A declarative meta modeling approach to define process migration constraints

Bram Leemburg, s1398334
Master thesis Software Engineering & Distributed Systems

University of Groningen
Supervisor: prof. dr. ir. Marco Aiello
Second supervisor: prof dr. Gerard Renardel de Lavalette

August 24, 2011

Contents

Abstract **3**

1 Introduction **4**
1.1 Overview of document . . . 6

2 Background **7**
2.1 Business Processes . . . 7
2.1.1 Business Processes in context . . . 7
2.1.2 Processes defined . . . 8
2.2 Meta processes . . . 10
2.2.1 Process migration defined . . . 11
2.3 IT-support for Business Processes . . . 12
2.3.1 Languages for Business Processes . . . 13
2.3.2 Business Process or Workflow? . . . 13
2.3.3 Perspectives of a Business Process; control-flow, data and resources . . . 14
2.4 Terminology . . . 14
2.5 Business Process Management . . . 16
2.5.1 Practices in Business Process Management . . . 16
2.5.2 The Business Process Management life cycle . . . 17
2.5.3 Web services as implementation of Business Processes . . . 17

3 Related work **18**
3.1 Variability Management . . . 18
3.1.1 A taxonomy of Flexibility . . . 18
3.2 Modeling formalisms . . . 19
3.2.1 Π-calculus . . . 19
3.2.2 Petri nets . . . 20
3.2.3 Declarative modeling . . . 21
3.2.4 Concluding remarks on modeling formalisms . . . 22
3.3 Dynamic change and process migration . . . 23
3.3.1 What is change: specification criteria . . . 23
3.3.2 How is change handled: different strategies . . . 24
3.3.3 Considering business context requirements: Semantic constraints . . . 25
3.4 Meta modeling approaches: change specification . . . 26
3.4.1 Change operation primitives . . . 26
3.4.2 Conditional migration rules . . . 27
3.4.3 Dynamic constraint model application . . . 27
3.4.4 Comparison of approaches . . . 27
3.5 Process Aware Information Systems . . . 28
3.5.1 The eFLOW framework . . . 28
3.5.2 Current migration support in DECLARE . . . 29
3.5.3 Advanced migration strategies and optimizations in ADEPT2 framework . . . 30

4 Problem statement **31**
4.1 Aim: A configurable solution for process migration . . . 31
4.2 Reflection on the state-of-the-art of frameworks . . . 32
4.3 The need for dynamic process change . . . 32
4.3.1 Some example stories from different domains . . . 33
4.3.2 Domain dependent requirements at the meta process level . . . 33
4.4 System context . . . 35
4.4.1 Actors . . . 36

4.4.2 Involved systems . . . 36
4.4.3 Use Case description: process migration configuration . . . 37
4.5 Research questions . . . 37
4.6 Research methodology . . . 38
4.6.1 TPL as a declarative process model . . . 38
4.6.2 Model validation . . . 39
4.6.3 Process quantification . . . 39

5 Realization **40**
5.1 Envisioned framework architecture . . . 40
5.1.1 Dynamic process enactment and service composition . . . 40
5.1.2 System decomposition . . . 41
5.2 Operation of the TPL Reasoning System . . . 43
5.2.1 External interfaces . . . 43
5.2.2 Internal operation . . . 43
5.3 The task of validating a TPL model . . . 44
5.4 Traces: a detailed description of run-time execution . . . 45
5.4.1 Modeling run-time execution . . . 45
5.4.2 Creation of process traces . . . 45
5.4.3 Traces related to execution paths . . . 46
5.5 Validating TPL-expressions . . . 48
5.5.1 Validation approach: step-by-step evaluation of partial expressions . . . 48
5.5.2 Validation logic implementation . . . 51
5.5.3 Recursive semantic evaluation of expressions . . . 52
5.5.4 Semantic evaluation challenges . . . 54
5.6 Towards configurable process meta models . . . 56
5.6.1 Meta process level trace generation . . . 57
5.6.2 TPL-meta model constructs . . . 57

