition before and after applying this pattern are visualized. We depict process definitions associated with different process instances separately to show the effect of momentary changes, i.e., changes resulting in modification of the process definition associated with a particular process instance. To denote that changes of a permanent nature, performed at the type level, impact all existing process instances, we show a single process definition common to all process instances before and after the permanent change has been performed. To illustrate different execution paths within a process definition, each task may have associated split and join connectors of XOR or AND type or may have no split/join behavior at all.

![Fig. 4. Process entities](image)

To describe the context in which the flexibility patterns can be used and to explain the operational semantics of our basic notations, we present a basic engine for executing process instances based on the process definitions using the CPN formalism. Figure 5(a) illustrates the top-level view of the basic engine able to handle both concrete and placeholder tasks. Two substitution transitions Engine and Placeholder definition interface correspond to the nets depicted in Figure 5(c) and (b) respectively.
Process definitions for which process instances will be created are stored in the Process definition place. In order to differentiate between distinct models, they are associated with identifiers of type ModelID to refer to a particular process model. The process definitions are of type ProcModel, and contain information about the model id (of type ModelID), the id of the start task, the id of the end task, a list of tasks in the process definitions and a list of arcs between the tasks. The execution of a process starts with the start task specified in the process definition, and completes after the end task has been executed. This engine is implemented based on the assumption that the process instance completes when no tasks are left which still can be executed (this corresponds to the Implicit Termination WCF-pattern described in [55]).

Each task is defined as a tuple of the Task type, specifying the id of the task, the type of the task (i.e. whether it is concrete or underspecified), the type of the join connector and the type of the split connector. The join connector and the split connector are of type ConnectorType, which specify for a given task whether it needs to synchronize multiple incoming branches and produce output to multiple outgoing branches. Where both connectors have value none, the task

![Diagram](image-url)

**Fig. 5.** Basic engine
can be characterized as sequential. The connector values XOR and AND represent XOR/AND-splits and XOR/AND-joins associated with a task. The definition of the main data types used in Figure 5 is visualized in Table 1.

To create a process instance, a user places a token of type ProcInst in the Begin place, specifying the id for the process definition that needs to be populated (i.e. mid of the ModelID type), and the id that has to be assigned to the process instance being created (i.e. pid of the piID type).

A process instance is created by the engine based on the input (mid,pid) provided by the user and the matching model m. The Create process instance transition matches mid, (the id of the model provided by the user), with the id associated with the process definition m available in the Process definition place. Where these match, the createinst(pid,m) function creates a process instance of type ProcInstState. The process instance contains the id for the specific process instance pid, a replica of the process definition m, and the current state st indicating which task is currently enabled. Note that this does not imply that we assume an implementation where the process model is replicated for every instance, i.e., the model could also be passed by reference.

A task in an activated process instance, information about which is stored in the Running instance place, can be executed only if the task is concrete (i.e. the content of the task has explicitly been defined at design-time). If the task has been left underspecified (i.e. it has the placeholder type), first the definition of this task has to be completed by the placeholder engine illustrated in Figure 5(b). The placeholder may be completed by creating a new process fragment, by selecting an element from the repository of predefined process fragments or by composing a new process fragment. By executing one of the transitions Create, Select or Compose, the task that previously had the placeholder type is replaced with the

Table 1. Data types used in Figure 5

| colset ModelID = string; |
| colset piID = int; |
| colset ProcInst = product ModelID * piID; |
| colset TaskID = smallstr; |
| colset TaskType = with concrete | placeholder; |
| colset ConnectorType = with XOR | AND | none; |
| colset Task = product TaskID * TaskType * ConnectorType | ConnectorType; |
| colset Tasks = list Task; |
| colset Link = product TaskID * TaskID; |
| colset Arcs = list Link; |
| colset ProcModel = product ModelID * TaskID | * TaskID | * Tasks | Arcs; |
| colset State = Arcs; |
| colset ProcInstState = product piID * ProcModel * State; |

1Task split
2Task join
3Start task
4End task
desired content and its task type is set to **concrete**. From this moment on, the **Execute task** transition in the basic engine (see Figure 5(c)) may fire when the Boolean function \texttt{existsEnabledTask(pid,m,st)} evaluates to \texttt{True}, indicating that there exist enabled tasks which can be executed.

The task to be executed next is selected non-deterministically from the set of enabled tasks\(^2\). The set of enabled tasks is formed by tasks whose enabling conditions are satisfied. A task without a join connector is enabled if the task preceding it has been executed and the arc connecting these two tasks is present in the state associated with the given process instance (as indicated by attribute \texttt{st} of type \texttt{State}). If a task has a join connector of XOR type, the process instance state must contain at least one enabled arc; otherwise, if the task has a join connector of AND type, all incoming arcs must be enabled and thus be present in the state \texttt{st}. After the task has been executed, the \texttt{ns(pid,m,st)} function determines the new state for the process instance. If the split connector associated with the executed task is of type XOR, one of its output arcs is selected non-deterministically and added to the state. If the split connector is of type AND, then all outgoing arcs are added to the state. The execution of the process instance completes when no enabled tasks are left, i.e. the \texttt{existsEnabledTask(pid,m,st)} function evaluates to \texttt{False}.

We will use the graphical notation described in Figure 4 to depict the flexibility options that are characterized by the flexibility patterns. To describe the operational semantics of the patterns, we illustrate the basic engine depicted in Figure 5 that is enhanced with flexibility extensions specific to the pattern being considered, thus focusing only on the differences between the basic and the enhanced engines.

We will describe each of the process flexibility patterns using the following pattern format:

- **description**: a summary of its functionality;
- **examples**: illustrative examples of its usage;
- **motivation**: the rationale for the use of the pattern;
- **overview**: a graphical notation illustrating the pattern and an explanation of its operation;
- **context**: a detailed operational definition of the pattern in terms of CPN, illustrated as an enhancement of the basic process engine;
- **implementation**: how the pattern is typically realized in practice\(^3\);
- **issues**: problems potentially encountered when using the pattern;
- **solutions**: how these problems can be overcome; and
- **evaluation criteria**: the conditions that an offering must satisfy in order to be considered to support the pattern.

We start with a description of patterns related to the flexible process initiation.

\(^2\) Note that we abstract from work distribution, etc. and hence can consider this choice to be non-deterministic.

\(^3\) We describe the implementation options for Oracle BPEL PM, CIG modeling languages and four systems illustrating different kinds of support for process flexibility (ADEPT1, FLOWer, YAWL, and Declare)
3.2 Flexible initiation

This group of patterns aims to describe flexibility options related to the initiation of a business process, i.e. flexibility considerations that influence the manner in which process instances are created. Figure 6 illustrates the scope of patterns presented in this subsection and their relation to the different types of flexibility outlined in Section 2. We will traverse cells in the highlighted row from left to right and analyze the mapping of the concept of flexible process initiation to each of the flexibility types.

<table>
<thead>
<tr>
<th>Flexible initiation</th>
<th>Design</th>
<th>Deviation</th>
<th>Under-specification</th>
<th>Momentary Change</th>
<th>Permanent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible termination</td>
<td>Alternative entry point</td>
<td>Entrance skip</td>
<td>Undefined entry</td>
<td>Momentary entry change</td>
<td>Permanent entry change</td>
</tr>
<tr>
<td>Flexible selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible rerouting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible elimination</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible extension</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible concurrency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible repetition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 6. Process Flexibility Matrix: flexible initiation

Usually a process definition has a single entry point which triggers the initiation of a new process instance. An example of such a process is shown in Figure 7. The execution of this process starts with task A each time this process is initiated. Such an approach to process definition forces the execution of all associated process instances to start with the same task. In some situations, the execution of the process may need to start from a task other than the nominated start task prescribed by the process definition. For rigid process definitions with a single start task this is impossible, thus the start task must be executed even if it is not required.

Fig. 7. Example of a process definition

Depending on the moment at which the need for an alternative entry-point to a process definition is recognized and the manner in which it is facilitated, we distinguish the following five patterns: Alternative Entry-Points, Entrance Skip, Undefined Entry, Momentary Entry Change and Permanent Entry Change.

Although the goal of all of these five patterns is to allow a user to start execution of a process instance from a task different to the nominated start task, these patterns differ by the moment at which the need for flexibility is anticipated. The Alternative Entry-Points pattern is associated with process design-time (i.e.
all decisions allowing for flexible process initiation are defined before a process instance has been created), while the other four patterns are associated with the run-time execution. In the Entrance Skip pattern the need to deviate from the nominated start task is only anticipated at the moment a process instance is created. In the Undefined Entry pattern the need for flexible process initiation is anticipated at design-time, however its realization is accomplished at run-time. In the Momentary Entry Change and Permanent Entry Change patterns the need for flexible process initiation is anticipated and realized at run-time by changing the process definition at the instance or type level.

Pattern FP-1 ALTERNATIVE ENTRY-POINTS
Description A process definition contains more than one start task, each of which represents an alternative entry point for the process definition. Any of the start tasks can be selected by a knowledgable user at run-time when creating a new process instance causing the process instance to commence from that task. This pattern characterizes a need for flexible process initiation that is anticipated at design-time, thus it corresponds to the flexibility by design type.

Examples
– The medical processes for handling patients in a hospital are defined in such a way that a patient may enroll into any stage of the treatment process depending on the needs of the patient and their current state of health.
– In order to be admitted to the driving exam, candidates first have to pass a verbal theory exam. However, a candidate who already possesses a certificate for the theoretical examination may advance to the practical examination directly.

Motivation The majority of structured-workflow systems prescribe a single start task as the entry point to a process definition. Despite differences in the context in which distinct process instances have to operate, the process definition created in such workflow systems enforces that the same start task has to be executed by each of the process instances. In some situations, it is necessary to skip the beginning of the process and commence execution from another task contained in the process definition. The Alternative Entry-Points pattern applies to situations when different execution paths, each having a distinct start task, are foreseen during the design of a process definition.

Overview Figure 8 illustrates two process notations: the top one corresponds to a rigid process with a single entry point, and the bottom one corresponds to a process with several alternative entry points, any of which can be selected to initiate the process from the related start task.

Context Figure 9 illustrates the process engine expressed using the CPN formalism extended to support the Alternative Entry-Points pattern.

To allow for process initiation from different start tasks, multiple start tasks have to be included in the process definition and are stored in the Process definition place. For this, the ProcModel type is modified to ProcModel’ as
Fig. 8. Alternative Entry-Points pattern

A process instance is created for a process model whose identifier mid matches with the id of existing process definitions, providing that the id of the start task (tid) supplied by a user is in the list of the alternative start tasks defined in the process-definition m.

Fig. 9. Engine enhanced with the Alternative Entry-Points pattern

shown below (note that this is different from the original process definition which contains only a single start task).

colset TaskIDs = list TaskID;
colset ProcModel’ = product ModelID * TaskIDs * TaskID * Tasks * Arcs;

In order to create a process instance, a knowledgable user supplies information in the form of (mid,pid,tid), where mid identifies the process definition based on which a process instance needs to be created, pid identifies the process instance to be created, and tid identifies the desired start task from which the given process needs to be initiated. For this, the ProcInst type is modified to ProcInst’ in order to include the identifier of the start task:

colset ProcInst’ = product ModelID * piID * TaskID;

The information provided by the user is used in the guard for the Create process instance transition, which, based on the input provided, identifies the process definition with the corresponding identifier and creates a process instance with the indicated start task tid. The isStartTask(tid,m) function is used to check whether the task identifier tid provided by the user corresponds to any of the start tasks defined for the indicated process definition. If the guard conditions have been satisfied, the createinst’(m,pid,tid) function creates a process instance which can be executed whilst the completion conditions are not met. Note that the createinst’(m,pid,tid) function is different from the orig-
inal createinst(m,pid) function in terms of the parameters it requires, and in order to create a process instance, a start task must be provided by the user.

**Implementation** Imperative systems intended for modeling structured workflows typically do not support this pattern directly, since they require a single entry point to the process definition that must be executed for all populated process instances. This applies to YAWL, FLOWer and ADEPT1. To realize multiple entry points in such systems, one has to introduce a dummy choice construct that allows one of the tasks included in the associated with the choice branches to be selected.

In contrast, declarative systems such as Declare allow any task, independent of the completion status of other tasks in the process definition, to be selected as a start task. CIG modeling languages such as EON and GLIF also allow multiple tasks to be marked as start tasks and depending on the patient’s state an appropriate task can be selected. PROforma combines both imperative and declarative approaches and allows any unconstraint task to be selected as a start task. In Oracle BPEL PM different <receive> activities can initiate a process instance by setting the createInstance attribute to yes.

**Issues** If, during run-time execution, a required start task cannot be found because the corresponding execution path has not been foreseen at design-time, a user may need to choose the most suitable of the foreseen execution paths, and execute a set of tasks, which normally would not need to be executed, in order to reach the required point in the process. By “jumping” into the process semantical problems, e.g. deadlocks and missing data, may occur.

**Solutions** One could solve this issue by applying the Entrance Skip pattern (see p. 16) in order to deviate from the executing tasks prescribed by the selected execution path. Alternatively, the Undefined Entry pattern (see p. 19) could be applied in order to determine the beginning of the process at run-time. Finally, the Momentary Entry-Point Change pattern (see p. 35) or the Permanent Entry-Point Change pattern (see p. 25) could be applied to change the entry-point prescribed by the process definition.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows several alternative tasks to be nominated for process definition each of which may play the role of the start task when a process instance is initiated.

**Pattern** FP-2  ENTRANCE SKIP

**Description** At process initiation, there is the possibility of deviating from the execution path prescribed by the process definition by ignoring the nominated start task. The execution of a process instance may start from any subsequent task specified in the process definition. The pattern assumes that the act of skipping the beginning of the process is a deviation that has no effect on the process definition. This pattern characterizes a need for flexible process initiation that is anticipated at the moment of process instance creation, thus it corresponds to the flexibility by deviation type.
Examples
- Treatment of patients at a hospital starts with registration of a patient. Patients who have been registered or are already being treated may immediately proceed to the required health unit.
- A housing agency identifies tenants to whom available accommodation will be allocated based on the length of time that they have waited. The normal selection process is skipped if a new client has requested accommodation under urgent conditions.

Motivation  In the **Alternative Entry-Points** pattern (described on p. 14), flexibility in process initiation is achieved by specifying alternative entry-points for a process definition at design-time. In some situations, where no alternative entry-points have been nominated at design-time or where no suitable entry-point can be found at run-time, the desired start task can only be reached by executing all tasks preceding it. This necessitates the execution of a number of tasks, which normally would be omitted. The *Entrance Skip* pattern allows the execution of a given process instance to start with the commencement of a desired task, by skipping all tasks on the path leading to it.

Overview  Figure 10 illustrates the graphical notation for the *Entrance Skip* pattern. The top view illustrates an initiated process instance before applying the pattern. In this process instance, the thread of control which is visualized by the filled triangle indicates a task that is currently enabled (in this case, the start task A is enabled).

The bottom view, illustrates the process instance after the *Entrance Skip* pattern has been applied. The process definition associated with the created process instance remains the same, however instead of enabling the start task A, the thread of control is moved to a subsequent task C. Thus the tasks A and B are skipped in order to commence the process instance at task C.

![Fig. 10. Entrance Skip pattern](image)

Context  Figure 11 illustrates the process engine expressed using the CPN formalism and shows how the *Entrance Skip* pattern should be realized.

In order to allow for process initiation from a task different to the nominated start task, a user needs to supply the identifier of the desired start task tid. For this, the information provided by a user at process initiation (which is of type Procinst) is modified to Procinst’ type as follows:

```
colset Procinst’ = product ModelID * piID * TaskID;
```
When an identifier for the desired start task \( \text{tid} \) together with corresponding model identifier and an identifier for the process instance to be created are made available by a user at the \text{Begin} \ place, this data is used by the \text{Create process instance} \ transition to create a process instance with the nominated start task or by the \text{Entrance Skip} \ transition to create a process instance without a nominated start task. As described in the \text{Alternative Entry-Point} \ pattern, in this engine the \text{isStartTask}(\text{tid}, \text{m}) \) function is introduced to check whether the identifier \text{tid} \ provided by the user corresponds to a nominated start task in the model \text{m}. Furthermore, the \text{createinst}^1(\text{m}, \text{pid}, \text{tid}) \) function is used to create a process instance, where the start task identifier is \text{tid}.

![Diagram](image-url)

**Fig. 11.** Engine enhanced with the \text{Entrance Skip} \ pattern

**Implementation** Of the wide range of contemporary offerings investigated, only FLOWer and Declare provide support for the \text{Entrance Skip} \ pattern. In Declare, a set of tasks including the start task can be skipped by ignoring optional constraints. Although in FLOWer it is not possible to advance to a desired start task directly, the same effect can be achieved by executing a series of skip operations. In ADEPT1 and YAWL it is not possible to skip start tasks at process initiation, neither is this possible in Oracle BPEL PM. However, in Declare if none of the tasks has been marked as a start task, the process execution may start from any of the tasks defined.

The decision-support systems used for enactment of clinical guidelines such as PROforma provide recommendations in the form of decisions where zero, one or more options can be selected. These can be used to encode decisions related to the execution of a certain task in the process, thus they are also applicable for describing optional start tasks. In Asbru, EON and GLIF no configuration with respect to a desired start task can be done at process initiation.

**Issues** When the execution of a specific task needs to be skipped, and a subsequent task needs to be enabled, problems related to data dependencies between these tasks may arise. Often in order to be enabled, a task requires input data which is provided by the previous task. When a task is skipped, the required data may not be available to the subsequent task.
**Solutions** In order to avoid the problem of required data elements from preceding tasks not being available at task enablement, it is possible to define a default value for data elements and use them instead of the output from the skipped task.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that provides an explicit operation to skip one or more tasks, including the start task, at process initiation.

**Pattern** FP-3 UNDEFINED ENTRY

**Description** A process definition contains a placeholder, associated with the beginning of the process, that is intentionally left underspecified. During process initiation there is the possibility to complete the specification of this placeholder with an appropriate start task. This pattern characterizes a need for flexible process initiation recognized at design-time, but whose actual realization is performed at run-time, thus it corresponds to the flexibility by underspecification type.

**Examples**
- The accounting program developed to calculate financial metrics which track the performance of a business can be supplied to and used in any organization. Depending on the context in which the program is to be used, the initial information which needs to be provided when initiating the accounting program may vary, thus allowing it to be tailored to a specific customer.

- An Internet banking transaction starts with the selection of a specific operation by a client. This field is initially empty and has no default option associated since multiple options may play this role.

**Motivation** Often in situations where the need for flexible process initiation is recognized at design-time, a set of possible start tasks is defined, each corresponding to an alternative execution path. The Alternative Entry-Points pattern (described on p. 14) allows different entry-points to a process to be specified by explicitly defining distinct start tasks at design-time. In some situations, it is not always clear at design-time which of the tasks may need to play the role of the start task or it may be impractical to specify all possible start tasks explicitly. The Undefined Entry-Point pattern allows the specification of the beginning of the process, including the start task, to be postponed until process initiation when more information related to the operational context becomes available.

**Overview** Figure 12 illustrates the graphical notation for the Undefined Entry pattern. The top view illustrates a process instance that has been populated from the process definition where the beginning of the process is left underspecified and marked as a placeholder. The static part of the process which consists of tasks C and D has been predefined at design time, and thus cannot be executed until the beginning of the process is defined.

The bottom view visualizes the completion of the placeholder performed at runtime after applying the pattern. After process instance creation, the placeholder becomes enabled as illustrated by the execution thread preceding it. Once
the placeholder is enabled, it needs to be completed. Once the content of the placeholder has been defined, the thread of control is moved to the specified start task. In this case, task B is chosen as a start task, however a process fragment consisting of tasks A and B could have been used instead. Note that the mechanism for completing the placeholder associated with the beginning of the process is the same as that for any other placeholder in the process definition.

**Context**  In this section, we illustrate how flexibility in process initiation as promoted by the *Undefined Entry* pattern can be incorporated in the process engine. In order for the definition of a start task to be completed at run-time, the start task needs to be left underspecified in the design-time process definition and the type of this start task has to be set to *placeholder*.

**Fig. 12.** Undefined Entry pattern

**Fig. 13.** Placeholder engine: top view

After a process has been initiated by the basic process engine presented in Figure 5(c) no tasks may be executed until the definition of the underspecified start task is completed by the placeholder engine whose main view is presented in Figure 13. In the placeholder engine a placeholder for a given process instance
can be completed either by creating a new task (e.g. the Create substitution transition), selecting a process fragment from a set of process fragments defined at design-time (e.g. the Select substitution transition), or by composing a new process fragment from existing and/or new tasks (e.g. the Compose substitution transition).

In order for a new task to be created the substitution transition Create, whose functionality is presented in Figure 14(a), needs to be executed. The new task created by the createnewtask() function is added to the repository of process fragments by the addtaskstorepos(tid,lrep) function. The content of the selected placeholder selp is replaced by the newly created task. In particular, the update_create() function modifies both the process definition, and also updates the state of the process instance so that the created task can be executed.

![Diagram of Placeholder Engine](image)

**Fig. 14.** Placeholder engine

When the placeholder needs to be completed with one of the previously defined process fragments, the Select substitution transition needs to be executed. The behavior corresponding to this transition is presented in Figure 14(b). A process element tp can be selected from the repository of process fragments lrep available in the Repository place under an assumption that this repository is non-empty.
A process fragment, that has been randomly selected from the repository, is used to substitute the given placeholder in the process definition and in the process instance state. The update of the process definition and process instance state is performed via the `update_select()` function.

Finally, the content of a placeholder can be defined by composing a process fragment from new tasks and process fragments available in the `Repository` place. In order to compose a process fragment the `Compose` transition, whose behavior is shown in Figure 14(c), needs to be executed. A process fragment that will be used for completion of the selected placeholder is formed by the `Select` and `Create` transitions, and stored in the `process fragment` place. The functionality of these transitions corresponds to the ones described earlier for the task creation and process fragment selection. The created and selected elements are coupled to each other in the sequential order. The content of the placeholder is then replaced by the process fragment obtained and the process state is updated by the `update_compose()` function in such a way that the first task in this fragment becomes enabled.

**Implementation** From the set of workflow offerings analyzed, only YAWL supports this pattern through its worklets extension [10]. A worklet is a reusable process fragment consisting of one or more tasks that can be included in the process at run-time. None of the other systems investigated, i.e. ADEPT1, Declare, FLOWer or Oracle BPEL PM provide any means of realizing of the *Under-defined Entry* pattern. Among CIG modeling languages, only PROforma provides a similar concept, where at design-time a keystone construct representing a generic task can be used. However, before deploying the process definition the keystone has to be specified explicitly. Thus the moment of specification is postponed not until run-time but until the latest possible moment at design-time.

**Issues** The repository of fragments used to determine the start task for a given process instance at run-time should not be empty, i.e. it must contain at least one process fragment if the placeholder has to be completed by selecting a fragment from the repository rather than by creating a new process fragment. An empty repository may potentially lead to process instances blocking.

**Solutions** When the definition of a placeholder at runtime cannot be completed because the repository of process fragments is empty, it should be possible to select a process fragment with empty content. The execution of a placeholder with empty content can be treated as if this placeholder would not exist or its completion would be skipped.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows an incomplete process definition to be enacted. An under-specified process-entry fragment must be explicitly marked as a placeholder at design-time, and it should be possible to complete its definition at run-time either before or when the placeholder is enabled.

**Pattern** FP-4  MOMENTARY ENTRY-POINT CHANGE
**Description** At process initiation, e.g. before the start task prescribed by the process definition has commenced, there is the possibility to change an entry-point for the process by modifying the process definition associated with the particular process instance being created. This pattern characterizes the change to the given process instance and has no effect on other (current or future) process instances, thus it corresponds to the *flexibility by momentary change* type.

**Examples**

- A patient transported from one hospital to another is accompanied by his/her X-ray photos. For this patient the X-ray examination should not performed and he/she should immediately proceed to surgery.

- A typical boarding procedure for clients at an airport is modified for premium clients according to their personal requirements (e.g. the issuing of the boarding pass and visa arrangement can be omitted to allow the client to proceed to the gate immediately).

**Motivation** In situations where at process initiation no suitable start task can be found in the set of foreseen start tasks and it is not possible to deviate from the prescribed execution path by ignoring the nominated start task, the prescribed start task and any subsequent tasks have to be executed until the desired task is reached. The *Momentary Entry-Point Change* pattern allows the process definition associated with the given process instance to be modified in such a way that the execution of a process instance may start with any task in the prescribed execution path.

**Overview** Figure 15 illustrates the graphical notation for the *Momentary Entry-Point Change* pattern. The top view depicts two distinct process instances populated from the same process definition. Each instance has a process definition associated with it that is used to determine the next task to be executed. The thread of control in each of the process definitions is illustrated by the shaded triangle and circle respectively. The pattern is to be applied to a process instance whose execution thread is depicted by the shaded triangle.

![Diagram](image-url)

**Fig. 15.** Momentary Entry-Point Change pattern
The momentary entry-point change that is performed for one of the process instances, does not affect the process definition associated with other process instances. The bottom view illustrates that after task A and task B have been removed from one process instance, the thread of control associated with this instance is moved to the subsequent task, whilst other process instances are unaffected.

**Context** Figure 16 illustrates how the process engine expressed using the CPN formalism needs to be extended in order for flexibility for process initiation as demonstrated by the Momentary Entry-Point Change.

In order to incorporate the functionality for supporting flexibility by momentary change, we will extend the basic process engine as shown in the Figure 16. For the context purposes, we will modify the name of transition performing the modification of the particular process instance, however the names of functions used will remain unchanged. Note that the structure of the engine allowing for momentary changes is uniform for all patterns enhancing the flexibility by momentary change. The main difference between the realization of these patterns is the functionality associated with the `change_possible()` and `update_pi()` functions. In order for the process definition associated with the given process instance to be changed, a set of conditions incorporated into the `change_possible()` function have to be satisfied. The `update_pi()` function makes the actual modifications in the process definition, and whenever necessary also updates the state of the process instance.

In the Momentary Entry-Point Change pattern, the process definition associated with the process instance stored in the Running instance place and which has not commenced yet can be modified in order to start execution from the task other than the nominated start task. The `change_possible()` function checks whether there are enabled tasks that can be executed and whether the task enabled is the nominated start task. This function would evaluate to False if the execution of the process instance has commenced and the start task has been executed.

![Diagram](image)

**Fig. 16.** Engine enhanced with the Momentary Entry-Point Change pattern
When the guard associated with the Momentary entry change transition is satisfied, it can be executed. The update_pi() function updates the process instance in the following way. First, an arbitrary task from the set of tasks associated with the given process definition m is selected. This task will be set as the new start task in the original process definition. All tasks preceding the newly selected start task will not be executed any more, therefore these tasks and all associated with them branches are removed from the process model. Then the process instance state is updated, e.g. the newly selected start task becomes enabled.

Note that this engine is realized under an assumption that momentary changes are performed on-the-fly, i.e. after the process instance has been created.

**Implementation** ADEPT1 and Declare allow the Momentary Entry-Point Change pattern to be applied after a process instance have been created and before the start task prescribed by the process definition has been enacted. In ADEPT1, a desired task cannot be initiated directly and requires a number of delete operations to be performed. In Declare, a start task can be deleted and execution may start from another task. None FLOWer, Oracle BPEL PM, YAWL or clinical guideline languages allow for changes to be performed at run-time at the process instance level.

**Issues** When the beginning of a process is being removed from the process definition in order for process execution to commence from another task, the problem may arise that data elements are missing that were previously provided by the tasks that have been removed.

**Solutions** In order to allow the beginning of a process to be inconsequentially removed, each of the tasks in a process definition that may play the role of a start task must be associated with a default value which could be used for enabling the execution of the given task if no other input has been or can be provided.

**Evaluation Criteria** Full support for this pattern is demonstrated by an offering that allows any task (including the start task) to be deleted from the process model associated with a given process instance at run-time. It should be possible to mark a task other than the dedicated start task as the entry point to the process definition.

**Pattern** FP-5 PERMANENT ENTRY-POINT CHANGE

**Description** At run-time execution, the possibility exists to permanently change the entry point for a process by modifying the process definition. This pattern characterizes the change that affects all future process instances directly and requires existing process instances to migrate from the old to the new process definition, thus it corresponds to the flexibility by permanent change type.

**Examples**
- Visitors traveling to Eastern Europe are expected to acquire a visa before making ticket reservations. For countries that joined the European Union, the visa requirement is abolished. Visitors already possessing a visa continue traveling as usual. For new visitors however, the travel organization may start directly with ticket reservation.
The master program for international students used to start with a half-year homologation period. Due to financial reasons, the reorganization of a school is performed requiring canceling the homologation program for new generations of students. This change aims at increasing the level of accepted applicants, and does not affect the students who already follow the homologation program.

**Motivation** When the start task identifying the entry point to a process definition needs to be removed for all future process execution, there may arise the need to permanently modify the process definition by removing the nominated start task. Such an adjustment may be more efficient than applying the *Momentary Entry-Point Change* pattern (described on p. 35) to each newly created process instance. The *Permanent Entry-Point Change* pattern allows the process definition to be modified at a type level and the execution of process instances to start from a task subsequent to the nominated start task.

**Overview** Figure 17 illustrates the graphical notation for the *Permanent Entry-Point Change* pattern. The top view depicts the execution state of several process instances (based on the same process definition) before applying the pattern.

![Fig. 17. Permanent Entry-Point Change pattern](image)

The bottom view illustrates that for all process instances the process definition has been modified by removing task \( A \) and task \( B \). The thread of control associated with the process instances where these tasks have not been executed yet is moved to the subsequent task \( C \). The process instance visualized by the square that has already passed this point is not affected by the change.

**Context** Figure 18 illustrates how the process engine expressed using the CPN formalism needs to be extended in order for flexibility for process initiation as offered by the *Permanent Entry-Point Change* pattern to be achieved.

In order to incorporate the functionality for supporting flexibility by permanent change, we will extend the basic process engine as shown in the Figure 18. For the context purposes, we will modify the name of transition performing the modification of the particular process instance, however the names of functions used will remain unchanged. Note that the structure of the engine allowing for changes on the type level is uniform for all patterns enhancing the flexibility by permanent change. The main difference between the realization of these patterns is the functionality associated with the `change_possible()`, `modify_m()`,
transferposisble() and migrate() functions. In order for the process definition to be changed on the type level, a set of conditions incorporated into the change_possible() function have to be satisfied. The modify_m() function makes the actual modifications in the process definition. After the process definition has been modified, the migration of existing process instances may need to be performed. The transfer_possible function checks whether the migration can be performed, and if yes, then the migrate() function updates the process definition and process state associated with the given process instance.

In the Permanent Entry-Point Change pattern the possibility to modify the start task associated with the process definition at run-time. The Permanent entry change transition associated with this functionality can be executed only if there exist a task that can be selected as a new start task. This realization assumes that the process definition must contain at least two tasks, including the start task and the end task. The modify_m() function changes the original start task to a new start task that has been selected on a random basis from the set of available tasks. The original start task, all tasks preceding the newly selected start task as well as the related branches are removed from the process definition.

The migration of existing process instances in the Running instance place to the new process definition can be performed only if there are enabled tasks in the process instance. For the process instances that have been initiated already, this change is inconsequential, however for process instances that have not commenced yet the change performed may impact the process execution. The actual migration strategy defining the mapping of existing process instances to the new process definition can be encoded into the migrate() function. Process instances whose original process definition m differs from the new process definition newm may either proceed with execution, be aborted, restarted or transferred to the new definition.

![Fig. 18. Engine enhanced with the Permanent Entry-Point Change pattern](image-url)
Implementation  ADEPT1 and Declare allow the Permanent Entry-Point Change pattern to be applied at run-time in order to adjust a process description. Such a change implies the need for process instance migration, which applies both to existing process instances that have not yet commenced and to all future process instances. Existing process instances ignore the change introduced and continue to execute. In ADEPT1 it is not possible to change the entry point for a process in one step, but tasks must be removed one-by-one until the desired start task is reached. Definition of a new process via worklets means that this worklet is available to all existing and future process instances via a shared repository. FLOWer, Oracle BPEL PM, and the CIG modeling languages offer no support for permanent changes to process during execution.

Issues  When existing process instances need to be migrated to a new process definition, a problem known as the ‘dynamic change bug’ may occur [65]. The dynamic change bug is characterized by errors that may occur when transferring an old process definition to a new one (e.g. tasks may be duplicated, omitted, or even deadlock situations may arise).

Solutions  In order to address the dynamic change bug problem an approach proposed in [65, 3, 5, 4, 25, 54] can be used. For a given process definition the change region, i.e. a part of the process definition that is affected by the change, is calculated. Each process instance is analyzed in regard to the identified change region. If the thread of control in a process instance is outside of this region, the transfer to the new process definition can be safely performed, otherwise the transfer is postponed until the thread of control in the process instance leaves this region.

Evaluation Criteria  Full support for this pattern is demonstrated by any offering that allows a task (including the start task) to be deleted from the process definition at run-time, and provides migration facilities for mapping existing process instances from the old to the new process definition. It should be possible to mark a task other than the dedicated start task as the entry point to the process definition.

3.3 Flexible termination

This group of patterns aims to describe flexibility options when terminating a business process on a premature basis. Figure 19 illustrates the scope of patterns presented in this subsection and their relationship to different types of flexibility. We will traverse cells in the emphasized row from left to right and analyze the mapping of the concept of flexible process termination for each of the flexibility types.

Usually a process definition has an explicit end task, which needs to be executed for a process instance to complete. Such single-exit processes force the execution of all process instances to follow all steps prescribed by the process definition until the end task is reached. In some situations, the execution of the end task may no longer be needed at run-time or the execution of the process may need to complete before the specified end task has been reached. Depending
on the moment at which the need for an alternative process termination point is recognized and the manner in which it is achieved, we distinguish the following five patterns: *Alternative Exit Points*, *Termination Skip*, *Undefined Exit*, *Momentary Exit Change* and *Permanent Exit Change*.

Although these patterns all describe means of achieving premature process termination, they differ in terms of the moment at which the need for such flexibility is anticipated. Of these five patterns, the *Alternative Exit Points* pattern is relevant at design-time. In this pattern the need for premature process termination is recognized during process design and all possible options for its realization are incorporated in the process definition. The other four patterns are characterized by the fact that the actual decision in regard to the selection of the process termination point is taken at run-time. In the *Termination Skip* pattern, the need to deviate from the nominated end-task is identified after process initiation. In the *Undefined Exit* pattern the need for flexible process termination is recognized at design-time, however the actual realization of this decision is postponed till the latest possible moment at run-time execution. In the *Momentary Exit Change* and *Permanent Exit Change* pattern the need for flexible process termination is anticipated and realized at run-time by modifying the process definition on the instance and type levels respectively.

**Pattern FP-6 ALTERNATIVE EXIT-POINTS**

**Description** A process definition specified contains more than one end task, execution of any of which at run-time results in the termination of a given process instance. The availability of alternative exit points in a process definition gives flexibility in terminating the process execution on the premature basis. This pattern characterizes a need for flexible process termination that is anticipated at design-time, thus it corresponds to the *flexibility by design* type.

**Examples**
- The 6-year education program can be terminated after four, five or six years of study corresponding to attainment of the bachelor, engineer and master levels respectively.
The treatment of the cancer consists of several steps, where a patient has to undergo a set of chemotherapies, followed by ablation. A patient may undergo as many chemotherapies as he/she chooses and may stop the treatment at any point if the side-effects become unbearable. For patients who have recovered from the disease after several chemotherapies the treatment process stops immediately.

**Motivation**  Typical process definitions contain a single end task. Once initiated such processes may complete only after the nominated end task has been completed. Under conditions which are different from the normal flow of tasks, it might be necessary to prematurely terminate the process execution. The Alternative Exit-Points pattern allows several end-tasks to be incorporated in the process definition at design-time, thus providing alternative exit points for process instances. The completion of an end task from the specified set of nominated end-tasks results in the process instance terminating.

**Overview**  Figure 20 illustrates the graphical notation for the Alternative Exit-Points pattern. The top view corresponds to a rigid process with a single exit point, before applying the pattern. The bottom view illustrates the process definition after applying the pattern. Several alternative exit points are now defined, any of which can be selected to terminate the process from the related end task. The process definition in the bottom view contains a static part (i.e. task A), which is fixed and has to be executed for any process instance, while a dynamic part consisting of tasks B, C, D defines alternative ways of terminating the process instance.

**Context**  Figure 21 illustrates the process engine expressed using the CPN formalism and how the Alternative Exit-Points pattern can be realized in order to provide for flexibility in process termination. To incorporate alternative end tasks into the process definition, the original ProcModel type needs to be changed to the ProcModel' type in order to include a set of end-tasks rather than a single end-task, as shown below:

```
colset ProcModel' = product ModelID * TaskID * TaskIDs * Tasks * Arcs;
```

In the basic process engine, a process instance stored at the Running instance place can be terminated when no enabled tasks remain. When one of the alternative end tasks has been selected for execution, no tasks can be executed after-
wards, and the execution of the process instance can be completed by executing the Complete process instance transition.

Fig. 21. Engine enhanced with the Alternative Exit-Points pattern

Implementation None of the systems analyzed except for Declare allow multiple tasks in a model to be marked as end tasks. YAWL explicitly forces all end tasks to be synchronized into a single exit point. Oracle BPEL PM, ADEPT1 and FLOWer operate with structured models, which are characterized by a single start task and a single end task, however in Oracle BPEL PM it is also possible to introduce <terminate> activity in order to terminate process execution earlier. In Declare different tasks can be associated with a process termination condition. After executing a task whose process termination condition is satisfied, no further tasks in the process definition can be executed. Amongst the considered CIG modeling languages, only PROforma allows process termination conditions to be associated with different tasks in a process. After executing a task whose process termination condition has been satisfied, the process terminates.

Issues When a decision to terminate a process instance by selecting one of the nominated end tasks is taken, there may exist tasks that are currently executing or which are enabled and may execute later. Early termination of tasks that have been commenced but not yet completed may result in loss of data.

Solutions Depending on whether the data produced by tasks executed at the moment of process instance termination, the decision may be taken either to allow these tasks to complete or to abort their execution and lose the data.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that allows several tasks to play the role of an end task, the execution of any of which causes a corresponding process instance to terminate.

Pattern FP-7 TERMINATION SKIP

Description During execution, i.e. after a process instance has been created, there is the possibility of deviating from the execution path prescribed by the
process definition by ignoring all subsequent tasks. The act of skipping the execution of the currently enabled and in future to-be-enabled tasks results in the premature termination of a process instance. This has no effect on the process definition, thus this pattern corresponds to the flexibility by deviation type.

**Examples**
- In the middle of the investigation, the patient’s complaint disappeared, therefore the patient decided to stop the process. All prescribed tests have been skipped, without affecting the medical guideline on the basis of which they were prescribed.
- The recruitment process for a candidate who did not pass the capacity test is terminated without affecting the established procedure for solicitation.

**Motivation** A typical process definition contains a set of tasks that have to be executed until the end of the process is reached. In some situations, the process needs to be terminated before the end task prescribed by the process definition has been reached. This situation may even occur if at design-time the possibility for premature process termination has been foreseen and multiple alternative exit tasks have been defined (as described in the Alternative Exit Points pattern on p. 29), however the desired end-task has not yet been reached. The Termination Skip pattern allows the execution of a given process instance to end at a particular task by skipping all currently enabled and in future to be executed tasks.

**Overview** Figure 22 illustrates the graphical notation for the Termination Skip pattern. The top view shows the process instance before applying the pattern. The execution thread in this process instances indicates that task C is enabled. The bottom view shows that after applying the pattern the thread of control has been moved beyond the task D, which corresponds to the process termination. Note that the process definition remains unchanged.

![Fig. 22. Termination Skip pattern](image)

**Context** Figure 23 illustrates the process engine expressed using the CPN formalism and shows how the Termination Skip pattern needs to be realized in order to support flexible process termination.

In order to realize the functionality associated with the flexibility by deviation, the model of the process engine is extended with a transition performing deviation operation and the Log place which illustrates which tasks have been executed (thus provides the means for tracking the deviations from prescribed execution order). When describing the behavior of patterns facilitating flexibility
by deviation, we will use a set of functions with uniform names, and will adjust the content of these functions in order to incorporate the desired behavior. As such, the \texttt{deviation\_possible()} function incorporates a set of conditions that have to be satisfied in order to allow for deviation operation. The \texttt{update\_dev()} function has only updates the process instance state but has no impact on the process definition associated with the given process instance.

Due to the introduction of the \texttt{Log} place the structure of the functions associated with enabling and execution of the \texttt{Execute task} transition have been slightly modified. A new function \texttt{exec\_task()} has been introduced, which nondeterministically selects a task from the set of currently enabled tasks for the execution. The \texttt{update\_exec()} function updates the process instance state after the selected enabled task has been executed.