6 Evaluation and results **60**
6.1 Results of trace validation with TPL . . . 60
6.1.1 A readable trace notation . . . 60
6.1.2 Process level metrics . . . 60
6.1.3 Process simulation results . . . 61
6.2 TPL meta model configuration and evaluation . . . 62
6.2.1 Building meta models with TPL . . . 62
6.2.2 Temporary process incompliancy . . . 64
6.2.3 Meta model level metrics . . . 65
6.2.4 Meta model test data collection . . . 66
6.2.5 Meta model test results . . . 66

7 Discussion and future work **68**
7.1 Discussion of TPL's expressiveness for meta models . . . 68
7.1.1 TPL meta models in perspective of migration strategies . . . 68
7.1.2 Comparison with existing frameworks on configurability . . . 69
7.1.3 Separation of concerns between models and meta models . . . 69
7.1.4 Suggestions for alternative notations in TPL . . . 69
7.1.5 Using TPL to express more constraint types . . . 70
7.2 Outlook and unresolved issues . . . 71
7.2.1 Ad hoc TPL process model creation . . . 71
7.2.2 Integrated process development environment . . . 71
7.2.3 Including data and resource perspectives . . . 72
7.2.4 Hybrid approach: combining the declarative and imperative . . . 72
7.2.5 TPL model satisfaction and planning . . . 72

8 Conclusion **73**

References **75**

Appendices **76**

A Prolog implementation **77**

Abstract

Business Process Management (BPM) aims to model the workflow of business operations providing means for management to monitor, control and improve operations. This practice has been of particular interest to computer science, as process enactment and analysis can benefit immensely from automation. Where process design and enactment used to be distinct phases, information systems now lack support for the emerging emerging life cycle of continuous process redesign. This life cycle requires that process changes can be handled dynamically. Process migration is used to describe what needs to be done when a new version of a process is defined. Various strategies describe how changes can be adopted by process instances. Instead of the usual division into migration strategies we propose a language to describes migration policies, based on a declarative process modeling approach. We propose a solution implementing Temporal Process Logic, a logic formalism for Business Processes. The logic is applied on the process level and can perform run-time validation of these models, providing a highly dynamic and flexible architecture for process enactment. Furthermore we show how TPL can be applied to the meta process level as well, describing in particular instantiation, completion, compliance and transfer of executions in the system at run-time. Behavior of the models and meta models is characterized by metrics based on the notions of execution graphs, compliance, propagation and variability often used within the field. Our solution is able to differentiate various cases for basic migration strategies among *transfer*, *proceed* and *deviation*.

Chapter 1

Introduction

In current businesses, from commercial enterprises to governmental organizations, a large part of the actual business is performed by information systems. The Web has transformed the way business is conducted and allows information systems to actively participate in the processes [1]. Customers access services offered by an enterprise via the web and business partners negotiate and trade via web services. Also internally tasks are either performed automatically by information systems or are done manually but the output is entered into the information systems. By implementing each step of the process as a Web service, both automated activities as human interactions can be treated the same. The challenge and added value of these web services lies in the way the services can be used in composition and work together to realize Business Processes in a dynamic way [2]. Web service protocols exist for orchestration and choreography of services. Existing techniques allow Web services to be composed independent of their location and implementation details. Orchestration describes how Web services interact and defines the business logic and execution order of the interactions [3]. Choreography typically describes describes communication between multiple parties, such as customers, suppliers, and business partners. Choreography and orchestration provide the standards for enterprises to offer services on the end-user level and allow inter and intra business operations.

Enterprises attempt to optimize their efficiency by analyzing and optimizing their Business Processes [4]. The goal is to run a highly successful business and to supply customers in their needs. The practice of Business Process Management (BPM) is involved with the (re)design, execution and enforcement of Business Processes. BPM is illustrated in figure 1.1. As mentioned in the previous section the information systems involved in modern businesses are not just tools for data storage and recall and communication, but actively take part in the business by means of web services. This means that the atomic parts that make up a Business Process in a Web service based system are the web services. Information systems are given the responsible for execution and enforcement of the Business Processes involved with the goal to reach higher efficiency due to high automation and combination of different systems. For this purpose the Business Processes have to be modeled within the information system. In other words, the system is made aware of the Business Processes involved, also referred to as a Process Aware Information Systems (PAIS). Currently IT provides various techniques and systems to support Business Process Management. Most work has been focused on development of different languages and notations to denote Business Processes and supporting composition of Web services according to a process definition.

Problems however are that the current Business Process Management Systems are too rigid, unable to make necessary exceptions and change the process as the market changes or new legislation is made [4]. Another problem is that implemented systems lack formalization of their processes, resulting in vague semantics of their operations, also when it comes to adopting process changes. The result often is lack of support for description of process, lack of support for changes and workarounds used in practice at the operational level [4, 5]. Although features are being added to systems to support Business Processes in more flexible ways, the lack is an approach with a solid formalization that includes flexibility.

Various techniques to make the process aware information systems more flexible have been

proposed and some are implemented in current Business Process Management Systems. Research focuses on retaining process adherence in the Process Aware Information Systems, but at the same time break the rigidity imposed by the strict process models. In the field of research various terminology is used to refer to the practice of making Business Process Models more variable or flexible. Process flexibility concentrates on dealing both with foreseen and unforeseen changes. Changes are accommodated by varying or adapting those parts of the Business Process that are affected by them, whilst retaining the essential format of those parts that are not impacted by the variations. [6] Variability management is closely related to the notion of flexibility. Other sources use the term change management of Business Processes. Research on accommodation of variability in Business Processes has