Unlike other extensions of the process engine to support momentary and permanent changes, the extension of the process model presented in Figure 23 represents a mix of the engine functionality and the environmental/user choice to deviate from the execution. Depending on the engine realization, these extensions can be incorporated as explicit functionality options or supported implicitly.

In the \texttt{Termination Skip} pattern, for running process instances, whose process definitions and current states are stored in the \texttt{Running instance} place, there is the possibility either to execute a currently enabled task or to skip all current and future tasks by executing the \texttt{Termination Skip} transition. The execution of this transition is equivalent to completion of the process instance on a premature basis. The guard for the \texttt{Termination Skip} transition specifies that it can only be enabled if there are tasks enabled in the current state. By executing this transition, it is possible to jump from any execution state to process instance termination state in which no tasks can be executed any more. The termination state is calculated using the \texttt{update\_dev()} function.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig23.png}
\caption{Engine enhanced with the Termination Skip pattern}
\end{figure}

\section*{Fig. 23. Engine enhanced with the Termination Skip pattern}
Implementation  Support for the Termination Skip pattern is provided by Oracle BPEL PM, FLOWer, and Declare. In FLOWer, a process instance can be terminated by executing a skip operation at the level of the root plan. In Oracle BPEL PM, an event handler can be defined at the outer scope of the process to perform the Terminate activity once a particular event occurs. In Declare, the process can be terminated after all mandatory constraints have been satisfied. The CIG modeling languages offer no support for this pattern.

Issues  None identified.

Solutions  N/A.

Evaluation Criteria  Full support for this pattern is demonstrated by any offering that provides an explicit operation to skip several tasks at once, resulting in the thread of control being moved to the end of the process.

Pattern FP-8  UNDEFINED EXIT

Description  A process definition contains a placeholder, associated with the end of the process, that is intentionally left underspecified. During process initiation there is the possibility to complete the specification of this placeholder with an appropriate end task. This pattern characterizes a need for flexible process termination recognized at design-time, but whose actual realization occurs at run-time, thus it corresponds to the flexibility by underspecification type.

Examples

– Students starting a high-school education are obliged to follow a basic set of courses. They may select additional elective subjects when their individual academic goals become clear.

– For patients arriving at the emergency center only the admittance procedure is defined: the patient’s insurance is checked and a questionnaire is filled in. The subsequent investigative or treatment steps depend on the state of the patient and are defined next.

Motivation  In many situations, it is not practical to explicitly specify how the process execution must end. This may either be because the actual end of the process is unknown or because it may vary for individual process instances. The Undefined Exit pattern allows the specification of the end of a process to be postponed until run-time, when more information related to the operational context becomes available.

Overview  Figure 24 illustrates the graphical notation for the Undefined Exit pattern. The top view shows a process instance where the end of the process represented by a placeholder is enabled. In order to complete the process, the placeholder needs to be defined. The bottom view shows that after applying the pattern, task C defining the content of the placeholder becomes enabled. Note that tasks A and B represent the static part of the process, which have to be executed for each process instance. The content of the placeholder may however vary for different process instances, e.g. instead of task C a process fragment consisting of tasks C and D could be used.
**Context** The semantics of completing the placeholder representing the end of the process is the same as for any other placeholder. Once the placeholder is enabled, its definition needs to be completed by the placeholder engine depicted on Figure 5(b) in a similar manner to how it has been described in the *Undefined Entry* pattern (see p. 19). In order to allow the process end to be defined at run-time, the end-task associated with the process definition has to be underspecified at design-time (its task type must be set to the *placeholder* value).

**Implementation** Of the set of workflow offerings analyzed only YAWL supports this pattern by means of worklets [10]. None of other systems investigated, i.e. ADEPT1, Declare, FLOWer or Oracle BPEL PM provide means for realization of the *Undefined Exit* pattern. Among CIG modeling languages, only PROforma provides a similar concept, where at design-time a keystone construct representing a generic task can be used. However, before deploying the process definition the keystone has to be specified explicitly. Thus the moment of specification is postponed not until run-time but until the latest moment of design-time.

**Issues** Same issues as identified for the *Undefined Entry* pattern (see p. 19) apply here also.

**Solutions** See solutions identified for the *Undefined Entry* pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows incomplete process definitions to be enacted. An underspecified process-end fragment must be explicitly marked by a placeholder at design-time, and it should be possible to complete its definition at run-time once the placeholder is enabled.

---

**Pattern** FP-9  **MOMENTARY EXIT-POINT CHANGE**

**Description** During execution, i.e. after process initiation, there is the possibility to change the exit-point for the process by *temporarily* modifying the process definition associated with the given process instance. In this pattern the change applies only to a specific process instance and has no effect on other (created and future) process instances, thus it corresponds to the *flexibility by momentary change* type.

**Examples**
- For a patient who developed signs of high blood-pressure an appointment made for an operation has to be canceled and the medication previously prescribed by the doctor has to be terminated immediately.

- A PhD student who has been ill for more than two years has been fired. As a consequence of this all subscriptions for new courses has to be stopped and involvement of all ongoing courses for the given student is terminated.

**Motivation**  A typical process definition contains a single exit task, which needs to be executed in order for the process to terminate. Although it is possible to include several alternative exit-points in a process definition at design-time (as the *Alternative Exit-Points* pattern on p. 29 describes), there exists the possibility that the desired exit-point will not be found in the set of exit tasks defined. When no suitable exit task for a terminating process instance can be found at run-time, and it is not possible to deviate from the execution path prescribed by applying the *Termination Skip* pattern (see p. 31), it may be necessary to temporarily modify the process definition in order to allow the given process instance to terminate prematurely. The *Momentary Exit-Point Change* pattern allows the execution of a process instance to be instantly completed at any point during process execution by modifying the nominated end task.

**Overview**  Figure 15 illustrates the graphical notation for the *Momentary Entry-Point Change* pattern. The top view depicts two distinct process instances populated from the same process definition before applying the pattern. The pattern is applied to the process instance whose execution thread is depicted by the shaded triangle. The bottom view shows that for the given process instance tasks C and D have been removed from the process definition, and the thread of control has been moved to the end of the process. This change does not affect the other process instance.

![Fig. 25. Momentary Exit-Point Change pattern](image)

**Context**  Figure 26 expresses the process engine using the CPN formalism and shows how the *Momentary Exit-Point Change* pattern is realized.

In order to incorporate flexibility facilitated by the *Momentary Exit-Point Change* pattern, the basic engine is extended as has been described earlier in the
Any task that has not yet been executed and which precedes the nominated exit task can be marked as the new end task.

Momentary Entry-Point Change pattern on p. 35. The Momentary exit change transition can be executed for an initiated process instance that is stored in the Running instance place and which has not yet reached the end task yet.

Any currently enabled task that is different from the nominated end task can be selected as a new end-task. By executing the Momentary exit change transition, the original process definition associated with the given process instance is modified. The update_pi() function sets the currently enabled task to be a new end-task, ensuring that all follow-up tasks and arcs are removed from the process definition associated with the given process instance. The process instance state does not change in this case and therefore is not modified.

Implementation ADEPT1 and Declare allow the Momentary Exit-Point Change pattern to be applied after process initiation. In ADEPT1 all tasks subsequent to a particular point in the process definition have to be removed one-by-one. The removal in ADEPT1 corresponds to disabling of tasks, i.e. after deleting a task a user still sees where the deleted task resided but cannot execute it. In Declare either all unnecessary tasks have to be removed or a condition associated with the desired end task needs to be modified such that it becomes the process instance termination condition. In YAWL when a desired point in the process has been reached, an exlet can be called which will result in the termination of the given process instance. In FLOWer, Oracle BPEL PM and CIG modeling languages no changes to the process definition can be temporarily performed at run-time.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that allows any group of tasks (including the end task) to be deleted from the process definition associated with a given process instance at run-time in a single step. It should be possible to mark any task other than the nominated end task as an exit point from the process definition.
**Description** At run-time, there is the possibility to permanently change the exit point for a process by modifying the process definition. In this pattern, the change performed directly affects all future process instances, while existing process instances may require migration from the old to the new process definition, thus it corresponds to the flexibility by permanent change type.

**Examples**
- The procedure of obtaining a visa has been modified for all applicants. Since the introduction of electronic applications, all documents are handled by an external organization, and no interviews at the embassy are required. All appointments made for interviews are canceled and no new appointments will be made from this point.
- In order to save costs, a company producing products on demand decided to eliminate the number of final tests of the product functionality. This change affects all ongoing and future product production lines.

**Motivation** Due to changes in the operational environment, there may arise the need to modify the end of a process when completing the execution of current and future process instances earlier than originally defined. Although the Momentary Exit-Point Change pattern could be used for modifying an exit-point for a specific process instance, it will require the changes to be made for all process instances separately. The Permanent Exit-Point Change pattern offers a more efficient way of modifying the exit-point associated with the process definition by changing it at the type level.

**Overview** Figure 27 illustrates the graphical notation for the Permanent Exit-Point Change pattern. The top view depicts the execution state of several process instances populated based on the same process definition before applying the pattern. The permanent entry change performed on the type level (eliminating tasks C and D) affects process definition associated with all existing process instances, as shown on the bottom view.

![Fig. 27. Permanent Exit-Point Change pattern](image)

**Context** Figure 28 illustrates an engine extended with the Permanent Exit Change pattern, which enhances flexibility in process termination, using the CPN formalism. The process engine has been extended using the structure described earlier in the Permanent Entry-Point Change pattern on p.3.2.

The content of the change_possible() function defines that the Permanent exit change transition is enabled only when the process models contains more
than two tasks (including the start and end tasks). The \texttt{modify.m()} function picks an arbitrary task from the set of the tasks associated with the process definition being modified, sets it as the new end task, and removes all succeeding tasks and related branches.

The process definition \texttt{m} associated with a process instance stored in the \texttt{Running instance} place can be replaced by the \texttt{migrate()} function with the new process definition \texttt{newm} if the change performed does not course enabling problems afterwards. If a task that has not been yet executed has been selected as the new start task, then the migration can be performed directly. However, if this task has been executed already, the change either will be inconsequential or will require the process instance to be restarted. This highly depends on the migration strategy chosen for the realization of the \texttt{migrate()} function.

\begin{center}
\hspace{-1cm}
\includegraphics[width=\textwidth]{engine.png}
\end{center}

\textbf{Fig. 28.} Engine enhanced with the Permanent Exit Change pattern

\textbf{Implementation} ADEPT1 and Declare are the only systems of those analyzed which allow the Permanent Exit-Point Change pattern to be applied at run-time in order to modify the end task in the process description. This change implies the need for process instance migration. Process instances which have not yet commenced or which have not reached the new exit task are migrated directly to the new process definition, while the process instances that have passed this execution point are terminated. In YAWL, an exlet can be called that allows all existing process instances to be terminated. In FLOWer, Oracle BPEL PM and the CIG modeling languages there is no possibility of changing the end task in a process definition on the permanent basis at run-time.

\textbf{Issues} Identical to the issues identified for the Permanent Entry-Point Change pattern (see p. 25).

\textbf{Solutions} See solutions identified for the Permanent Entry-Point Change pattern.
**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows several tasks in a process (including the nominated end task) to be deleted from the process definition in a single step, and which provides support for process instance migration.

### 3.4 Flexible selection

This group of patterns aims to provide flexibility when selecting an execution path appropriate to the operational circumstances. Figure 29 illustrates the scope of patterns presented in this subsection and their relation to different types of flexibility.

In sequential processes, all tasks have to be executed in a predefined order. Such processes are very rigid and offer no possibility for deviating from the default execution path and selecting an appropriate alternative to it. In some situations, different execution paths may need to be incorporated in the process definition in order to allow a knowledgeable user to select a suitable execution alternative during process execution. The flexible selection patterns address different ways of realizing choices between several alternative tasks, each of which corresponds to an alternative execution path. The main purpose of these patterns is to underpin the availability of multiple alternatives and the ability to make a choice between them, rather than to specify the exact semantics of such a choice. In order to specify the type of a choice explicitly one has to consider different variants of branching control-flow patterns described in [55].

Depending on the moment at which the need for realizing the choice between alternative tasks is recognized and the manner in which it is achieved, we distinguish the following five patterns: **Choice**, **Task Substitution**, **Late Selection**, **Momentary Choice Insertion**, and **Permanent Choice Insertion**. Of these five patterns, the **Choice** pattern corresponds to the design-time choice. In this pattern the need for several alternative execution paths is recognized and incorporated.
into the process definition at design-time. The Task Substitution pattern relates to situations when the need for alternative execution path is recognized at run-time, i.e. after the process instance creation, and is realized by deviating from the prescribed execution path rather than by modifying the corresponding process definition. In the Late Selection pattern the need for alternative execution paths is recognized at design-time, however the actual realization of the chosen execution alternative is postponed until the latest possible moment at run-time. The Momentary Choice Insertion and Permanent Choice Insertion patterns correspond to the realization of future decisions related to the selection of an appropriate execution path by introducing a choice construct into the process definition at run-time on the instance and type levels respectively.

**Pattern** FP-11  **CHOICE**

**Description** A process definition specified at design-time contains a choice construct whose execution at run-time results in the selection of one out of several associated choice tasks. Such a decision may depend, for example, on the evaluation of a particular data expression or the availability of an external trigger. This pattern characterizes a need for flexible selection of an execution path anticipated at design-time, thus it corresponds to the *flexibility by design* type.

**Examples**
- When enrolling in a driving course a student has to decide what course of education he/she wants to follow: in-class lectures or self-study.
- A patient visiting a doctor may choose their preferred method of treatment: conventional medical treatment or homeopathic.

**Motivation** In order to allow for flexible selection of an execution path from a set of possible alternatives, the decision points associated with the selection of the task to be executed next can be incorporated in the process definition at design-time. By selecting selecting a task from the set of available options at run-time, a corresponding execution path becomes enabled. Such choices are common in practice as has already been illustrated by the Exclusive Choice and Multi-Choice Workflow Control-Flow patterns (see [55]) and the (Deterministic XOR-Split and Non-deterministic XOR-split) CPN patterns (see [43]). The decision-making may be either non-deterministic or based on the status of a data condition associated with a particular branch. Although multiple options may potentially be selected by a user, in this pattern we assume that a user makes an exclusive choice. The Choice pattern facilitates flexibility in selecting an appropriate execution path by defining a decision point in the process, associating a corresponding XOR-split construct with it and defining a set of alternative tasks which may be selected from.

**Overview** Figure 30 illustrates the graphical notation for the Choice pattern. The top view illustrates the original process definition. Before applying the pattern, in this process definition after task B has completed task C always needs to be executed. The bottom view shows the process definition after applying the
pattern. In this process definition an alternative to task C, i.e. task E, is incorporated. After completing task B, a user has to decide which task to execute next, thus selecting either task C or task E.

**Context**  Figure 31 illustrates the basic process engine which incorporates the functionality for supporting the *Choice* pattern. In order realize this pattern, the process definition used as the basis for process initiation has to include a task after execution of which a choice of one out several alternative branches needs to be performed. For instance, in a process where after task B either task C or task E needs to be executed, the process model needs to be defined as follows: the type of the split-connector associated with task B has to be set to “XOR” (i.e. the task definition should be defined as (‘B’, *generic*, *none*, *XOR*)), and two arcs representing alternative branches have to be added to the list of arcs associated with the process definition (i.e. (B, C), (B, E)). The *Execute task* transition after executing the currently enabled task (B) calculates the new process instance state by means of the *ns()* function. For this, the type of the split-connector of the currently executed task is evaluated, and if it corresponds to the XOR-split, one of the subsequent tasks is chosen non-deterministically.

![Choice pattern](image)

**Implementation**  All of the systems analyzed allow the *Choice* pattern to be realized. In YAWL, the type of the split is associated with a task, and evaluated after is has computed. For an exclusive choice it has to be set to ‘XOR’ in order for exclusive choice to be realized. In ADEPT1, in order for such choice to be defined,
a dedicated construct needs to be inserted and in each of the branches a task from a predefined set templates needs to be selected. The number of alternatives can be increased by adding extra branches. In Oracle BPEL PM, a task cannot be associated with a connector type, therefore in order to represent the XOR-split, a task must be followed by the <switch> construct with multiple cases, each representing an alternative branch. In FLOWer, a choice can be specified either by means of a plan of the user-decision type or the system-decision type. In Declare, both deterministic and non-deterministic choices can be realized by means of constraints. Having multiple unconstrained tasks enables a non-deterministic choice, while by defining constraints for each of the tasks a choice between them becomes enabled. In CIG modeling languages GLIF and EON there are dedicated constructs that can be used to denote an exclusive choice in the process definition. Ashbru achieves such behavior by means of the If-then-else plan. In PROforma, there is a dedicated decision construct that allows one or more options to be selected at run-time.

**Issues** Depending on the number of options available and the conditions for defining the number of options which will execute at the same time, different types of choice constructs are possible. Having only one, several or all options selected, may require the completion of the selected options to be synchronized.

**Solutions** In order to know when to synchronize execution options offered by the choice construct, one has to keep track of the branches selected by means of Boolean variables as described in the Structured Synchronizing Merge and Local Synchronizing Merge WCF patterns (see [55]).

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a decision construct which allows the selection of one of several alternative tasks to be included in the process definition at design-time.

**Pattern FP-12 TASK SUBSTITUTION**

**Description** During execution, i.e. after a process instance has been created, there is the possibility to deviate from the execution path prescribed by the process definition by substituting the currently enabled task with another task contained in the process definition. Eventually, the enabled task is ignored and another task is executed instead. This has no effect on the process definition and other process instances, thus this pattern corresponds to the flexibility by deviation type.

**Examples**
- A student following a course at the driving school is unable to attend a lecture. Instead, he/she studies the lecture material at home using the material provided for self-education or for e-distance learning.
- For a patient who develops an allergy to penicillin, an alternative antibiotic treatment can be made available.

**Motivation** The Choice pattern allows execution alternatives that have been foreseen to be included in the process definition at design-time. However, it is possible that not all execution options have been foreseen. Therefore, at run-time
there may arise the need to execute a task that is not currently enabled but which is included in the process definition. The Task Substitution pattern promotes the flexibility by deviation by allowing a currently enabled task to be substituted with another task in the process definition. Note that such a deviation affects only the execution order of tasks but does not result in the process definition being modified.

Overview Figure 32 illustrates the graphical notation for the Task Substitution pattern. The top view shows that a process instance, in which task C is enabled, before applying the pattern. The bottom view shows that instead of executing task C, a decision to execute another task E has been taken by moving the thread of control from task C to task E. Task E is a generic representation of any task contained in the process definition. After executing this task the thread of control progresses to the task D.

Context Figure 33 illustrates the basic engine, which is enhanced with the Task Substitution pattern, using the CPN formalism. In this diagram, the Substitute task transition is added in order to execute a task different to the currently enabled task. The tid variable represents a task to be executed, which is selected non-deterministically by means of the dev_action() function. After executing this task, the new state of the process instance is calculated by means of the update_dev() function. In this state, task(s) subsequent to the originally enabled task are enabled. Note that since the task substitution does not affect the process definition, the only way to track the effect of this operation is to record tasks that have been executed. For this purpose, the Log place is added. Whenever a task has been completed, a record containing the id of the task is added to the log.

Implementation Of all systems analyzed only FLOWer and Oracle BPEL PM offer some support for this pattern. In FLOWer, this pattern is not supported directly, but a similar result can be achieved by invoking a required task and performing a skip operation for moving the thread of control to a subsequent task. In Oracle BPEL PM, when a task becomes enabled, an exception needs to be raised, which can be handled by executing an alternative task.

Issues When substituting the currently enabled task with another task contained in the process definition, the problem may arise where the substitute task blocks as it requires specific input data elements which should have been provided.
A task which has not yet been executed is selected from the process definition non-deterministically. The execution of the Substitute task transition corresponds to execution of the selected task, which can be tracked only based on the log.

Solutions In order to avoid blocking of a task caused by the absence of required input data, a default data value needs to be defined allowing this task to be enabled and executed when necessary.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that offers a means of substituting a currently enabled task with another task contained in the process definition, such that the process definition remains unchanged, and after the completion of the invoked task the thread of control can move forward as if no substitution has taken place.

Pattern FP-13 LATE SELECTION

Description A process definition created at design-time contains a placeholder which may be completed at run-time in order to allow an alternative execution path that has not been foreseen at design-time to be taken. The completion of the placeholder is optional and is performed only if an execution path different from the default one needs to be taken at run-time. The need for flexible execution path selection in this pattern is recognized at design-time, although its actual realization is performed at run-time. It corresponds to the flexibility by underspecification type.