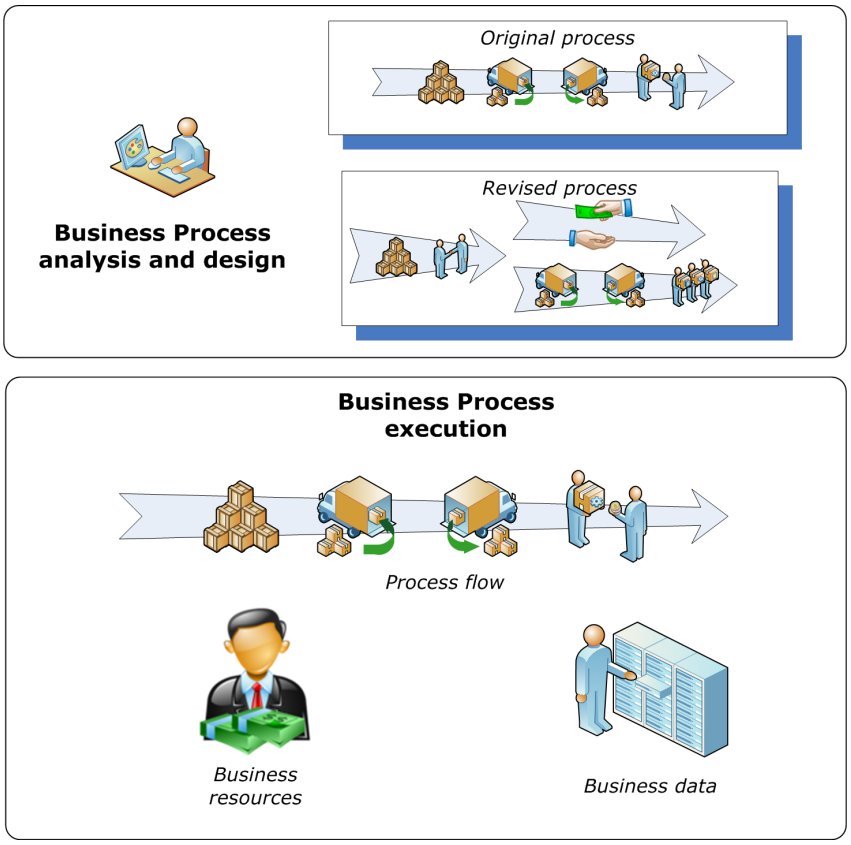


Figure 1.1: Illustration of Business Process Management

identified various requirements [7]. One category of variability requirements that still is poorly covered by existing techniques is about managing evolutionary changes the processes may undergo. This typically involves management of changes at run-time. Simultaneously multiple executions of a process may be active. Executions of the process that undergoes change will have to be treated with particular care [6]. Transferring running executions to a new process definition may not always be possible or cause inconsistencies. In this thesis management of evolutionary changes will be examined into detail. We will provide formalization of processes and provide a model for process execution on which process specification given in a formal language can be validated. Validation here means that we can say if the execution happens according to the specification provided. Figure 1.1 illustrates how Business Process designers may improve existing process models. We will show how process specifications can be altered along the way and how the behavior of executions can be controlled in this case.

1.1 Overview of document

This thesis consists of the following chapters.

- **Background:** We begin by providing the necessary backgrounds on Business Process Management and define concepts such a processes, meta processes, executions and compliance formally.
- **Related work:** In this chapter we describe the efforts already performed to provide formalization of process models. We show what has been researched already to provide variability management and solutions for handling dynamic change.
- **Problem statement:** We describe how process migration is a problem of configuration and not one of optimization. The business context imposes the constraints on the strategy, that will have to be handled differently depending on the context.
- **Realization:** The implementation is in the form of a logic program, allowing models to be specified that configure a change policy. The chapter describes how the declarative language TPL can be used to describe processes and change policies and how these can be evaluated at run-time.
- **Evaluation and results:** In this chapter we provide the results by expressing models and meta models and showing how the differences between them can be quantified.
- **Discussion and future work:** The solution is compared to the related work and unresolved scientific issues are mentioned.
- **Conclusion:** We summarize how the solution covers the aim that was intended.

Chapter 2

Background

Throughout this thesis Business Processes (BPs) are key, therefore we first need to define what is meant by Business Process. In the field terminology used with respect to Business Processes is highly cluttered, and definitions may vary considerably. In this section we therefore define what we consider a Business Process, an execution and finally describe a Process Aware Information System in abstract. The definitions given in this section will be reused in the remainder of the thesis. The definition of a process and an execution closely follows [8]. In related work processes are not always approached in a formal way and the semantics are not always clear. The definitions given here serve as a guiding thread and reference on which literature is reflected. Also later in the thesis definitions given here are used to provide argument for correctness and support the realization.

2.1 Business Processes

In this section we provide more background information on Business Processes. We examine the context of Business Processes and provide some insight in the current field of research. We explain what are called the perspectives of Business Processes and finally explain the terminology used in the field.

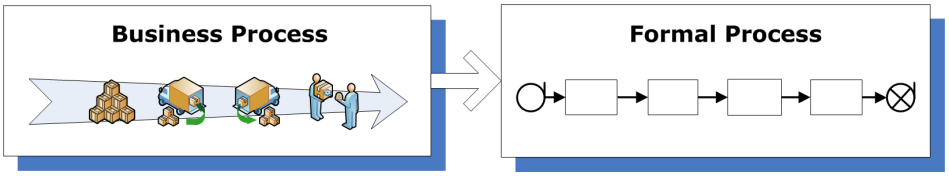


Figure 2.1: Processes are formal descriptions that correspond to real-life Business Processes

2.1.1 Business Processes in context

Business Processes have been first used by business analysts to describe the operations taking place in a business domain. Using their own informal notations they have been able to describe the flow of work activities, the usage of resources and relations to documents and data. From a business point of view such descriptions are of use to find possible improvements, typically in the form of reduction of variability, increase in productivity or improvement of service to the customer. Automation within businesses has proceeded to the

extent that now it encompasses the Business Process level. To correctly provide automation of Business Processes from a computer science perspective we need a well defined model of what is considered to be a Business Process. It is important to realize that the formal processes defined relate to Business Processes, as shown in figure 2.1, to understand the modeling aspects involved. Definition 1 is introduced as a formalization of Business Processes within this thesis. In literature the definitions and describing models vary considerably.

Business Processes are abstractions from connected work activities performed in an application domain. Application domains vary from automotive parts manufacturing, to health care, to administrative governmental processes. Abstraction from specific application domains has enabled powerful IT-support for businesses in which every activity is considered a service, even if that activity is not an automated one, but implemented by human decision making. Web-service technology is used in practice to realize Business Processes in IT-solutions. In the abstraction Business Processes are built up from atomic units of work, called activities.

As business operations take place people and entities will start to run activities and as we say instantiate the defined processes. Depending on the domain these may be long-running processes and a population of instances is formed for a given process.

2.1.2 Processes defined

We define a process as a set of activities connected using AND-gates and OR-gates. A process describes the behavior of executions. We reuse the definitions provided in [8]. Note that this representation provides alternatives for the execution whenever OR-gates are involved.

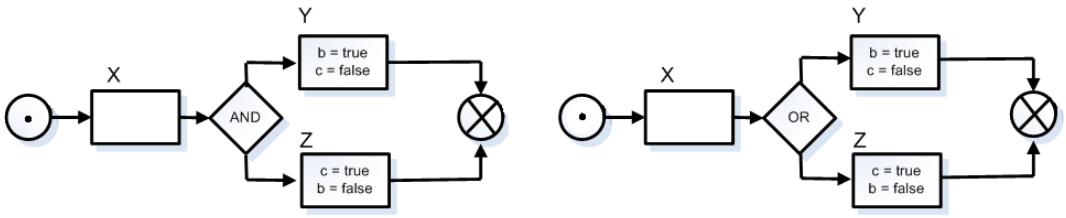


Figure 2.2: Two process models, shown as example [8]

Definition 1 (Process)**.** *A process P is a tuple $\langle A, G, T\rangle$ where:*

- *A is a set of activities, with selected start activity $\odot$ and final activity $\otimes$;*
- *$G = G_{AND} \cup G_{OR}$ is a set of gateways, where G_{AND} are the gates of type AND and G_{OR} of type OR.*
- *$G_{AND} \cap G_{OR} = \varnothing$*
- *T is a relation on $A \cup G$: $T \subseteq (A \cup G)^2$, defining the transition edges, satisfying:*
  1. *$dom(T) = (A \backslash \{\otimes\}) \cup G$.*
  2. *T is functional on A.*

Execution graph

An execution graph as in definition 2 represents any run of a process instance, not necessarily following the process. This is where we deviate from [8]. All we assume is that activities are