Examples
- The procedure of acquiring travel insurance via the Internet assumes that by default only one product is requested by a client. However, the client may decide to take this insurance as a substitute part for a bigger package of products offered by the insurance company.
- When requesting a new credit card the client may choose a standard design, or create their own design before submitting their request.
**Motivation**  When different execution paths are foreseen at design-time, they can be included explicitly into the process definition by applying the *Choice* pattern (see p. 41). However, in some situations not all execution options can be foreseen or it may be impractical to specify them all explicitly in the process definition. The *Late Selection* pattern offers the possibility of selecting a particular task in the process definition or an alternative for it that has intentionally been left underspecified at design-time.

**Overview**  Figure 34 illustrates the graphical notation for the *Late Selection* pattern. The top view shows the process instance where after executing task B a decision to take an alternative path has been taken. This is visualized by the enabled placeholder. The bottom view illustrates that at run-time execution the content of the placeholder has been completed with task E, to which the thread of control has been moved.

**Context**  Figure 35 illustrates the the basic process engine enhanced with the flexibility facilitated by the *Late Selection* pattern. The realization of this pattern is similar to that described in the *Choice* pattern with the only difference being that when including an alternative task E in the model, the type of this alternative task has to be set to *placeholder*. Because in the basic process engine no tasks can be executed when an underspecified task is encountered in the set of enabled tasks, in the extended process engine the enabling condition associated with the *Execute task* transition has to be weakened. For this purpose, the $\text{isConcrete}(\text{pid}, \text{m}, \text{st})$ function is replaced with the $\text{existsConcrete}(\text{pid}, \text{m}, \text{st})$ function which evaluates to *True* if a task of the concrete type is available in the set of enabled tasks. Thus, when both a concrete and an underspecified task are enabled, a user may decide whether to execute the concrete task or take an alternative path and define the content of the underspecified task.

**Implementation**  To realize this pattern, a system must support both the *Choice* pattern and allow a task to be left underspecified at design-time. Of all of the systems examined, only YAWL supports this pattern via the worklet service
Fig. 35. Engine enhanced with the Late Selection pattern (note the guard of the `Execute task` transition)

and its ability to associate the XOR-type (i.e. exclusive choice) behavior with a task split construct.

**Issues** The issues discussed for the *Undefined Entry* pattern (see p. 19) and the *Choice* pattern (see p. 41) apply here also.

**Solutions** See the corresponding solutions identified for the *Undefined Entry* and the *Choice* pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a decision in regard to selecting a task or an underspecified alternative to be included in the process definition at design-time, such that the content of the task which has been left underspecified, can be completed when required at run-time.

**Pattern** FP-14  **MOMENTARY CHOICE INSERTION**

**Description** During execution, the process definition associated with a particular process instance can be modified to include a choice between a task predefined in the design-time process definition and an alternative task that previously has not been foreseen. The change applies only to the given process instance and has no effect on other (current or future) process instances. It corresponds to the flexibility by momentary change type.

**Examples**

- A standard education trajectory for students assumes the attendance at lectures for all courses. For a handicapped student a modification to the education program can be made. Attendance at lectures can be replaced with study of the lecture material at home.

- For a passenger who did not confirm their departure date, no place on the flight could be found. The passenger has been offered the option of waiting until all passengers complete the check-in in order to find out whether any free places remain or to be booked on the next connecting flight.
Motivation  In situations where no alternative execution paths have been foreseen and/or incorporated into the process definition at design-time, there may arise the need to introduce an additional execution path for a given process instance. The Momentary Choice Insertion pattern anticipates the need to allow for the selection of an appropriate execution path at a later point in time. The need for flexible execution path selection in this pattern is facilitated by identifying a task for which an alternative task needs to be inserted and associating a choice construct (as described in the Choice pattern on p. 41) with it.

Overview  Figure 36 illustrates the graphical notation for the Momentary Choice Insertion pattern. The top view illustrates two distinct process instances populated from the same process definition. The execution threads indicate that task B is enabled in each of the process instances. For one of the process instances at this point of time an optional path has been foreseen where instead of task C another task E may be selected.

The bottom view illustrates the process instances after the pattern has been applied to the given process instance. As the result of change, the process instance modified includes a choice construct with two alternative tasks C and E, while the other process instance remains unaffected.

Context  Figure 37 illustrates the basic engine enhanced with the Momentary Choice Insertion pattern using the CPN formalism. The engine extension is of the same form as for all patterns facilitating flexibility by momentary change, which has been described earlier on p.35. The change_possible() function specified that a choice construct can be inserted into the given process instance only if there is a sequential task with no split and joins which has not yet been executed. If there are several tasks qualifying this condition, then one task is randomly selected. The update_pi() function creates for the selected task a by-pass path, modifies the type of the split-connector of the task at which the choice needs to
made to the XOR-split, modifies the join-connector of a subsequent synchronization task to the XOR-join, and inserts a new task on the created by-pass path. When during subsequent execution of the process instance the task associated with the choice is executed, either the default path or the newly created alternative path is taken.

Fig. 37. Engine enhanced with the Momentary Choice Insertion pattern

Implementation Of all systems analyzed, only Declare allows a choice to be inserted in a process definition at run-time. By inserting a task and associating a particular constraint with it, an alternative execution path can be inserted. In ADEPT1, although it is possible to add and remove tasks at run-time, there is no functionality for inserting a choice (in one of the versions of ADEPT1, it is possible to associate a code region with outgoing arcs in order to obtain the data-based routing). It is also not possible to add or alternative branch and to modify the type of a task split construct.

Issues This pattern is similar to the Foreseen Bypass pattern (see p. 62), where the ability to omit the execution of a particular task by taking a by-pass is foreseen during design-time. To achieve an optional by-pass path for a given task, no task or an empty task need to be inserted into the alternative path created by the Momentary Choice Insertion pattern.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that allows an alternative execution path, containing a task that has not previously been foreseen, to be inserted in the process definition associated with a given process instance at run-time.

Pattern FP-15 PERMANENT CHOICE INSERTION

Description During execution, there is the possibility to permanently add a choice construct between a task defined in the design-time process definition and an alternative task that previously has not been foreseen. This adds flexibility
when selecting an appropriate execution path. The change performed affects all future process instances, while existing process instances may require migration from the old to the new process definition, thus it corresponds to the flexibility by permanent change type.

Examples

- The product offerings of a software house are extended with a new product. All ongoing product development is unaffected, however for future orders a client can be offered a broader selection of product offerings to address their requirements.

- Due to the increased number of patients, not all patients can undergo blood tests on the same day. To distribute the load, the medical center has signed an agreement with a laboratory, to which the tests are delivered for the required analysis. Since signing this contract, patients are free to choose where their blood will be analyzed.

Motivation

As a result of extension to services/products within a particular process, there may arise the need to include a variety of choices (and thus alternative execution paths) in the original process definition. Although this change can be incorporated in each of the process instances by applying the Momentary Choice Insertion pattern (see p. 47), this approach is less efficient than performing the same change at the type level. The Permanent Choice Insertion pattern allows a process definition to be extended with an alternative execution path at run-time by identifying a decision point in the process and associating a new task with it.

Overview

Figure 38 illustrates the graphical notation for the Permanent Choice Insertion pattern. The top view depicts the execution state of several process instances populated based on the same process definition before applying the pattern. For this process definition, a decision has been taken to include task E which represents an alternative to task C.

![Fig. 38. Permanent Choice Insertion pattern](image)

The bottom view shows that the process definition associated with all process instances has been affected by inserting the choice construct preceding with these
two tasks. For the process instance that has passed task C, this change is inconsequential, while for the other two process instances the ability to choose from several execution paths has become possible.

**Context** Figure 39 illustrates the basic engine enhanced with the *Permanent Choice Insertion* pattern using the CPN formalism. This model adopts the structure described earlier in the *Permanent Entry-Point Change* pattern. The execution of *Permanent choice insertion* transition allows to include an alternative path with a task that has not been foreseen during design-time into the process definition at the type level. The *change_possible()* function specifies that the choice can be inserted only if a task with no splits and joins (other than the start and end tasks) is available in the model. The *modify_m()* function picks an arbitrary task of the sequential type, adds a by-pass path, creates a new task and inserts it in the by-pass path introduced, and modifies the splits and joins of related synchronization tasks to the XOR-type. The *Migrate* transition becomes enabled only if process instances following the process model with the same identifier as the model recently changed is available at the *Running instance* place. Furthermore, the *transfer_possible()* function checks whether the migration is possible, e.g. there must be no changes impacting the enabling of the currently enabled tasks. The *migrate()* function defines the strategy for migrating the existing process instances to the new process definitions.

**Implementation** Of the systems examined, only Declare and YAWL allow an alternative execution path to be inserted by adding a new task and defining scheduling constraints for it. Declare also ensures that all instances are migrated (if possible). In YAWL, an exlet can be called allowing a new worklet to be defined which is associated with a choice between several alternatives tasks. The worklet

---

**Fig. 39.** Engine enhanced with the Permanent Choice Insertion pattern
defined becomes available for other process instances and can be used for the same purpose.

**Issues** Similar issues apply as identified for the *Permanent Entry-Point Change* pattern (see p. 25).

**Solutions** See solutions identified for the *Permanent Entry-Point Change* pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a new task representing an execution alternative to an existing task to be added to the process definition, such that only one of these tasks can be selected at run-time. In addition to this, there must be support for process instance migration.

### 3.5 Flexible reordering

This group of patterns aims at achieving flexibility in alternating the execution order of tasks in a process at run-time execution. Figure 40 illustrates the scope of patterns presented in this subsection and their relationship to different types of flexibility.

<table>
<thead>
<tr>
<th>Flexible Reordering</th>
<th>Design</th>
<th>Deviation</th>
<th>Underspecified</th>
<th>Momentary Change</th>
<th>Permanent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interleaving</td>
<td></td>
<td>Interleaving</td>
<td></td>
<td>Momentary change</td>
<td>Permanent change</td>
</tr>
<tr>
<td>Swap</td>
<td></td>
<td>Swap</td>
<td></td>
<td>Momentary change</td>
<td>Permanent change</td>
</tr>
<tr>
<td>Momentary Reordering</td>
<td></td>
<td></td>
<td></td>
<td>Permanent change</td>
<td></td>
</tr>
<tr>
<td>Permanent Reordering</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 40.** Process Flexibility Matrix: flexible reordering

In situations, where a set of predefined tasks each of which need to be executed is available, but whose eventual execution order may vary for different process instances, it may be necessary to allow for flexible reordering of tasks in the process in order for a suitable execution path to be achieved. This group consists of four patterns: *Interleaving, Swap, Momentary Reordering and Permanent Reordering*. The *Interleaving* pattern allows different task orders to be included in the process definition at design-time on an interleaving basis, i.e. only one of them will be selected at run-time. The other three patterns are characterized by the fact that the need for an alternate execution order is required only at run-time. In the *Swap* pattern the possibility of deviating from the prescribed execution order can
be accomplished by swapping the execution order of two tasks. The Momentary Reordering and Permanent Reordering patterns allow the desired execution order to be achieved by moving a particular task to a desired location in the process definition, thus modifying the process definition both at the instance and type level respectively.

Note that this pattern group has no pattern related to flexibility by under-specification. The reason for this is that if at design-time the need for alternative execution paths is foreseen, they can be either explicitly incorporated into the process definition at design-time (which corresponds to the Interleaving pattern), or the (re-)definition of the ordering relations between the tasks can be postponed until run-time as described in the Late Selection pattern. Effectively, any placeholder associated with a set of predefined tasks can be replaced by an ordered sequence of the tasks from the given set.

**Pattern** FP-16 INTERLEAVING

**Description** A process definition specified at design-time contains two execution paths, each representing alternate execution sequences for two tasks which may be executed in either order but not concurrently. The need for flexible tasks ordering in this pattern is recognized at design-time, and it corresponds to the flexibility by design type.

**Examples**
- When diagnosing patients both a blood test and an MRI scan have to be performed. The order in which the tests are undertaken is not important, however they cannot be conducted concurrently.
- When interacting with another party, a receiver may both send and receive messages, however these operations cannot be done concurrently.

**Motivation** In situations where a specific execution ordering of two tasks needs to be relaxed in order to allow their execution sequence to vary for different process instances, it may be necessary to incorporate alternate execution sequences for these two tasks in the process definition at design-time. The Interleaving pattern allows two execution paths representing alternate sequences of the given tasks to be included in the process definition at design-time such that only of these sequences can be selected at a time.

**Overview** Figure 41 illustrates the graphical notation for the Interleaving pattern. The top view illustrates the process definition before applying the pattern. In this process definition tasks B and C have been identified, as tasks which may execute in either order.

The bottom view illustrates the process definition after applying the pattern. To allow for alternate orderings of these two tasks, a choice construct has been associated with task A. After executing this task the decision needs to be taken in regard to selecting the desired execution order of tasks B and C.

**Context** The basic process engine incorporates the functionality for handling with the Interleaving pattern. In fact, the same functions which are used to support the Choice pattern provide the support for the Interleaving pattern. The
alternate order of tasks that need to be interleaved has to be specified in the process definition. For instance, the process model with id \( m1 \), the start task \( A \) and the end task \( D \), and two alternate sequences of tasks \( B, C \) and \( C, B \) have to be defined as follows:

\[
\]

In this definition, the types of split and join connectors associated with tasks \( A \) and \( D \) are set to the XOR-type, indicating that only one of the branches \( B, C \) or \( C, B \) would be taken.

**Implementation** Oracle BPEL PM supports this pattern by means of serializable scopes or by defining alternative branches for the <switch> construct. In YAWL, ADEPT1 and FLOWer this pattern can be realized by explicitly defining branches with alternative execution sequences, one of which needs to be selected using the means of the Choice pattern (see p. 41). In Declare this pattern is also supported by associating exclusive constraints with the tasks whose execution order needs to be interleaved. In Asbru, this pattern is supported by means of Any-Order plan. PROforma, GLIF and EON require the alternative execution sequences to be explicitly defined as branches associated with a choice construct.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a decision in regard to selecting an execution order of two or more sequential tasks (each of which has to be executed) to be explicitly included in the process definition at design-time.

**Pattern** FP-17  SWAP

**Description** During execution, at any time after a process instance has been created, there is the possibility to deviate from the execution path prescribed by the process definition by swapping the execution order of the currently enabled task and its successor. This has no effect on the process definition and corresponds to the flexibility by deviation type.

**Examples**
In an emergency situation, it may necessary to swap the order of the register patient and perform triage tasks.

Before preparing a patient for medical treatment at a private hospital, an examination by a doctor and preparation of a patient file have to be performed. Depending on the availability of the doctor, the execution order of these tasks may be swapped.

**Motivation** Where the execution order of two sequential tasks needs to be reversed at run-time and it is not possible to include an alternate path in the process definition, there is a need to deviate from the prescribed execution order by swapping the currently enabled task with its successor task. The Swap pattern allows the the execution of the currently enabled task to be postponed until the moment when the execution of the subsequent task completes.

**Overview** Figure 42 illustrates the graphical notation for the Swap pattern. The top view shows a process instance in which task B is enabled. For the given process instance task C needs to be executed first. The bottom view shows that after applying the pattern, the execution order of tasks B and C has been swapped and the thread of control has been moved to task C.

![Fig. 42. Swap pattern](image)

**Context** Figure 43 illustrates the basic engine enhanced with the Swap pattern using the CPN formalism. To support flexibility in task reordering this engine is extended using a generic structure for deviations described earlier in the Termination Skip pattern on p.31. Next to modifying the content of the functions, in this model the information about the initiated process instances stored in the Running instances place has been extended with additional state information:

```
colset ProcInstStSt = product piID * ProcModel * State *State;
```

This information is required in order to keep track of tasks that had to be executed but the execution of which has been postponed because another task has been executed instead. The Deviate transition allows a task that has not yet been executed and which is not currently enabled to be executed, thus deviating from the normal flow of control. The deviation_possible() function specifies that the Deviate transition can be only executed if there is an enabled task that needs to be executed, the process end task has not yet been reached and there is a task other than currently enabled task which can be executed instead. The
dev_action() function selects an arbitrary task from the set of tasks that have not yet been executed and which are not enabled. After executing the selected task \( \text{tid} \), the record about its execution is added to the Log place.

The update_dev() function does not modify the model, but updates both states associated with the process instance. The normal state contains information about subsequent tasks that become enabled after the task execution, and the negative state keeps the record of the task which had to be executed but whose execution has been postponed. This information is used by the exectask() function for determining what task needs to be executed next. If there are tasks in the negative state, one of these tasks is selected non-deterministically. If no tasks whose execution has been postponed can be found, e.g. the negative state is empty, a task from the normal state is selected. The update_exec() function updates both states after the task has been executed. If the task from the negative state has been selected, this task is removed, otherwise a new state according to the process model is calculated.

**Implementation** Only FLOWer and Declare offer direct support for this pattern. In FLOWer, a task that is not currently enabled can be immediately initiated. After completion of this task the thread of control returns back and originally enabled task can be executed.

**Issues** Swapping the execution order of tasks may result in data dependencies between them being invalidated. The absence of data required for enabling the subsequent task may result in its blocking when the decision to execute this task before the preceding task has been made and the data dependencies not been satisfied.

**Solutions** To loosen the dependency between tasks and allow their execution order to be switched, input variables for each of the tasks have to be set to a default value.

Fig. 43. Engine enhanced with the Swap pattern
Evaluation Criteria  Full support for this pattern is demonstrated by any offering that allows the execution order of a task and its successor in a process to be reversed without modifying the process definition.

Pattern  FP-18  MOMENTARY REORDERING
Description  During execution, at any time after process initiation, the process definition associated with a particular process instance can be modified by changing the execution order associated with a particular task. Having a task moved to another place in the process definition allows the execution of the task to be postponed until a desired moment, thus enabling an execution outcome that has not been foreseen at design-time to be effected. The need for task reordering in this pattern is recognized at run-time. The change applies only to the given process instance and has no effect on other (current or future) process instances, thus it corresponds to the flexibility by momentary change type.

Examples
- Based on a request from a client, the travel advisor changed the order of events planned for the booked trip. Because of the late arrival, the client cannot attend the first planned social event. The ticket for this activity has been re-booked on a later date.
- A typical procedure for car-leasing starts with the recording of payment details. For repeat clients, an exception can be made, where they can pick-up the car on an urgent basis and arrange the payment afterwards.

Motivation  In situations where an execution order of two sequential tasks needs to be reversed for a particular process instance at run-time and it is not possible to deviate from the order prescribed by swapping these tasks (as described in the Swap pattern), there is the need to temporarily change the order of these tasks in the process definition in order to achieve the desired behavior. The Momentary Reordering pattern allows the currently enabled task to be moved later in the process definition in order to allow a subsequent task to be executed first.

Overview  Figure 64 illustrates the graphical notation for the Momentary Reordering pattern. The top view shows two distinct process instances populated based on the same process definition. For the process instance, whose execution thread is depicted by triangle, the need to interchange the order of the currently enabled task B with its successor task C is recognized.

The bottom view illustrates that after applying the pattern the process definition associated with the given process instance has been modified by reordering these tasks. The thread of control in this process instance has been moved to task C, whilst the other process instance remains unaffected.

Context  Figure 45 illustrates the basic process engine enhanced with the Momentary Reordering pattern using the CPN formalism. To allow for momentary choice insertion, the functions update_pi() and change_possible() have been modified as follows. The Momentary choice insertion transition can be only
executed if there is an enabled task and if the enabled task is not the end task.

The \texttt{update_pi()} function selects a task without splits or joins that has not yet been executed and an arc between two other tasks which also have not been executed, where the selected task will be moved to. The selected task is deleted, tasks from the preset and postset of the removed task are reconnected, and the removed task is inserted into the selected arc.

\begin{center}
\textbf{Fig. 44.} Momentary Reordering pattern
\end{center}

**Implementation** Declare supports this pattern by modifying constraints defining the execution order of tasks. In ADEPT1 it is not possible to move tasks from one place to another, however two tasks can be swapped by deleting these tasks first and inserting them in the reverse order. None of the other tools investigated support this pattern.

**Issues** Similar issues as identified for the \textit{Swap} pattern apply here too.

**Solutions** See solutions identified for the \textit{Swap} pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the process definition associated with a given process instance
to be modified in order to reverse the execution order of two subsequent sequential tasks which have not yet been executed.

**Pattern** FP-19 PERMANENT REORDERING

**Description** During execution, there is the possibility to permanently modify the execution order associated with a particular task in the execution sequence. The task can be moved before or after its current position in order to hasten or postpone its execution respectively. The change performed affects all future process instances directly, while existing process instances may require migration from the old to the new process definition, thus it corresponds to the *flexibility by permanent change* type.