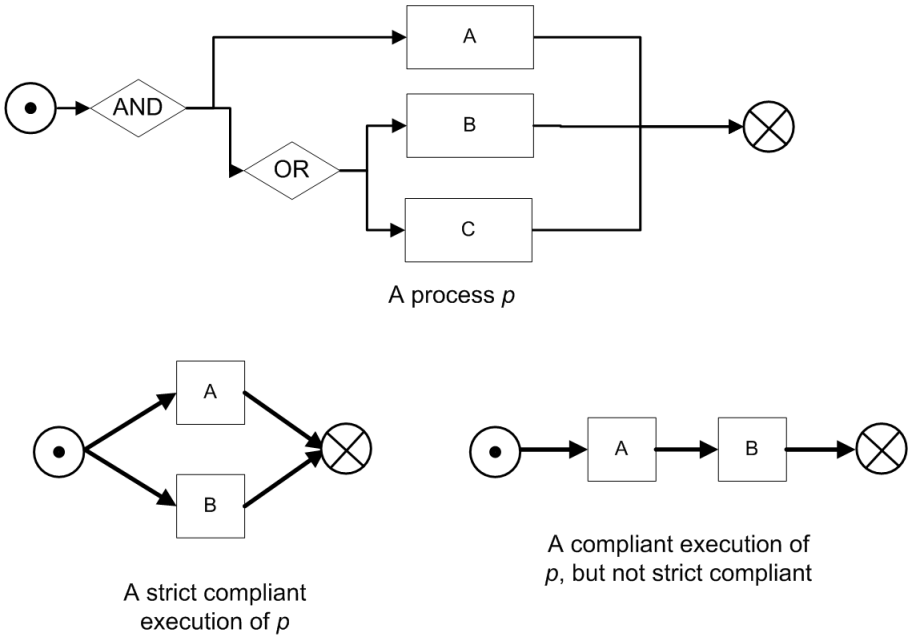


Figure 2.3: A process is illustrated and some executions. Note the absence of the gates in executions. For strict compliance the structure matches the process description. For the less strict case a directed path links the activities when the process describes a following activity but the structure does not necessarily maps to the process.

executed in some order. This is not necessarily the same order as specified in the process, see the definition 5.

Definition 2 (execution graph)**.** *An execution graph is an ordered graph* $\sigma = \langle A, T \rangle$*, where:*

- A *is a set of activities*
- T *is a relation assigning a set of next steps for each activity:* $T \subseteq A^2$
- $\otimes \notin dom(T)$

An execution graph may be progressed to a following execution graph by adding a non-empty set of states A_n and transitions T_n as seen in definition 3.

Definition 3 (execution graph progression)**.** *We say* $\sigma = \langle A, T \rangle$ *progresses to* σ_n*, denoted as* $progress(\sigma) = \sigma_n$ *when the following holds:*

- $\sigma_n = \langle A \cup A_n, T \cup T_n \rangle$
- $A_n \neq \varnothing \wedge T_n \neq \varnothing$
- $T_n \subseteq (A \cup A_n) \times A_n$

Compliant executions of processes

Compliance is the notion we use to define that an execution follows a process. In definition 4 we work this out. The structure of the process graph determines the structure of the execution graph. In the process graph three types of transitions occur. Those that lead to an activity should also be present in the execution graph. For transitions to an AND-gate,

all transitions to the activities that follow should be present. For gates of type OR at least one transition must follow.

Definition 4 (strict compliance)**.** *Given a execution $\sigma = \langle A_\sigma, T_\sigma \rangle$ and a process $\langle A_p, G, T_p \rangle$, we define that sigma is strictly compliant to P if it is constructed in the following way:*

- $A_\sigma \subseteq A_p$
- *If $a \in A_\sigma$ and $g \in G_{AND} \cap T_p(a)$, then $T_p(g) \subseteq A_\sigma \cap T_\sigma(a)$.*
- *If $a \in A_\sigma$ and $g \in G_{OR} \cap T_p(a)$, then $T_p(g) \cap A_\sigma \cap T_\sigma(a) \neq \varnothing$.*
- *If $a \in A_\sigma$ and $b \in A_p \cap T_p(a)$, then $b \in A_\sigma \cap T_\sigma(a)$.*

This definition relies on definition 4, which builds a subgraph of the process following the semantics of the OR and AND gates in the process. This execution strictly follows the process. We relax this definition a bit to allow additional activities and slightly loosened restrictions on the transitions, we do not demand that the transitions are precisely mapped, as long as the order of events is maintained.

Definition 5 (compliance)**.** *An execution $\tau = \langle A_\tau, T_\tau \rangle$ is compliant to a process $P = \langle A_p, G, T_p \rangle$ if and only if:*
There is an execution $\sigma = \langle A_\sigma, T_\sigma \rangle$ strictly compliant to P for which holds that if $(a, b) \in T_\sigma$ then $a, b \in A_\tau \wedge (a, b) \in T_\tau^+$.

2.2 Meta processes

Now that we have defined processes, we need to realize that processes are there to be executed by users. This creates instances of the process. Executions may take different paths through the process (choose different activities when encountering OR-gates), may fail at the execution of activities or deviate from the process all together by following activities not in the process. Also we usually have more than one process, in which case different executions may initiate different processes.