**Examples**
- In order to improve communication between students working together in projects, the presentation and communication skills course has been moved at the beginning of the education program. The current generation of students continues with the old program, while new students participate in the updated program.
- The procedure for issuing residence permits to foreign visitors has been changed in the following way: before the application for a residence permit can be accepted, the applicant must personally visit the office and pay the fee. In the past, applicants could submit applications at any time, which required the payment to be performed only when the application has been accepted.

**Motivation** In situations where at run-time the need to rearrange the order of tasks in a process is anticipated, a set of actions may need to be taken in order to redesign the original process definition and to adapt the execution of current and future process instances. The *Permanent Reordering* pattern allows the order of two sequential tasks in a process to be changed by permanently moving the currently enabled task in order to postpone its execution and let its subsequent task to execute instead.

**Overview** Figure 46 illustrates the graphical notation for the *Permanent Reordering* pattern. The top view shows the execution state of three distinct process instances, sharing the same process definition. At run-time execution the need to change the order of tasks B and C at the type level is recognized.

![Fig. 46. Permanent Reordering pattern](image)
The bottom view illustrates the process definition after applying the pattern. The order of tasks B and C has been swapped. For process instances where task B was enabled, the thread of control has been moved to task C. For process instances that have passed task B, this change is inconsequential.

Context Figure 47 illustrates the basic process engine enhanced with the Permanent Reordering pattern using the CPN formalism. In order to permanently change the execution order of two sequential and independent tasks, the process definition stored in the Process definition place needs to be adjusted and all existing process instances have to be migrated. For this, the model of the basic process engine is extended with the Permanent task move transition, which modified the process definition both for a given process instance and the original process definition stored in the Process definition place, and the Migrate transition which transfers old process definition oldm associated with each process instances stored in the Running instance place to the new process definition newm. The mechanism of reordering tasks in a process definition is accomplished using the moveTask(tID,inarc,m) function in the same way as described for the Momentary Reordering pattern.

Implementation Of all of the systems analyzed only ADEPT1, Declare and YAWL support this pattern. ADEPT1 allows the tasks to be removed and to be inserted in the reversed order, providing that all existing process instances migrate to the new process definition. In Declare, the execution order between tasks can be changes by modifying constraints associated with these tasks, and migrating all existing process instance to the updated process definition. In YAWL it is possible to invoke an exlet for executing tasks in an alternate order and to skip the execution of these tasks in the original order when the execution of the exlet

---

**Fig. 47.** Engine enhanced with the Permanent Reordering pattern
completes. The alternate ordering of tasks in this case is defined via a worklet, which becomes available for other process instances.

**Issues** The same issues apply as identified for the *Permanent Entry-Point Change* pattern (see p. 25) and the *Swap* pattern (see p. 54) apply.

**Solutions** See solutions identified for the above mentioned patterns.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the process definition to be permanently modified at run-time execution in order to reverse the execution order of two tasks which have not yet been executed. In addition, there must be support for process instance migration.

### 3.6 Flexible elimination

This group of patterns aims to facilitate flexibility by avoiding the execution of a particular task. Figure 48 illustrates the scope of patterns presented in this subsection and their relation to different types of flexibility.

<table>
<thead>
<tr>
<th>Flexibility by</th>
<th>Design</th>
<th>Deviation</th>
<th>Underspecification</th>
<th>Momentary Change</th>
<th>Permanent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible initiation</td>
<td>Alternative entry-point</td>
<td>Entry point</td>
<td>Undefined entry</td>
<td>Momentary entry change</td>
<td>Permanent entry change</td>
</tr>
<tr>
<td>Flexible termination</td>
<td>Alternative exit-point</td>
<td>Termination point</td>
<td>Undefined exit</td>
<td>Momentary exit change</td>
<td>Permanent exit change</td>
</tr>
<tr>
<td>Flexible selection</td>
<td>Choice</td>
<td>Task substitution</td>
<td>Late selection</td>
<td>Momentary choice operation</td>
<td>Permanent choice operation</td>
</tr>
<tr>
<td>Flexible embedding</td>
<td>Embedding</td>
<td>Swap</td>
<td>Momentary embedding</td>
<td>Permanent embedding</td>
<td></td>
</tr>
<tr>
<td>Flexible elimination</td>
<td>Foreseen bypass</td>
<td>Task skip</td>
<td>Momentary task elimination</td>
<td>Permanent task elimination</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 48.** Process Flexibility Matrix: flexible elimination

In situations, where the execution of a particular task needs to be avoided, there may arise the need to provide the ability to omit the execution of a task. Depending on the moment at which the need for eliminating the execution of a particular task is realized, we distinguish four patterns: *Foreseen Bypath*, *Task Skip*, *Momentary Task Extraction* and *Permanent Task Extraction*. Of these four patterns, the *Foreseen Bypath* pattern is characterized by anticipating and realizing an optional bypass for a task, whose execution may need to be avoided, at design-time. In the other three patterns the need to eliminate the execution of a given task is anticipated and realized at run-time. The *Task Skip* pattern allows a currently enabled task to be skipped, which corresponds to deviation from the execution path prescribed by the process definition. The *Momentary Task Extraction* and *Permanent Task Extraction* patterns allow the execution of a given
task to be avoided by removing it from the process definition on the temporary or permanent basis.

Note that the idea of flexible task elimination has no straightforward mapping to flexibility by underspecification. The context conditions of this pattern group assume that the execution order and the content of the task which may need to be eliminated is known and fixed at design-time, whilst flexibility by underspecification assumes that the content of particular process fragment that is marked as placeholder cannot be precisely defined until runtime.

Pattern FP-20 FORESEEN BYPASS

Description A process definition specified at design-time contains a bypass path associated with a particular task, which if taken at run-time results in the task not being executed. This pattern recognizes the need for flexible task elimination which is anticipated at design-time, and corresponds to the flexibility by design type.

Examples

– The composition of the education program for postgraduate students allows some of the obligatory courses to be skipped if they have been completed during previous study.

– At the end of a course all students are offered a questionnaire to fill in investigating possible areas for improvement. Completion of the questionnaire is optional.

Motivation A typical process definition specifies the order in which tasks need to be executed. In some situations, it may be necessary to allow the execution of a particular task to be executed on an optional basis. This requires flexibility in choosing whether or not to execute a given task. The Foreseen Bypass pattern allows the possibility for optionally excluding a particular task to be included into the process definition at design-time.

Overview Figure 49 illustrates the graphical notation for the Foreseen Bypass pattern. The top view shows the process definition before applying the pattern. According to this process definition, after executing task A always task B has to be executed. In this process, the optional execution of task B recognized at design-time.

![Fig. 49. Foreseen Bypass pattern](image-url)
The bottom view shows the process definition after applying the pattern. In order to offer a choice between executing task $B$ and bypassing it, task $A$ is associated with a choice construct and an alternate direct path to task $C$ is inserted.

**Context** The *Foreseen Bypass* pattern is supported by the basic process engine in the similar manner as by the *Choice* pattern described on p.41. The realization of a bypass is equivalent to realization of a choice construct where one of branches has no tasks. In order to incorporate the bypass in the process definition, two tasks performing the synchronization of branches need to be identified and their connectors have to be set to the XOR-type.

**Implementation** In Declare it is possible to by-pass a task by violating an optional constraint. In AshbruView, the *If-Then-Else* plan can be used where either a desired task or an empty task can be selected. In all other systems investigated, a by-pass associated with a particular task can be realized the same way as for the *Choice* pattern, providing that one of the branches contains no tasks.

**Issues** Selection of a by-pass associated with a specific task may result in a subsequent task blocking due to missing data that should have been provided by the bypassed task.

**Solutions** In order to avoid blocking of the subsequent task, mandatory data input elements for this task should be set to have a default value.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows support for optional task execution to be included into the process definition at design-time.
bypass for the task (as described in the Foreseen Bypass pattern) that can be taken in order to obviate the need to execute it, other means of eliminating the currently enabled task are required. The Skip pattern allows the execution of a given process instance to deviate from the execution path prescribed by the process definition by skipping the currently enabled task.

**Overview**  Figure 50 illustrates the graphical notation for the Task Skip pattern. The top view shows a process instance before applying the pattern. For this process instance the need to avoid the execution of the currently enabled task \( B \) is recognized.

![Fig. 50. Task Skip pattern](image)

The bottom view shows that after applying the pattern the currently enabled task \( B \) is skipped (without modifying the process definition), and the thread of control has been moved to task \( C \).

**Context**  Figure 51 illustrates the basic process engine enhanced with the Task Skip pattern using the CPN formalism. In order to allow for the execution of a currently enabled task to be ignored, the content of the functions `deviation_possible()`, `dev_action()`, and `update_dev()` have been modified as follows. The `dev_action()` function determines the currently enabled task \( \text{tid} \) whose execution needs to be ignored. The Skip transition can only be executed if there are tasks enabled in the process instance state which have not yet been executed. The `update_dev()` function updates the state of the process instance by moving the thread of control to a subsequent task. Note that the process definition remains unchanged.

**Implementation**  In Declare it is possible to skip a task by violating an optional constraint. In FLOWer there is a special operation that allows a currently enabled task to be skipped. In PROforma it is possible to skip execution of the prescribed task. All other systems investigated offer no support for this pattern.

**Issues**  Similar issues and solutions as for the Entrance Skip pattern (see p.16) apply here too.

**Solutions**  See solutions identified for the Entrance Skip pattern.

**Evaluation Criteria**  Full support for this pattern is demonstrated by any offering that provides an explicit operation to skip the execution of a specific task at run-time, without affecting the process definition associated with the given process instance.
**Pattern FP-22  MOMENTARY TASK ELIMINATION**

**Description** During execution, at any time after the process initiation, the process definition associated with a particular process instance can be modified by removing a currently enabled task from the process definition associated with the given process instance. This allows for flexible elimination of a currently enabled task, should this task no longer be required. The change applies only to the given process instance and has no effect on other (current or future) process instances, thus it corresponds to the *flexibility by momentary change* type.

**Examples**

- For visitors to a football match who hold a European passport, no visa requirements apply. This modification in the customs procedure is performed on a person-by-person basis for the period of the championship competition.

- A time registration program provides a timesheet with predefined fields. A task which is included in the timesheet (for instance, planned sick leave for a doctor’s appointment), but which has not been undertaken during the reporting period, can be removed from the timesheet.

**Motivation** When at run-time a particular task needs to be ignored, and there is no a by-pass defined in the process definition for skipping the given task, and it is not possible to deviate from the prescribed execution order, there may arise the need to temporarily modify the process definition to exclude the given task. The *Momentary Task Elimination* pattern allows a particular task to be removed from the process definition in order to avoid executing the task for a given process instance.

**Overview** Figure 52 illustrates the graphical notation for the *Momentary Task Elimination* pattern. The top view shows two distinct process instances populated based on the same process definition before applying the pattern. For one of these process instances the need to eliminate the currently enabled task B is recognized.

The bottom view shows that after applying the pattern the process definition associated with one of the process instances has been modified. In particular,
task B has been removed from the process definition, and the thread of control
associated with the given process instance has been moved forward to task C.

**Context** Figure 53 illustrates the basic process engine enhanced with the *Momentary Task Elimination* pattern using the CPN formalism. The model adopts
the structure presented earlier in the *Momentary Entry-Point Change* pattern (see p.35), however the content of functions change_possible() and update_pi() has
been modified. The *Momentary task extract* transition can only be executed if
there are enabled tasks in the process instance state which have not been executed
and if the end of the process has not been reached. The update_pi() function
selects a currently enabled task, deletes it from the process definition, and connects
the task preceding it to a subsequent task by means of an arc. Once that task has
been removed, the process instance state is also recalculated.

**Implementation** In Declare it is possible to remove a specific task from
the model. ADEPT1 also allows a task to be deleted from the model, however
the deleted task is visualized as disabled rather than extracted from the process
definition associated with the given process instance. In YAWL there is the pos-
sibility to invoke an exlet for removing a workitem, however when the workitem
has been removed, no subsequent tasks can be executed. Instead, it is possible to force failure or completion of the given task in order to proceed execution of the next task. No other systems that have been investigated offer the support for this pattern.

**Issues**  Removing a task from the process definition may trigger blocking of subsequent tasks requiring output data provided by the task removed.

**Solutions**  After removing a task, either all data dependencies have to also be removed or a default value has to be assigned to the input data elements of subsequent tasks.

**Evaluation Criteria**  Full support for this pattern is demonstrated by any offering that allows a task to be deleted from the process definition associated with a particular process instance.

**Pattern**  FP-23  PERMANENT TASK ELIMINATION

**Description**  During execution, there is a possibility to permanently eliminate the execution of a particular task in the process by removing it from the process definition at the type level. This allows the selected task to be ignored for all future process instances and existing process instances which have not passed this point yet. The change performed affects all future process instances directly, while existing process instances may require migration from the old to the new process definition, thus the pattern is of the flexibility by permanent change type.

**Examples**

- A template for project schedule can be updated by removing tasks which do not need to be executed any more.
- Due to centralization of data by the city hall, tax office and insurance companies, changes in family living arrangements do not need to be reported to the insurance company directly. This step is removed from the predefined procedures of the insurance company. The data is instead reported to the insurance company by the tax office.

**Motivation**  In situations where after process initiation the need to eliminate the execution of a particular task for current and future process instances is recognized, there may arise a need to modify the process definition on the type level. The Permanent Task Elimination pattern allows a particular task to be permanently removed from the process definition such that this task will never be executed again.

**Overview**  Figure 54 illustrates the graphical notation for the Permanent Task Elimination pattern. The top view shows the execution state of several process instances populated based on the same process definition. At run-time execution the need to eliminate task B at the type level is recognized.

The bottom view shows that the process definition after applying the pattern. Task B has been removed from the process definition. For process instances where task B was enabled, the thread of control has been moved forward to its subsequent task C.
**Context**  Figure 55 illustrates the basic process engine enhanced with the *Permanent Task Elimination* pattern using the CPN formalism. The model of the engine is extended according to the structure described earlier in the *Permanent Entry-Point Change* pattern (see p.25). The functions used have been changed as follows. The *Permanent task extract* can only be executed if the process definition contains more than two tasks (e.g. start and end tasks). This condition is incorporated in the `change_possible()` function. The `modify_m()` function removes the selected task from the process definition \( m \), and reconnects the tasks preceding and following it. The `Migrate` transition performs migration of existing process instances stored in the *Running instance* place only if the process instance has reached the end. For the process instance whose process model id matches with the id of the process definition earlier modified, the `migrate()` function transfers old process definition \( oldm \) to the new process definition \( newm \) according to a desired migration policy.

---

**Implementation**  Of all systems analyzed, only Declare and ADEPT1 support this pattern. In addition to the functionality for removing a task from the model, they also provide a means for process instance migration.
Issues  The same issues as identified for the Permanent Entry-Point Change pattern (see p. 25) and the Momentary Task Elimination pattern apply here too.

Solutions  See solutions identified for the above mentioned patterns.

Evaluation Criteria  Full support for this pattern is demonstrated by any offering that allows a task to be permanently removed from the process definition, providing the process instance migration.

3.7 Flexible extension

This group of patterns aims at providing flexibility in enabling an execution path alternative to the one prescribed by the process definition by incorporating a task that has not previously been foreseen (i.e. no reordering or selection). Figure 56 illustrates the scope of patterns presented in this subsection and their relationship to different types of flexibility.

<table>
<thead>
<tr>
<th>Flexibility by</th>
<th>Design</th>
<th>Deviation</th>
<th>Underspecification</th>
<th>Momentary Change</th>
<th>Permanent Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible invocation</td>
<td>Alternative entry point</td>
<td>Undefined entry</td>
<td>Undefined entry</td>
<td>Momentary entry change</td>
<td>Permanent entry change</td>
</tr>
<tr>
<td>Flexible termination</td>
<td>Alternative entry point</td>
<td>Undefined entry</td>
<td>Undefined entry</td>
<td>Momentary entry change</td>
<td>Permanent entry change</td>
</tr>
<tr>
<td>Flexible selection</td>
<td>Choose</td>
<td>Task substitution</td>
<td>Late selection</td>
<td>Momentary choice insertion</td>
<td>Permanent choice insertion</td>
</tr>
<tr>
<td>Flexible ordering</td>
<td>Introducing</td>
<td>Swap</td>
<td>Momentary ordering</td>
<td>Permanent ordering</td>
<td></td>
</tr>
<tr>
<td>Flexible elimination</td>
<td>Foreseen bypass</td>
<td>Task skip</td>
<td>Momentary task elimination</td>
<td>Permanent task elimination</td>
<td></td>
</tr>
<tr>
<td>Flexible extension</td>
<td>Task invocation</td>
<td>Late creation</td>
<td>Momentary task insertion</td>
<td>Permanent task insertion</td>
<td></td>
</tr>
<tr>
<td>Flexible concurrency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flexible repetition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 56. Process Flexibility Matrix: flexible extension

In situations, where at run-time a task needs to be executed which has not been foreseen in the process definition, it should be possible to easily incorporate the desired task in order for desired behavior to be achieved. The need for flexible extension in this group of patterns is recognized at run-time. The concept of flexible extension cannot be facilitated at design-time, because all execution alternatives that are foreseen at design-time can explicitly be included into the process definition by applying the Choice pattern (see p. 41) as illustrated in Figure 57.

There are four flexible extension patterns: Task Invocation, Late Creation, Momentary Task Insertion and Permanent Task Insertion which describe four approaches to incorporating unforeseen tasks in order for a desired execution option to be obtained at run-time. The Task Invocation pattern allows for deviation from the execution path prescribed by invoking a task different from the one that
is currently enabled without replacing any existing task. The *Late Creation* pattern offers the possibility of creating an alternative execution path at a particular point in the process and at the last possible moment based on the need to introduce optional behavior that could not be foreseen at design-time. The *Momentary Task Insertion* and *Permanent Task Insertion* patterns allow for inclusion of a task that has not been foreseen at design-time in the process-definition on a temporary and permanent basis respectively.

**Pattern** FP-24 **TASK INVOCATION**

**Description** During execution, at any time after a process instance has been created, there is the possibility to deviate from the execution path prescribed by the process definition by invoking a task which has not yet been executed and completing it before the currently enabled task is executed. This allows for flexible extension of the execution options associated with a given process instance with one of the tasks contained in the process definition. Such a deviation does not affect the process definition, and corresponds to the *flexibility by deviation* type.

**Examples**
- Students possessing diploma's from European institutions do not have to pass a language test in order to enroll into the masters program. In some cases however, the level of English ability may need to be assessed. For this an appointment is made with an advisor handling English tests for foreign students.
- A patient treated in the department of internal diseases has developed a skin rush. To handle this problem a doctor from the dermatological department has been called.

**Motivation** In some situations, after a process has been initiated, it may be necessary to postpone the execution of a certain task because another task that has not been previously foreseen at design-time needs to be executed first. The *Task Invocation* pattern allows for suspension of the execution of tasks prescribed by the process definition in order for another task to be invoked and executed first. The possibility of extending the set of tasks prescribed by the process definition
in this pattern corresponds to flexibility by deviation, which has an impact only on the manner in which the given process instance is executed, and does not affect the original process definition.

**Overview** Figure 58 illustrates the graphical notation for the *Task Invocation* pattern. The top view illustrates a process instance where after executing task B the need for a task different to the one prescribed by the process definition is recognized.

The bottom view shows that after applying the pattern task E has been invoked and the thread of control from task C has been temporarily moved to task E. Note that task E is a generic representation of a task that is contained in the process definition but which has not been executed yet. The invocation of this task is performed without modifying the process definition.

**Context** Figure 59 illustrates the basic process engine enhanced with the *Task Invocation* pattern using the CPN formalism. The model adopts the structure for deviation operations described earlier in the *Termination Skip* on p.31. The *Invoke Task* transition can only be executed for process instance whose execution has not completed yet. The *dev_action* function selects a task that has not yet been executed and which will be invoked before executing the currently enabled task. The invoked task is recorded in the Log place, and the state of the process instance is updated by the *update_dev()* function so that the originally enabled task becomes enabled again.

**Implementation** Of all of the systems analyzed, only FLOWer, Declare, and Oracle BPEL provide support for this pattern. In FLOWer another task can be invoked at any point in the process. In Oracle BPEL PM, an exception needs to be raised and in the corresponding exception handler an invocation activity needs to be included.

**Issues** None identified.

**Solutions** N/A.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a task contained in the process definition which has not been executed yet to be invoked. Execution of the main process must be suspended until the execution of the invoked task completes.
**Pattern** FP-25  **LATE CREATION**

**Description** A process definition created at design-time contains a placeholder which may be completed at run-time in order to allow for an extension of the process definition with a task whose content cannot be foreseen in advance. The completion of the placeholder is optional and is performed only if an additional task needs to be added at run-time. The need for flexible execution path selection in this pattern is recognized at design-time, whose actual realization is performed at run-time, thus this pattern is of the flexibility by underspecification type.

**Examples**
- The process of submitting electronic photos for printing starts with the uploading of photos into a photo application, selection of the print format and submission of the order. The client may submit the order immediately, or order additional products offered by the company, and only then proceed to order submission.
- The software program supplied with the navigation system offers a default voice for route directions. When defining a route, a user may also choose to record their own instructions and then move on to planning the route.

**Motivation** At design-time when creating a process definition it may be foreseen that an additional task may need to be added at run-time, whose content is either unknown or may vary for different process instances. The Late Creation pattern allows for the possibility for creating an additional task at the latest possible moment to be included in the process definition in the form of a placeholder. The definition of this placeholder may be completed at run-time if for a given process instance a new task needs to be added. Otherwise it can be ignored.

**Overview** Figure 60 illustrates the graphical notation for the Late Creation pattern. The top view shows a process definition where after executing task \( B \)
there is the possibility of executing task C or creating another task and executing it before proceeding to task C. The possibility for late creation of a task is incorporated into the process definition by means of a placeholder.

![Diagram showing Late Creation pattern]

**Fig. 60. Late Creation pattern**

The bottom view shows that after applying the pattern, a new task E is created when completing the definition of the enabled placeholder, and the thread of control is passed to this task.

**Context** The CPN semantics of the Late Creation pattern can be described in terms of the Choice pattern (see p. 41) and the Late Selection pattern (see p. 45). In order for flexibility in process extension facilitated by the Late Creation pattern to be obtained, the process definition created at design-time has to include the choice construct with two alternative branches: one representing the normal flow of tasks and another containing a placeholder task whose content can be created when needed at run-time. Figure 61 pattern shows the basic process engine modified in a similar way as has been done for the Late Selection pattern. The enabling condition for the Execute task transition has been weakened in order to allow the user to choose whether to execute a concrete task or to define the content of an underspecified task. This condition is incorporated in the \texttt{existsConcrete(pid,m,st)} function.

**Implementation** Of all systems analyzed, only YAWL and Oracle BPEL PM support this pattern. By means of incorporating a worklet into one of the alternative branches associated with the exclusive choice construct (see the Choice pattern on p. 41) it is possible to postpone the decision for late task creation until run-time. In Oracle BPEL PM, a task responsible for invoking a process whose implementation details are not yet known can be defined at design-time. The implementation details can be set dynamically during the process execution.

**Issues** The same issues as for the Late Selection pattern (see p. 45) apply here also.

**Solutions** See solutions identified for the Late Selection pattern.
**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a decision in regard to executing a task prescribed by the process definition or creating and executing a previously unforeseen task, to be included in the process definition at design-time.

**Pattern** FP-26  **MOMENTARY TASK INSERTION**

**Description** During execution, at any time after process initiation, the process definition associated with a particular process instance can be modified by extending the process definition with a task that has not been foreseen. The execution of the currently enabled task is postponed until the newly identified task has been completed. The change applies only to the given process instance and has no effect on other (current or future) process instances, thus it corresponds to the flexibility by momentary change type.

**Examples**

- A timesheet used for recording activities performed by an employee during the day contains a list of predefined tasks. If a non-standard task that is not included in the list needs to be performed, the timesheet for the corresponding day can be adjusted by inserting the new task.

- Due to the unexpected illness of an employee, the tasks of this employee are divided amongst other members of the group. Each employee to whom an extra task has been assigned adjust their personal planning accordingly.

**Motivation** When at run-time a desired task cannot be found in a process definition, it should be possible to introduce this task on a temporary basis. The Momentary Task Insertion pattern allows the set of tasks to be executed by a particular process instance to be extended by temporarily inserting a task into the corresponding process definition.

**Overview** Figure 62 illustrates the graphical notation for the Momentary Task Insertion pattern. The top view shows two process instances populated based on the same process definition before applying the pattern. Both process instances are
in a state where task C is enabled and needs to be executed. For one of the process instances, the need to execute a task not contained in the process definition is recognized.

Fig. 62. Momentary Task Insertion pattern

The bottom view shows the process definitions associated with given process instances after applying the pattern. The process instance where a new task was introduced has been extended with task E, and the execution thread has been moved backward from the previously enabled task C to task E. The other process instance remains unaffected.

**Context** Figure 63 illustrates the basic process engine enhanced with the *Momentary Task Insertion* pattern using the CPN formalism. The process engine is extended according to the structure adopted by all patterns facilitating flexibility by momentary change (see p.35. To realize the functionality for extending the set of tasks associated with a particular process instance the change_possible() and update_pi() functions have been updated as follows. The change_possible() function specifies that a new task that previously has not been foreseen inserted into the process definition by the *Momentary task insertion* transition only if for a given process instance there are enabled tasks that have not yet been executed. The update_pi() function creates a new task, selects an arc connecting two tasks between the created task will be inserted, and replaces this arc with two arcs connected to the created task respectively.

**Implementation** Of all systems analyzed, only Declare, ADEPT1 and YAWL support this pattern. In Declare and ADEPT1 there is an explicit operation defined allowing a new task to be inserted in a model. In YAWL, at run-time execution it is possible to invoke an exlet in order for a process fragment that has not been earlier foreseen to be executed while the main process is suspended.

**Issues** By inserting a task between two tasks in the process, any data assumptions that exist between these tasks can potentially be disrupted resulting in the last task blocking.

75
Fig. 63. Engine enhanced with the Momentary Task Insertion pattern

Solutions In order to solve this problem, the visibility of data used in the process has to be set to the process instance level, so that data produced at each step during the process becomes visible to other tasks.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that allows a task to be inserted into a process definition associated with a given process instance at run-time.

Pattern FP-27 PERMANENT TASK INSERTION

Description During execution, there is the possibility to permanently modify the process definition by adding a new task. This allows for functionality that has not been foreseen earlier to be incorporated in all existing and future process instances. The change performed affects future process instances directly, whilst existing process instances require their old process definition to be migrated to the new process definition, thus this pattern corresponds to the flexibility by permanent change type.

Examples
- The introduction of electronic identity cards requires all traveling passengers to hold a passport with an electronic id. Passengers whose passport has not expired yet, continue to use it during the transition period, however when this passport expires a new one must be requested. The issuing of an electronic id is an additional step performed after the new passport has been acquired.
- The takeover of one company by another requires product certification, which previously has not been done. This implies that all ongoing requests have to be terminated and redone according to the new steps dictated by the certification standard.

Motivation In situations where at run-time the need to extend the set of tasks contained in a process is anticipated, a set of actions may need to be taken in order to redesign the original process definition and to adapt the execution of current and future process instances. The Permanent Task Insertion pattern
allows a task that has not previously been foreseen to be included in the process definition on a permanent basis.

Overview Figure 64 illustrates the graphical notation for the Permanent Task Insertion pattern. The top view shows several process instances sharing the same process definition. At run-time execution the need to execute task E which has not previously been foreseen is recognized on the type level.

![Fig. 64. Permanent Task Insertion pattern](image)

The bottom view shows that after applying the pattern, the process definition has been extended with task E. The process instance that has passed this point remains unaffected, while for other process instances the thread of control has been migrated from previously enabled task C to the newly inserted task E.

Context Figure 65 illustrates the process engine enhanced with the Permanent Task Insertion pattern using the CPN formalism. The model of the process engine is extended with the Permanent task insertion and Migrate transition according to the structure adopted by all patterns facilitating flexibility by permanent change and described earlier on p.25. In order to insert a task that previously has not been foreseen into the process definition at the type level by means of the Permanent task insertion transition, functions change_possible() and modify.m() have been modified as follows. A newly created task is inserted into the selected arc by means of the modify.m() function by splitting a selected arc into two parts. The process instances adhering to the process model with the same identified as the process definition recently modified, which are stored in the Running instance place, may need to be migrated to the new process definition. The transfer_possible() function checks whether transfer is possible for the given process instance (there are enable tasks which have not yet been executed and transfer is possible). If migration needs to be performed, the migrate() function transfers the corresponding process instances from the old to new process definition.

Implementation Of all of the systems analyzed, only Declare and ADAPT1 support this pattern. They offer an operation for inserting a task into a process model and allow existing process instances to be migrated to the new process definition.
Fig. 65. Engine enhanced with the Permanent Task Insertion pattern

**Issues** The same issues as identified for the Permanent Entry-Point Change pattern (see p. 25) and the Momentary Task Reordering pattern apply here also.

**Solutions** See solutions identified for the Permanent Entry-Point Change pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a task to be permanently inserted into a process definition at runtime, and provides facilities for migrating existing process instances from old process definition to the new one.

### 3.8 Flexible concurrency

This group of patterns aims at achieving flexibility when executing several independent tasks concurrently, thus avoiding unnecessary dependencies between them. Although the similar outcome can be achieved by applying patterns facilitating flexible reordering, the focus of this group of patterns is not to change the order of tasks, but to attain the desired concurrency. Figure 66 illustrates the scope of patterns presented in this subsection and their relationship to different types of flexibility.

Often sequential processes which require that tasks be executed one after another contain a set of tasks whose execution does not depend on each other. To decrease the duration of the process and make its execution more efficient, independent tasks should be undertaken concurrently. The flexible concurrency patterns describe different ways in which dependencies between independent tasks can be decreased by letting them run in parallel. This group contains four patterns: *Parallelism, Momentary Task Parallelization* and *Permanent Task Parallelization* corresponding to flexibility by design, momentary change and permanent change respectively. Note that no mapping of flexible concurrency to the concepts of flexibility by underspecification can be made. Flexibility by underspecification assumes...
that the content of a particular process fragment is not foreseen at design-time and control-flow dependencies are not known, whilst the tasks whose execution needs to be parallelized in this case are well-defined and no dependencies between these tasks are required. The mapping on flexibility by deviation corresponds to the *Swap* pattern described on p.54, which enables the deviation from executing the currently enabled task and invoking another task. Considering the interleaving semantics of CPNTools the same realization mechanism as used for the *Swap* pattern applies for supporting flexible concurrency. In the *Parallelism* pattern the need for concurrent task execution is recognized at design-time and is realized by means of parallel branching. The *Momentary Task Parallelization* and *Permanent Task Parallelization* allow the introduction of concurrent behavior based on parallel branching at the instance and type levels respectively.

**Pattern** FP-28  **PARALLELISM**

**Description** A process definition specified at design-time contains a construct that allows two independent tasks to execute simultaneously or in any order. This allows for more concurrency in the execution of tasks and avoids unnecessary waiting and dependencies. The need for flexible process initiation in this pattern is anticipated at design-time, and corresponds to the *flexibility by design* type.

**Examples**
- In high-performance computing large problems are often divided into small units which are then processed concurrently.
- Where the production of parts for car assembly do not dependent on each other, their production processes can be undertaken in parallel.

**Motivation** When a set of tasks is identified at design-time which can be executed independently, it may be necessary to specify that they can be executed concurrently or in any order. The *Parallelism* pattern allows two independent
tasks to execute concurrently by introducing two branches which can be undetaken in parallel. At run-time execution these tasks may be executed in either order or concurrently.

**Overview** Figure 64 illustrates the graphical notation for the **Parallelism** pattern. The top view shows the process definition where task C is executed after task B has completed. At design-time it is recognized that tasks B and C are independent of each other, and there is no need to enforce the execution ordering between them.

![Fig. 67. Parallelism pattern](image)

The bottom view shows the process definition after applying the pattern. In this process definition, tasks B and C have been put in parallel, i.e. they can be executed in either order or concurrently after the completion of task A.

**Context** The basic process engine (see Figure 68) supports the **Parallelism** pattern as follows. In order to specify the parallelism between two tasks B and C, which have been identified as independent tasks at design-time, and which need to be become enabled concurrently after task A has completed, the split and join connectors of associated synchronization tasks have to be set to AND. After executing task A by means of the **Execute task** transition, the **ns()** function analyzes the type of split of the executed task, and if it corresponds to the AND-split enables all outgoing subsequent arcs. Note that although theoretically these tasks can be executed concurrently, CPN Tools assumes the interleaving semantics allowing these tasks to be executed in any order but not simultaneously.

![Fig. 68. Basic process engine: support for Parallelism](image)
Implementation This pattern is supported by all of the systems analyzed. In Oracle BPEL PM, this pattern is supported by means of the `<flow>` construct with several concurrent branches. In GLIF and EON, a special branching block needs to be inserted with which parallel branches have to be associated. In AsbruView, the Parallel plan can be used for this purpose. YAWL enables parallel execution by associating the AND-split connector with a task. In PROforma, parallel execution can be enabled by incorporating the Decision construct.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that allows two or more tasks to be executed concurrently at run-time, based on the parallel execution paths specified for them in the process definition at design-time.

Pattern FP-29 MOMENTARY TASK PARALLELIZATION

Description During execution, at any time after the process initiation, a process definition associated with a particular process instance can be modified by parallelizing the execution of two subsequent independent tasks. This gives flexibility in initiating these tasks allowing them to execute in any order as well as concurrently. The change applies only to the given process instance and has no effect on other (current or future) process instances, thus it corresponds to the flexibility by momentary change type.

Examples – Due to change in planning, all activities are required to be completed earlier than usual, therefore in addition to the currently executing task an employee initiates another activity that has been planned for a later time.

– The workshop program consists of a set of presentation, conducted by a feedback session where students complete evaluation forms. To shorten the workshop program, students have been asked to fill in the evaluation form while attending one of the presentations.

Motivation When at run-time several independent tasks have been identified, who are prescribed by the process definition to execute sequentially, it should be possible to let them run concurrently or in any order. The Momentary Task Parallelization pattern allows for concurrent execution of two independent tasks by temporarily parallelizing them for a given process instance.

Overview Figure 64 illustrates the graphical notation for the Momentary Task Parallelization pattern. The top view shows two process instances populated from the same process definition. For one of the process instances the need in executing currently enabled task B concurrently with its subsequent task C is recognized.

The bottom view shows that after applying the pattern for the given process instance the process definition has been changed: tasks B and C have been parallelized, and an additional execution thread has been spawned off in order to enable task C.
Context  Figure 70 illustrates the basic process engine enhanced with the Momentary Task Parallelization pattern using the CPN formalism. The model of the engine is extended using the structure adopted by all patterns facilitating flexibility by momentary change (see p.35). The Momentary parallelization transition can be executed in order to parallelize a currently enabled (sequential) task with a subsequent task that is yet to be executed. The change_possible() function checks whether for the given process instance stored in the Running instance place there exist en enabled task that still has to be executed, and whether there exist two subsequently following each other tasks with no join and split connectors (these tasks should differ from the start and end tasks). The update_pi() function selects an arbitrary arc between two sequential tasks, deletes the sequential dependency between them, modifies the join and split connectors of the corresponding synchronization tasks and adds missing arcs from the synchronization tasks to the tasks recently put in parallel. After the change the process instance state is updated. If the currently enabled task has been involved in the change, then the updated state would contain two enabled tasks.

**Fig. 69.** Momentary Task Parallelization pattern

**Fig. 70.** Engine enhanced with the Momentary Task Parallelization pattern
Implementation Of all of the systems analyzed, only Declare supports this pattern. To allow another task to be executed concurrently with currently enabled task, the constrains associated with the task to be invoked have to be removed. Although ADEPT provides flexibility by change, it does not allow for the parallelization of sequential tasks.

Issues None identified.

Solutions N/A.

Evaluation Criteria Full support for this pattern is demonstrated by any offering that allows the order of tasks in the process definition associated with a given process instance to be modified in such a way that two independent sequential tasks are parallelized, and may be executed concurrently.

Pattern FP-30 PERMANENT TASK PARALLELIZATION

Description During execution, there is the possibility to permanently modify the process definition by transforming the sequential ordering of previously independent tasks into a concurrent structure, which allows these tasks to execute simultaneously or in any order. The parallelization of tasks is performed in the process definition at the type level on a permanent basis. The change performed affects future process instances directly, and existing process instance require old process definition to be migrated to the new process definition, thus it corresponds to the flexibility by permanent change type.

Examples
– The education program at a school has been revised allowing students to follow independent courses in parallel. The previous version of the program allowed only for incremental knowledge gathering that allowed the participation in a new course only after the previous course has been completed.
– It is common practice in hospitals to solve the health problem of one patient at a time. In order to decrease waiting times, patients in the same family are accepted together and treated concurrently.

Motivation When at run-time the possibility for concurrent execution of independent tasks is recognized there may arise the need to restructure the original process definition by parallelizing the execution order of these tasks. The Permanent Task Parallelization pattern allows two independent tasks to be parallelized on a permanent basis in order to provide flexibility in defining the execution order for these tasks.

Overview Figure 64 illustrates the graphical notation for the Permanent Task Parallelization pattern. The top view shows three distinct process instances populated from the same process definition. At run-time execution for the given process definition the need to relax the dependency between tasks B and C at the type level is recognized.

The bottom view shows that after applying the pattern tasks B and C have been parallelized. For the process instance that has passed these tasks this change is inconsequential. For the other two process instances, where task B has been
enabled, an additional thread of control has been spawned off in order to execute task C independently of task B. Both tasks have to complete in order for task D to become enabled.

**Context** Figure 72 illustrates the basic process engine enhanced with the **Permanent Task Parallelization** pattern using the CPN formalism. The model of the process engine is extended using the structure described earlier in the **Permanent Entry-Point Change** pattern on p.25.

Two sequential tasks can be put in parallel by the **Permanent parallelization** transition in the process definition only if conditions specified by the `change_possible()` function are satisfied. In particular, there must be an arc between two sequential tasks (with no splits or joins), and these tasks must be different from the start and end tasks. The `modify_m()` function selects an arbitrary arc between sequential tasks, identifies their synchronization tasks, sets the join and split connectors of the synchronization tasks to **AND**, deletes the selected arc, and adds two other arcs connecting the synchronization tasks with the tasks recently parallelized instead. After the change performed, there may be necessary to migrate existing process instances stored in the **Running instance** place. The `transfer_possible()` function checks whether there mismatches between old and new versions of the process definitions and whether the transfer is possible. The `migrate()` function performs the transfer of a process instance from the old to new process definition according to the specified strategy.

**Implementation** Of all of the systems analyzed, only ADEPT1, Declare and YAWL offer some support for this pattern. Declare allows previously sequential tasks to be concurrently enabled by removing the dependency constraints between them. Furthermore, it offers the possibility to transfer existing process instances to the new process definition. In order for two tasks to be put in parallel in ADEPT1, one of them have to be removed and inserted back in parallel with the other task. In YAWL, an exlet needs to be called in which given two tasks will be removed from the process model and inserted in the desired order by defining a new worklet.

**Issues** The same issues as identified for the **Permanent Entry-Point Change** pattern (see p. 25) also apply here.
Two tasks in a sequence are selected for parallelization. The connectors of the corresponding synchronization tasks are modified to AND-type.

**Fig. 72.** Engine enhanced with the Permanent Task Parallelization pattern

**Solutions** See solutions identified for the Permanent Entry-Point Change pattern.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the execution order of two sequential tasks to be parallelized, providing that existing process instances migrated from the old process definition to the new one.

### 3.9 Flexible repetition

The last group of patterns aims at flexibility when repeating the execution of a particular task variable number of times. Figure 73 illustrates the scope of patterns presented in this subsection and their relationship with different types of flexibility.

In some situations, when creating a process definition it may be foreseen that a particular task may need to be executed multiple times, but the number of times it will need to execute may vary or be unknown. The flexible repetition patterns describe different ways of achieving the behavior allowing a certain task to be executed multiple times. Of the four patterns identified, the *Iteration* pattern represents a generic form of a loop introduced in the process definition at design-time, which allows tasks residing in the body of the loop to be executed repeatedly until the loop termination condition is satisfied. There are different forms of iterations possible, such as those described by the control-flow repetition patterns described on p. ??, However, in the context of this research we abstract from different forms of the loops and consider its generic form. The *Redo* pattern applies to situations where after executing a particular task the decision to repeat it at some later point in time is taken. The *Momentary Loop Insertion* and *Permanent Loop Insertion* patterns represent variants of the *Iteration* pattern realized at run-time by modifying the process definition at the instance and type...
levels respectively. Note that this group contains no patterns corresponding to flexibility by underspecification. This has to do with the fact that the content of the task and its position are known and thus do not need to be represented as placeholders.

**Pattern** FP-31  **ITERATION**

**Description** A process definition specified at design-time contains a loop which allows the execution of a given task to be repeated. Typically, the decision to enable the task in the loop or continuation with the subsequent task depends on the evaluation of a data condition associated with this task. This gives flexibility in realizing execution sequences where the task embedded in the loop may need to be executed multiple times. The need for flexible task concurrency in this pattern is anticipated at design-time and corresponds to the *flexibility by design* type.

**Examples**
- For a driving license a candidates is required to take a theory exams. If the first attempt is unsuccessful, a candidate may repeat the test immediately without having to make an appointment for the examination again.
- The blood examination in a laboratory is repeated for sample until all required elements have been identified.

**Motivation** In situations where at design-time it is recognized that the number of times a certain task may need to be executed a variable number of times, it should be possible to specify that the execution of this task should can be repeated. The *Iteration* pattern provides flexibility in repeating the execution of a certain task by embedding it in the body of a loop, which can be iterated as many times as required.
Overview Figure 74 illustrates the graphical notation for the Iteration pattern. The top view shows the process definition in which each task executes only once. At design-time for this process definition the possibility of executing task C multiple times is recognized.

The bottom view shows that after applying the pattern task C has been modified in such a way that it can be executed multiple times. The number of times the given task will be executed in distinct process instances may vary.

Context The Iteration pattern is supported by the basic process engine in a similar way as for the Choice pattern (see p.41). The possibility of repeating the execution of a specific task can be incorporated into the process definition at design-time by inserting a choice construct forming a self-loop. For instance, in order to specify that after executing task C either another instance of task C needs to be executed or task D, both the join and split connectors of task C have to be set to XOR, and an additional arc (C,C) needs to be added to the set of arcs associated with the process model. In Figure ref:cpnIteration after executing task C by means of the Execute task transition, the ns() function calculates a new process instance state. When the value of the split connector associated with a task has been evaluated, for the XOR-type only one task out of several possible is selected non-deterministically. This means that the number of times the task will be iterated is not predefined, but is determined on a random basis.

Implementation In each of the systems analyzed, it is possible to achieve task iteration by embedding a task in a loop (for this, the same support as for the Choice pattern is required). In PROforma there exists an option to set an
iteration attribute associated with a specific task to a value or to an expression, thus allowing the number of task iterations to vary. In Oracle BPEL PM, a task needs to be inserted into a `<while>` loop, whose enabling condition defines whether a task residing in the loop will be iterated again. In FLOWer, iteration of a specific task can be achieved by inserting this task in the body of a sequential plan.

**Issues** In order to define whether a given task can be repeated, a data-based condition identifying whether to repeat the task has to be defined. The number of times the execution of the given task will be repeated depends on the moment of the data condition is evaluated.

**Solutions** There exist two variants of task iteration: a 'while' loop and a 'repeat-until' loop. In case of the 'while' loop the condition is evaluated before executing the task, whilst in case of the 'repeat-until' loop the data condition is evaluated after the task has been completed. For implementation details see the *Structured Loop* WCF pattern on p. ??.

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the execution of a specific task to be repeated multiple times based on a data condition included in the process definition at design-time.

**Pattern** FP-32 REDO

**Description** During execution, after process instance initiation, there is the possibility of deviating from the execution path prescribed by the process definition by repeating the execution of a task that has recently been completed. The need to execute a particular task more often than prescribed by the process definition is anticipated at run-time. This has no effect on the process definition, and corresponds to the flexibility by deviation type.

**Examples**
- Due to unexpected changes in travel plans, one of the reservations had to repeated in order to reset it to another date.
- The results of software testing showed some inconsistencies in the software design. This triggered revision of the design.

**Motivation** When at run-time it has been recognized that the execution of a particular task did not go as well as expected, it should be possible to execute the given task again. The *Redo* pattern offers flexibility in deviating from the execution path prescribed by the process definition by allowing the execution of the currently enabled tasks to be deferred until the execution of previously performed task has been repeated.

**Overview** Figure 76 illustrates the graphical notation for the *Redo* pattern. The top view shows the process instance for which the need to repeat the previously executed task C is recognized.

The bottom view shows that after applying the pattern the thread of control has been moved in the given process instance from task D to task C, thus allowing its execution to be repeated.
Context Figure 77 illustrates the basic process engine enhanced with the Redo pattern using the CPN formalism. The Redo transition allows to deviate from the normal prescribed execution order by executing a task that has already been completed (and the record about which is available in the Log place). The deviation_possible() function specifies that in order to redo a task, the list of executed tasks stored in the Log place should not be empty. The task whose execution needs to be redone is randomly selected by the dev_action() function. Note that the execution state of the given process instance and the process definition remain unchanged.

Implementation Of all systems analyzed only FLOWer and Oracle BPEL PM support this pattern. FLOWer offers a special operation to redo the execution of a previously executed task. In Oracle BPEL PM the only possible way to redo a task is to raise an exception or trigger an event upon which a previously executed task will become enabled again.

Issues When the execution of a specific task needs to be redone, the problem of overwriting data obtained during the first execution can occur.

Solutions Depending on whether there are data dependencies between the task that needs to be repeated and its subsequent task(s), it may be necessary to
roll back the execution of these tasks. In order to avoid overwriting of data, the values provided for data elements when executing the task to be redone can be marked as ‘unconfirmed’ rather than being discarded [28].

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows the execution of a previously executed task to be redone.

**Pattern** FP-33  **MOMENTARY LOOP INSERTION**

**Description** During execution, at any time after the process initiation, a process definition associated with a particular process instance can be temporarily modified by inserting a particular task in a loop. This allows the selected task to be executed more often that originally prescribed by the process definition. The need for momentary task repetition is recognized at run-time. This change applies only to the given process instance and has no effect on other (current or future) process instances, thus is of the *flexibility by momentary change* type.

**Examples**
- A patient who arrived to the emergency care center has been given an injection in the stomach for a dog bite. The injection is required to be repeated daily for 10 days.
- During process execution there arose a need to repeat a particular task over a specific period of time. The task planning section of the configuration window has no option to repeat a particular task. To make this possible the advanced settings of the configuration window have been changed by marking the “task repetition” field.

**Motivation** When at run-time a task has been identified which needs to be executed more often than prescribed by the process definition, it should be possible to adjust the process definition in order to allow the execution of the given task to be repeated multiple times. The **Momentary Loop Insertion** pattern allows a sequential task to be incorporated into a loop on a temporary basis in order for multiple subsequent executions of the given task to be achieved.

**Overview** Figure 78 illustrates the graphical notation for the **Momentary Loop Insertion** pattern. The top view shows two distinct process instances populated from the same process definition. At run-time execution, the need for iteration of the currently enabled task *C* is recognized in one of these process instances.

The bottom view shows that after applying the pattern the process definition associated with the given process instance has been modified by transforming task *C* into a repetitive task. Note that the other process instance remains unaffected.

**Context** Figure 79 illustrates the basic process engine enhanced with the **Momentary Loop Insertion** pattern using the CPN formalism. The model of engine adopts the structure for momentary changes described earlier in the **Momentary Entry-Point Change** pattern on p.35. In order to allow the execution of the currently enabled task to be repeated, it needs to be inserted in the loop by executing the **Momentary loop insertion** transition. The *change_possible()* function specifies that a loop can be inserted only if for a given process instance
exists an enabled task that still needs to be executed, and if the currently enabled task has no split or join connectors of **AND** type. The `update_pi()` function modifies the splits and join connectors of the selected task to **XOR**, and adds an arc representing a self-loop to the set of arcs associated with the given process instance. Note that the `update_pi()` function can be changed in such a way that not only a currently enabled task but any other task that still needs to be executed can be inserted in the loop.

**Implementation**  Of all of the systems analyzed, only Declare and YAWL support this pattern. Declare allows the execution of a task to be repeated by changing its cardinality constraint. In YAWL, an exect neededs to be called, which required a new worklet containing a loop to be defined. Once the defined worklet becomes available to other process instances.

**Issues**  Depending on the moment when the decision to repeat the execution of a particular task is made (i.e. before or immediately after it has been executed), different forms of a loop may need to be used.

**Solutions**  Having completed a specific task, the only possibility to repeat its execution is to insert it in the loop of ‘repeat-until’ type. The condition associated
with this loop will be evaluated at its conclusion, allowing the execution of this task to be repeated if the loop termination condition has not been satisfied. When the need to repeat the execution of a particular task is recognized prior to task execution, both the ‘repeat-until’ and ‘while’ forms of a loop are possible. In the ‘while’-loop, the decision about executing the task is taken before entering the loop (see the Structured Loop WCF pattern on p. ??).

**Evaluation Criteria** Full support for this pattern is demonstrated by any offering that allows a specific task in a process definition associated with a given process instance to be inserted into a loop in order to allow this task to be executed multiple times.

**Pattern** FP-34 PERMANENT LOOP INSERTION

**Description** At run-time, a process definition can be permanently modified by inserting a particular task in a loop. This allows the number of times the selected task will be executed to vary for different process instances. The need for task repetition is recognized at run-time. The change performed affects future process instances directly, and existing process instances may require migration from the old process definition to the new process definition, thus this pattern is of the flexibility by permanent change type.

**Examples**
- According to changes in the medicine prescription policy, patients do not need to visit a doctor each time the recipe prolongation is required, but can pick up their medicine in their preferred pharmacy. The repeat prescription request is processed automatically by a new system installed at the drugstore.
- As a result of a reorganization an electronic time-registration system has been introduced. Employees of this company involved in reorganization have to register their time working hours weekly using the system installed.

**Motivation** In situations where at run-time execution the need to execute a certain task multiple times is identified, it should be possible to modify the number of times the given task is allowed to execute. The Permanent Loop Insertion pattern allows the execution of a sequential task to be repeated multiple times by embedding this task into a loop.

**Overview** Figure 80 illustrates the graphical notation for the Permanent Loop Insertion pattern. The top view shows three distinct process instances populated from the same process definition. At run-time execution, the need to iterate task C is recognized at the type level.

The bottom view shows that after applying the pattern, the process definition has been modified by transforming task C into an iterative task and all existing process instances have been affected.

**Context** Figure 81 illustrates the basic process engine enhanced with the Permanent Loop Insertion pattern using the CPN formalism. The model of the engine has been extended using the structure adopted by all patterns facilitating the flexibility by permanent change (as described in the Permanent Entry-Point Change...
The execution of the **Permanent loop insertion** transition inserts a selected task in a loop. This transition can only be executed if the process description \( m \), stored in the **Process definition** place, contains more than two tasks and if there is a task that has no split and join connectors of **AND**-type. The \( \text{modify}_m() \) function selects an arbitrary task of the sequential type and modifies its split and join connectors to **XOR** value. The migration of existing process instances from old process definition \( m \) to new process definition \( \text{newm} \) is performed using the \( \text{migrate}() \) function.

**Implementation** Of all of the systems analyzed, only Declare offers support for this pattern by allowing the cardinality constraints associated with a specific task to be modified and migration of existing process instances to the new process definition to be performed.

**Issues** The same issues as identified for the **Permanent Entry-Point Change** pattern (see p. 25) also apply here.

**Solutions** See solutions identified for the **Permanent Entry-Point Change** pattern.
Evaluation Criteria  Full support for this pattern is demonstrated by any offering that allows the process definition to be modified by permanently inserting a specific task into a loop. In addition, there must be support for process instance migration.

3.10 Discussion

In this section, we give an overview of the flexibility patterns identified and define relationships between them. Figure 82 depicts the process flexibility matrix with 34 patterns identified. Some of the cells in this matrix are empty, indicating that the mapping of the process flexibility aspect to the specific flexibility type cannot be made or is not meaningful. In particular, there is no pattern providing flexibility by extension that can be mapped to flexibility by design. The reason for this is that all alternative execution paths that have been foreseen at design-time, can be included into the process definition using the Choice pattern. In the column for flexibility by underspecification, there are no patterns providing flexibility in task reordering, bypassing a task, task concurrency and task repetition. This has to with the fact that these operations apply to tasks which are known at design-time (i.e. there is no need in underspecifying their content).

Amongst the seven patterns characterized by flexibility by design, only the Alternative entry points and Choice patterns require the direct support in order for flexibility in process initiation and flexibility in execution path selection to be achieved. The other five patterns if not directly supported can be expressed on the basis of the Choice pattern. For instance, to realize the Parallelism pattern the condition for selecting one of several available branches defined for the Choice pattern needs to be relaxed, allowing all branches to be selected. The Foreseen By-pass pattern represents a choice between executing a specific task or not executing it at all, thus corresponds to the Choice pattern also. The Interleaving pattern can be also expressed using the Choice pattern by explicitly defining all possible ordering sequences, and selecting only one of them. The Iteration pattern can be seen as a special kind of choice where a decision to iterate a loop or to continue with subsequent activities needs to be taken.

Patterns related to flexibility by deviation have a corresponding mapping to each of the eight aspects of process flexibility identified. In this group of patterns, the direct support for the Task Skip pattern and the Task Invocation pattern are required in order for the flexibility features offered by other six patterns to be realized. In particular, the Entrance Skip pattern can be considered as a special variant of the Task Skip pattern. In order to start the execution of a process from a task other than the nominated one, the Task Skip pattern can be iteratively applied until the desired point in the process has been reached. The Termination Skip pattern can be also seen as a variant of the Task Skip pattern. The difference between them is that the effect of the Termination Skip pattern can be achieved by iteratively applying the Task Skip pattern, which moves the execution thread to a subsequent task rather than to the end of the process immediately. Having support for the Task Invocation pattern available, a task that is not currently enabled can be invoked in order to substitute the currently enabled task (i.e. the
Task Substitution pattern), or to repeat a previously executed task (i.e. the Redo pattern). Combinations of the Task Skip and Task Invocation pattern are required in order for the Swap pattern to be realized.

The group of patterns supporting flexibility by underspecification consists of four patterns. To realize these pattern support for placeholders is required. The completion of placeholders in each of these patterns can be done according to the late modeling and late binding strategies, introduced earlier on p. 4.

Patterns characterized by flexibility by momentary change and flexibility by permanent change can be mapped to each of the eight process flexibility aspects. Support for the Momentary Entry Change can be accomplished by applying the Momentary Task Elimination pattern until the desired entry-point is reached. The support for this pattern is also required in order for flexibility in process termination offered by the Momentary Exit Change pattern to be achieved. The Momentary Loop Insertion pattern requires support for the Momentary Choice Insertion pattern, whereas the Momentary Reordering can be achieved via the Momentary Task Insertion and the Momentary Task Elimination patterns. The same relations apply to the group of patterns supporting flexibility by permanent change.

This concludes the discussion of requirements for PAIS from the process flexibility perspective. We now move on to a comprehensive evaluation of flexibility pattern support in a selection of PAISs.

### 4 Tool evaluations

In this section, we show how the Process Flexibility Patterns can be used for assessing and comparing process modeling languages and enactment mechanisms employed by a wide range of PAISs. We concentrate on ADEPT1, FLOWer,
YAWL and Declare because they provide support for different types of flexibility. Furthermore, we analyze the capabilities of Oracle BPEL PM that is often used for the realization of interactive processes. Changes in one of the interacting processes may impact other processes involved in the interaction, thus it is interesting to see up to which extent this tool is capable to adapt to changes in the operating environment. In the health-care environment, unpredictable situations often occur, which require a very sensitive mechanism for adapting to changes. In such dynamic and unpredictable environment flexibility is an important issue.

To check the degree of flexibility offered by CIG modeling languages used in the health-care domain for encoding medical guidelines we evaluate PROforma, EON, GLIF, and Asbru\(^4\). We will define the degree of support offered by these languages by analyzing them against subsets of the flexibility patterns identified.

Table 2 summarizes the results of evaluating the selected PAISs. The full support (marked in Table 2 as “+”) for each of the patterns is defined on the basis of the evaluation criteria associated with them. Typically, there must be an explicit operation available that allows the effect of the flexibility approach identified by a pattern to be easily achieved. Partial support (marked in Table 2 as “+/-”) indicates that there is no direct way of realizing the desired behavior, however it can be achieved by executing a number of other steps. The absence of support (marked in Table 2 as “-”) indicates that either the desired behavior cannot be achieved or that workaround solution requires significant effort and is hard to realize.

In addition to the overview of supported patterns, Table 3 illustrates the support of patterns related to each of the five flexibility types.

Oracle BPEL PM offers good support for flexibility by design. By setting the createInstance attribute to “yes” different <receive> activities in a process model can serve as alternative start tasks. The ability to put the <Terminate> activity in different places in a process model allows the process to be terminated once any of these activities has been executed. The support for flexible selection, flexible reordering and flexible elimination can be accomplished in Oracle BPEL PM using the <switch> construct, where alternative tasks, sequences of tasks or empty tasks can be defined in each of the branches. The support for flexible concurrency is provided by the <flow> activity, whilst flexible repetition can be obtained by using the <while>-loop construct. The ability to throw an exception in the scope of a specific task, allows a compensation action to be performed. In this way the execution of one task can be substituted with another, a previously executed task can be redone, a process instance can be terminated or another task or process can be invoked, thus providing support for four deviation patterns. Al-

\(^4\) Of these four modeling languages only PROforma can be evaluated from the perspective of five flexibility types identified: the Tallis tool is based on PROforma semantics and provides functionality for process execution. The other three languages are analyzed only from the perspective of flexibility by design. The run-time behavior associated with flexibility by deviation, flexibility by underspecification, flexibility by momentary and permanent change cannot be assessed due to non-availability of enactment engines for these languages.
Table 2. Support for the Process Flexibility Patterns in (1) Oracle BPEL PM, (2) ADEPT1, (3) FLOWer, (4) Declare, (5) YAWL, (6) PROforma, (7) EON, (8) GLIF, and (9) Asbru

<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flexible Initiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-1 Alt. Entry-Points</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-2 Entrance Skip</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-3 Undefined Entry</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-4 Mom. Entry Change</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-5 Perm. Entry Change</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexible Termination</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-6 Alt. Exit-Points</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-7 Termination Skip</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-8 Undefined Exit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-9 Mom. Exit Change</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-10 Perm. Exit Change</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexible Selection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-11 Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>FP-12 Task Substitution</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-13 Late Selection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-14 Mom. Choice Insertion</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-15 Perm. Choice Insertion</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexible Reordering</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-16 Interleaving</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>FP-17 Swap</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-18 Mom. Reordering</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-19 Perm. Reordering</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexible Elimination</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-20 Foreseen By-path</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>FP-21 Task Skip</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-22 Mom. Task Elimination</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-23 Perm. Task Elimination</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexible Extension</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-24 Task Invocation</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-25 Late Creation</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-26 Mom. Task Insertion</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-27 Perm. Task Insertion</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexible Concurrency</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-28 Parallelism</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>FP-29 Mom. Task Parallelization</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-30 Perm. Task Parallelization</td>
<td>+</td>
<td>/</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Flexible Repetition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-31 Iteration</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>FP-32 Redo</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FP-33 Mom. Loop Insertion</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>FP-34 Perm. Loop Insertion</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 3. Support for the Process Flexibility Patterns in (1) Oracle BPEL PM, (2) ADEPT1, (3) FLOWer, (4) Declare, (5) YAWL, (6) PROforma, (7) EON, (8) GLIF, and (9) Asbru

<table>
<thead>
<tr>
<th>ID</th>
<th>Pattern name</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Flexibility by Design</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-1</td>
<td>Alt. Entry-Points</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>FP-6</td>
<td>Alt. Exit-Points</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>FP-11</td>
<td>Choice</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-16</td>
<td>Interleaving</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-20</td>
<td>Foreseen By-path</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-28</td>
<td>Parallelism</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>FP-31</td>
<td>Iteration</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td><strong>Flexibility by Deviation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-2</td>
<td>Entrance Skip</td>
<td>-</td>
<td>-</td>
<td>+/-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-7</td>
<td>Termination Skip</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>FP-12</td>
<td>Task Substitution</td>
<td>+</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>FP-17</td>
<td>Swap</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>FP-21</td>
<td>Task Skip</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>FP-24</td>
<td>Task Invocation</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>FP-32</td>
<td>Redo</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td><strong>Flexibility by Underspecification</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-3</td>
<td>Undefined Entry</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-8</td>
<td>Undefined Exit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-15</td>
<td>Late Selection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-25</td>
<td>Late Creation</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td><strong>Flexibility by Mom.Change</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-4</td>
<td>Mom. Exit Change</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-9</td>
<td>Mom. Exit Change</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-14</td>
<td>Mom. Choice Insertion</td>
<td>-</td>
<td>+/-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-18</td>
<td>Mom. Reordering</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-22</td>
<td>Mom. Task Elimination</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-26</td>
<td>Mom. Task Insertion</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-29</td>
<td>Mom. Task Parallelization</td>
<td>-</td>
<td>+/-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-33</td>
<td>Mom. Loop Insertion</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-5</td>
<td>Perm. Exit Change</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td><strong>Flexibility by Perm.Change</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP-10</td>
<td>Perm. Exit Change</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-15</td>
<td>Perm. Choice Insertion</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-19</td>
<td>Perm. Reordering</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-23</td>
<td>Perm. Task Elimination</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-27</td>
<td>Perm. Task Insertion</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-30</td>
<td>Perm. Task Parallelization</td>
<td>-</td>
<td>+/-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FP-34</td>
<td>Perm. Loop Insertion</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

98
though Oracle BPEL PM does not support the concept of a placeholder, it allows the definition of credentials of a process that needs to be invoked to be postponed until run-time. These can be set dynamically based on the input provided, thus corresponding to the Late Creation pattern. This system provides no means of support for flexibility by momentary change or flexibility by permanent change, because the process definition cannot be modified at an instance nor at a type level.

ADEPT1 is characterized by its support for processes of the structured nature with a single entry and a single exit point. This feature is used for checking the soundness of models when defining process models and migrating process instances from an old process definition to a new process definition, obtained as the result of change. This systems scores well in supporting flexibility by design, flexibility by momentary change and flexibility by permanent change. ADEPT1 allows the process definition to be modified at run-time in order to facilitate adaptation to changes in the operating environment. The basic operations that are directly supported are adding and deleting a task, these correspond to direct support for flexibility in process extension and flexibility in task elimination. Patterns such as Momentary Entry Change, Momentary Exit Change, Momentary Choice Insertion, Momentary Reordering and Momentary Parallelization cannot be realized directly, however the effect of the flexibility mechanisms they facilitate can be achieved by applying the insert and delete task operations. The desired start task in a process can be achieved by deleting tasks in the beginning of the process one-by-one. Similar steps have to be performed in order for a process to be terminated earlier. To reorder two tasks, they have to be removed and re-inserted in the correct order. The same holds for task parallelization. It is however not possible to insert a task in a loop, therefore the only patterns not supported in the group of momentary and permanent change patterns are the Momentary Loop Insertion and Permanent Loop Insertion. A remarkable feature supported by ADEPT1 is migration of process instances to a new process definition. This explains why patterns in the group of permanent change are supported as well as those in the group of momentary change.

FLOWer is a case-handling system offering a lot of flexibility in deviating from the execution order prescribed by the process definition. Similar to ADEPT1, FLOWer allows only a single entry to and single exit from the process model. This explains, why the Alternative Entry Points and the Alternative Exit Points patterns are not supported. The availability of various types of plans (i.e. sequential, dynamic) provide good support for the other six patterns in the flexibility by design group. The majority of patterns facilitating flexibility by deviation are supported by FLOWer directly. It is possible to start the execution of a non-enabled task directly, skip a task, invoke a non-enabled task concurrently with a currently enabled task or rollback and repeat a previously executed task. There is the possibility to perform a termination skip by executing a skip operation at the level of a root plan. The only two deviation patterns that are supported indirectly are the Entrance Skip pattern and the Task Substitution pattern. In order to skip all tasks preceding a desired start task, a skip operation has to be applied for
each of the tasks individually. In order to execute a task different to the currently enabled one, the task needs to be invoked and the skip operation needs to be applied to the originally enabled task. FLOWer offers no support for flexibility by underspecification, flexibility by momentary change and flexibility by permanent change.

Declare is a system based on the declarative approach to process specification. It offers full support for flexibility by design-time, flexibility by momentary change and flexibility by permanent change. Furthermore, it supports the majority of patterns facilitating flexibility by deviation. From this group, the Task Substitution pattern and the Redo pattern are not supported. Such a broad range of patterns are supported by means of inserting and deleting tasks and/or constraints defining the execution order of these tasks. Support for process instance migration allows process instances to be transferred from an old process definition to a newly defined one, thus facilitating flexibility by momentary and permanent change.

The CIG modeling languages analyzed offer support for almost all patterns related to flexibility by design. Of all these languages, only PROforma allows a process to be started from and ended with alternative tasks (this is possible because of the combined imperative and declarative approaches). In GLIF, various start tasks can be defined allowing a patient to enter a process at any suitable stage. The rest of the patterns facilitating flexibility by design can be directly expressed in each of the languages examined.

Based on the results of the analysis, we can conclude that the intent of the system/language analyzed influences the types of flexibility supported. Systems based on declarative approaches (i.e. Declare) score well from the perspective of flexibility by deviation, because various types of behaviors may be achieved by simply adding, removing or modifying constraints associated with tasks. Case-handling systems such as FLOWer offer an explicit set of deviation operations in order for the desired behavior to be easily achieved at run-time, without requiring an underlying process definition to be modified. The support for flexibility by momentary and permanent change is facilitated in systems where the possibility for reacting on non-foreseen events has been incorporated in the system design (i.e. migration mechanisms of ADEPT1, YAWL and Declare). The ability to leave a process definition underspecified, which can be completed at run-time is clearly an aspect of process flexibility that is less widely supported. The fact that Oracle BPEL PM and YAWL’s worklets extension accommodate service-based approaches illustrates the applicability of concepts related to flexibility by underspecification in the service-oriented domain. From the perspective of the health-care domain, where more flexibility is required due to unpredictable nature of events, CIG modeling languages score slightly better than traditional workflow systems. Namely in their the ability to initiate a process at an arbitrary point and terminate it “on spot” are typical operations in this domain.

Tables 4 and 5 summarize the results of evaluating the offerings against the process flexibility groups and process flexibility types respectively. In these tables, the rating (+++)) is given to an offering that support more than half of the pat-
terns per a given group or flexibility type; the rating (+++) is given to an offering supporting at least half of the patterns; the rating (+) is given to an offering if it supports one pattern; the rating (+/-) is given to an offering that offers no direct support for patterns; finally, the rating (-) indicates that no patterns are supported by the offering.

Table 4. Support for the Process Flexibility groups in (1) Oracle BPEL PM, (2) ADEPT1, (3) FLOWer, (4) Declare, (5) YAWL, (6) PROforma, (7) EON, (8) GLIF, and (9) Asbru

<table>
<thead>
<tr>
<th>Pattern group</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible Initiation</td>
<td>+</td>
<td>+/-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Flexible Termination</td>
<td>++</td>
<td>+/−</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Flexible Selection</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Flexible Reordering</td>
<td>++</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Flexible Elimination</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Flexible Extension</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Flexible Concurrency</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Flexible Repetition</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Support for the Process Flexibility types in (1) Oracle BPEL PM, (2) ADEPT1, (3) FLOWer, (4) Declare, (5) YAWL, (6) PROforma, (7) EON, (8) GLIF, and (9) Asbru

<table>
<thead>
<tr>
<th>Flexibility by</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>+++</td>
<td>+++</td>
<td>+++</td>
<td>+++</td>
<td>+++</td>
<td>+++</td>
<td>+++</td>
<td>+++</td>
<td>+++</td>
</tr>
<tr>
<td>Deviation</td>
<td>++</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Underspecification</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Mom. Change</td>
<td>-</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Perm. Change</td>
<td>-</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

5 Related work

The need for process flexibility has been acknowledged in the workflow and process technology communities as a critical quality of effective business processes in order for organizations to adapt to changing business circumstances [35, 52, 48]. The notion of flexibility is often viewed in terms of the ability of an organization’s processes and supporting technologies to adapt to these changes [60, 19]. An alternate view advanced by Regev and Wegmann [46] is that flexibility should be considered from the opposite perspective, i.e. in terms of what stays the same not what changes. Indeed, a process can only be considered to be flexible if it is possible to change it without needing to replace it completely [47]. Hence flexibility is effectively a balance between change and stability that ensures that the identity of the process is retained [46, 49].
A series of proposals have been made to classify process flexibility based on factors which motivate it and the ways in which it can be achieved within business processes. Snowdon et al. [60] identify three causal factors: type flexibility (arising from the diversity of information being handled), volume flexibility (arising from the amount of information types) and structural flexibility (arising from the need to operate in different ways. Soffer [61] differentiates between short-term flexibility, which involves a temporary deviation from the standard way of working, and long-term flexibility, which involves changes to the usual way of working. Kumar and Narasipuram [40] distinguish pre-designed flexibility which is anticipated by the designer and forms part of the process definition and just-in-time responsive flexibility which requires an “intelligent process manager” to deal with the variation as it arises at runtime. Carlsen et al. [13] identify a series of desirable flexibility features for workflow systems based on an examination of five workflow offerings using a quality evaluation framework.

Heinl et al. [35] propose a classification scheme with distinct approaches – flexibility by selection, where a variety of alternative execution paths are designed into a process, and flexibility by adaption, where a workflow is “adapted” (i.e., modified) to meet with the new requirements. Two distinct approaches to achieving each of these approaches are recognized: flexibility by selection can be implemented either by advanced modeling (before execution time) or late modeling (during execution time) whereas flexibility by adaption can be handled either by type adaption (where the process definition is changed but individual process instances currently running are unaffected) or instance adaption where selected (or all) process instances are changed to meet with new operational requirements. Van der Aalst and Jablonski [6] adopt a similar strategy for supporting flexibility. Moreover they propose a scheme for classifying workflow changes in detail based on six criteria: (1) reason for change, (2) effect of change, (3) perspectives affected, (4) kind of change, (5) when are changes allowed and (6) what to do with existing process instances. In [58] Schonenberg et al. presented a preliminary version of the taxonomy of process flexibility, covering four types of flexibility: design, deviation, underspecification and change. This work has been elaborated in a more detail in this chapter, and for each flexibility types a set of process flexibility patterns have been defined that allow evaluation of offerings in a more precise way.

Regev et al. [47] made an initial attempt to define a taxonomy of concepts that are relevant to business process flexibility. This taxonomy has three orthogonal dimensions: the abstraction level of the change, the subject of the change and the properties of the change. Whilst it incorporates elements of the research initiatives described above, it is not comprehensive in form and does not describes the relationships that exist between these concepts or link them to possible realization approaches.

There are a variety of approaches to incorporating flexibility within a design-time process definition. Traditional process design methods [52, 41, 12] have centered on the separation of business logic from the actual application processing and utilizing constructs such as hierarchy, conditional elements and business rules within the process definition to explicitly cater for various execution scenarios that
might be encountered. Whilst effective, these strategies require that all possible situations be captured a priori at design-time, an assumption that proves to be unrealistic in practice [35]. The use of exceptions [56, 64, 23] provides one means of handling expected but infrequently occurring processing errors without requiring their explicit inclusion in the process definition. Various techniques to implementing exception handling strategies in workflow systems have been demonstrated by offerings including WAMO [22], ConTracts [53], Exotica [11], OPERA [31, 32], TREX [63] and WIDE [14].

Another approach that has been investigated for embedding flexible constructs in business processes involve the augmentation of control-flow routing constructs operators based on fuzzy logic [8]. Indeed one area that offers significant opportunity for increasing the potential flexibility of a business process is the replacement of the strict graph-based structures that are generally used to describe control-flow dependencies between the tasks in a process with other means of describing these dependencies. ConDec [?] is a declarative language that specifies control-flow dependencies using linear temporal logic expressions. Other research initiatives in this area have investigated a variety of other means of defining control-flow including the use of process grammars to specify dependencies between tasks and documents (i.e., data elements) in a process [27], the introduction of the notion of “anticipation” [29] which allows the execution of sequential tasks to overlap at the discretion of workflow users where there are not specific data dependencies between them, the inclusion of flexible elements in process definitions that describe alternate execution options, alternate task orderings and optional tasks [39] and basing control-flow on rule-based invariants that must hold during process execution [46] or constraints based on task pre and postconditions [66] that determine when individual tasks can start and complete.

The potential for increasing process flexibility by allowing deviations from the specified process definition at runtime is supported in PROSYT [18] which allows a deviation policy to be specified for a process, identifying which forms of deviation are tolerated, together with a consistency handling policy, which ensures any allowed deviations do not impact the overall correctness of the system. In the context of the WASA system, Weske [18] nominates three user-initiated operations – SkipActivity, StopActivity and RepeatActivity – that allow for deviations from normal workflow execution.

Several approaches have been proposed that support the underspecification of processes thus allowing for greater flexibility in the actual tasks initiated at runtime. Noll [44] advocates the use of low fidelity models which specify the major tasks and main sequence in a process, but leave the actual sequence of execution at the discretion of the user. This essentially corresponds to a more general notion of the case handling paradigm [7] as it allows distinct tasks in a given process instance to be undertaken by differing users. In a similar vein, Herrmann and Loser [36] advocate the inclusion of “vagueness” in socio-technical process definitions allowing concepts such as arc conditions and task ordering to be deliberately omitted from models and also supporting the inclusion of specific modelling constructs to identify aspects of the model that are incomplete or unspecified. Van der Aalst
advances the notion of generic process definitions \cite{1, 2} which allow placeholders elements (termed generic processes) to be specified in models which correspond to fragments of the overall process whose actual composition is determined at runtime. Mangan and Sadiq \cite{42} propose an analogous scheme where a process is partially specified as a set of fragments and the actual format of the process definition undertaken for a given instance of the process is deferred to runtime at the discretion of individual users. In a subsequent paper Sadiq et al. \cite{57} describe a flexible workflow modelling language which incorporates “pockets of flexibility” which denote regions of the process whose actual content is determined at runtime based on workflow fragments (tasks or sub-processes) and composition rules that are associated with them. The OPENflow system \cite{33} is an example of an actual system that supports this approach to process flexibility. The issue of managing dynamic change to executing processes has been widely researched in the fields of adaptive and evolutionary workflow \cite{38, 24, 15, 69, 17, 59}. A number of significant research prototypes have been developed in this area including ADEPTflex \cite{50}, ADOME \cite{16}, CBRFlow \cite{68}, DYNAMITE \cite{34}, WASA2 \cite{69}, Declare \cite{58} and YAWL worklets \cite{9}.

When during process execution, the process needs to be changed on an ad-hoc basis, such that all existing process instances are migrated, various kind of errors can occur, e.g. introduction of duplicate tasks, deadlocks or lifelocks. This problem is known as ‘dynamic change bug’. One of the first steps towards resolving this problem have been made by Ellis et al. \cite{25}, who introduced a mathematical formalism for modeling and analysing dynamic changes in workflow. This work was not complete, and was addressed by many investigations. In \cite{5} van der Aalst et al. propose to use inheritance-preserving transformation rules in order to prevent the occurrence of the dynamic change bug. The authors define a set of transformation rules that can be used to restrict changes in process definitions such that new process model inherits desirable properties of the old workflow process. In \cite{4} van der Aalst describes an approach for calculating a safe change region. If a process instance is in such a change region, the transfer of the process instance from the old process definition to the new one is postponed. A comprehensive evaluation of various approaches (both conceptual and implementation-based) to managing dynamic changes to workflow processes is presented by Rinderle et al. in \cite{54}. As a means of comparing various approaches to process change, Weber et al \cite{67} have proposed a set of 17 change patterns and 6 change support features. The majority of these patterns can be mapped on process flexibility patterns presented in this paper.

Reijers et al \cite{51} present yet another view on flexibility. The authors categorize exceptions as expected and unexpected. Expected exceptions are addressable if they are technologically solvable. They authors describe how each type of addressable exception should be handled, i.e., either by executing the main workflow in a specific way; or a by executing a separate workflow. To increase the flexibility of WFMSs the authors propose extending the functionality of WFMS with case variables and preconditions, direct access to the workflow execution status and jump facilities (this suggestion has been implemented by Staffware).
De Moor and Jeusfeld [20] propose a legitimate user-driven approach which enhances the acceptability of workflow changes in the context of virtual communities. Chun and Atluri in [62] propose an approach for workflows in the context of eGovernment to adapt to run-time changes. The authors classify different types of run-time changes and propose an ontology-based framework for dynamic Workflow Change Management system which adopts the changes based on profile change, exceptions and rule change using migration rules. Van der Aalst [3] discusses the problem of case migration caused by workflow change leading to task duplication, skipping of tasks, deadlocks, and livelocks. To address this problem, the author proposes an approach for calculating a safe change region. Several approaches for adaptive process management have been proposed in [54]. Adaptive processes enable users to evolve process definitions, so that they meet new requirements [30].

In several contemporary offerings, the process flexibility aspect has played a significant role in defining the manner in which these offerings were developed. Staffware Process Orchestrator [26] has been developed with an approach to assign process components (that are not known at design-time) dynamically at run-time, when sufficient amount of information becomes available. This feature is especially useful in service-oriented environment, where the selection of a service needs to be based on an event or a response from an external system. InConcert [21] allows the task structure of a process instance to be modified dynamically by a user with appropriate authorization using operations such as adding and deleting tasks, adding and deleting dependencies, and specifying task properties such as durations and conditional execution.

6 Conclusions

In this paper, we have focused on the issues of process flexibility and defined a taxonomy consisting of five flexibility types: flexibility by design, flexibility by deviation, flexibility by underspecification, flexibility by momentary change and flexibility by permanent change. This classification shows that there are different approaches to facilitating flexibility for a user in selecting a desired execution outcome. The need for flexibility may be recognized during design-time when creating a process definition or at run-time. However, the actual decision regarding which execution path will be taken is made at run-time execution depending on the constraints posed by an external environment. Since it is very hard to foresee all possible behaviors in advance, contemporary PAISs must be able to adapt to continuously-changing requirements. In terms of the functionality offered by PAISs, it is important to clearly identify which type of flexibility the system supports:

– Does a system allow for enactment of incomplete process definitions?
– Is it possible to deviate from the steps prescribed by the process definition?
– Is it possible to change the behavior of specific process instance?
– Is it possible to redesign the process definition in such a way that the changes made directly become available in current and future process instances?
Since the taxonomy of process flexibility does not give sufficient details about how each of the flexibility types can be operationalized, we defined a set of process flexibility patterns, each addressing a specific aspect of flexibility and corresponding to one of the five flexibility types. The 34 patterns reason about flexibility in process initiation, flexibility in process termination, flexibility in execution path selection, flexibility in task reordering, flexibility in task elimination, flexibility in process extension, flexibility in task concurrency and flexibility in task repetition.

The Process Flexibility Patterns identified have a wide range of potential application areas including:

- Understanding of process flexibility requirements;
- Precise, systematic description of flexibility aspects;
- Benchmarking PAISs; and
- Common process flexibility vocabulary.

Process flexibility is a topic that has been attracting a lot of attention both from research and industry sites. A clear understanding of the requirements for process flexibility is important for process designers who need to deal with non-typical behavior whilst designing process definitions; for architects and developers of information systems to define the degree of flexibility a system needs to offer and how these needs will be realized; and for users who need to have sufficient knowledge of the system and its capabilities from the process flexibility perspective in order to make correct steps for adapting the process to changes in an operating environment.

Process Flexibility Patterns provide a precise and systematic description of wide variety of flexibility aspects. We defined a map of process flexibility requirements, where for each of eight process flexibility aspects a set of patterns corresponding to a specific flexibility type are defined. In order to distinguish patterns between each other, for each of them we defined a graphical notation. The purpose of this notation is to show the effect of a pattern by visualizing the process definition on the type/instance level before and after applying the pattern. In order to avoid ambiguous interpretation, for each of the patterns identified we designed a CPN diagram illustrating the behavior of a pattern in the context of a basic process engine.

As we showed in Section 4, patterns identified can be used as a means of benchmarking the degree of process flexibility support in PAISs. Interestingly to note that none of the offerings analyzed provide support for all flexibility types. Patterns can be used as a tool for selecting a system supporting desired types of flexibility.

Finally, the systematic approach used for identifying Process Flexibility Patterns helps to structure knowledge accumulated in the domain. Concepts defined in the taxonomy of process flexibility, i.e. flexibility types, flexibility groups and pattern names can enrich the communication vocabulary, and be used for improving communication between process designers, system developers, and end users.
The scope of the topic addressed in this paper is limited to the process control-flow flexibility. In order to get a holistic understanding of requirement in process flexibility, other relevant perspectives (such as data and resources) have to be investigated. Nevertheless, information presented in this paper can be seen as a first step towards a universal flexibility model (for example, this could be done using the meta-or ontology-format). There are many other possible applications of flexibility patterns. For example, issues related to process mining of patterns applied during run-time execution can be investigated.
References