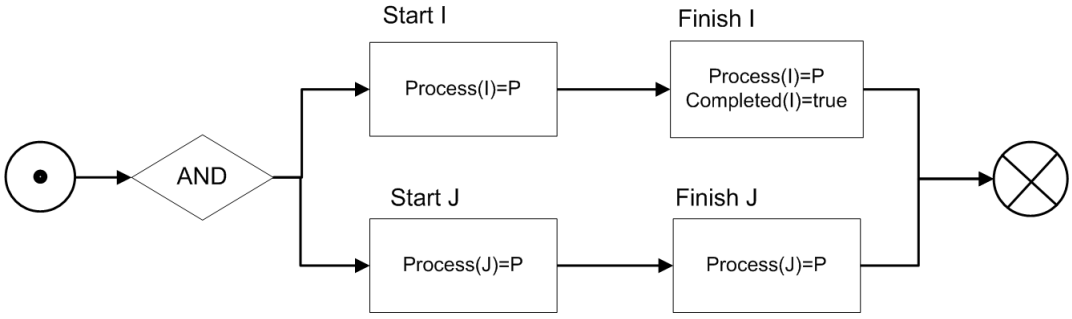


Figure 2.4: A meta process

The meta process describes how instantiation of the process takes place and what happens to instances. This will be particularly important as we discuss process migration later in this thesis. We define a meta process as a process in which the activities relate to process instances. Examples of activities in the meta process are the start and finish of an instance.

Also we may include exceptional activities such as a migration to another process and possibly other exceptional events, like cancellation.

2.2.1 Process migration defined

Process migration can be defined in the context of a dynamic system s, corresponding to a running Process Aware Information System (PAIS).

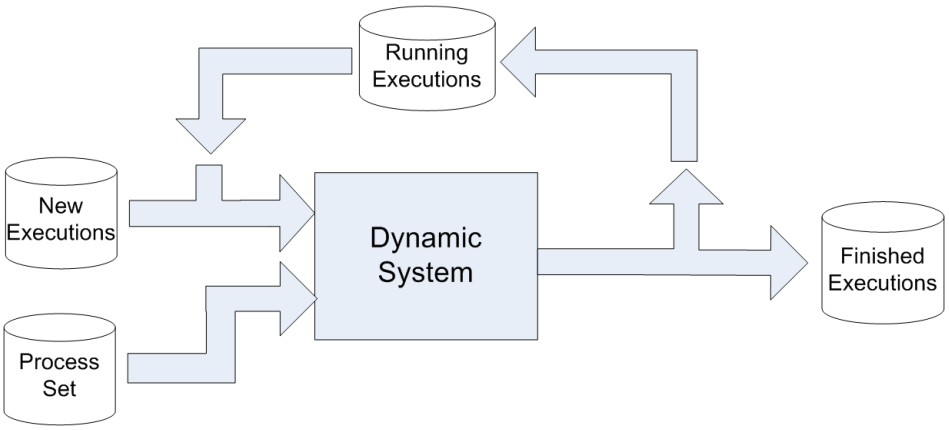


Figure 2.5: A PAIS as a dynamic system

Definition 6 (A Process aware information system)**.** *The system $s = \langle E, P, T$ is a process aware information system if:*

- *$E \subseteq EXEC$ is a collection of executions $e = \langle i, t, \sigma, \pi \rangle$, where:*
  - *$i \in I$ is a identification token of a collection I*
  - *$t \in \{\texttt{new}, \texttt{running}, \texttt{finished}, \texttt{aborted}\}$*
  - *σ is an execution graph*
  - *a process $\pi \in P$*
- *P is a collection of processes.*
- *A transition function $T : EXEC \rightarrow EXEC$. This transition function T is defined by:*

$$
\begin{aligned}
T(\langle i, \texttt{new}, \sigma, \pi \rangle) &= \langle i, \texttt{running}, \sigma, \pi \rangle && (2.1)\\
T(\langle i, \texttt{running}, \sigma, \pi \rangle) &= \langle i, \texttt{running}, \sigma', \pi \rangle && (2.2)\\
&\quad \textit{where } \sigma' = progress(\sigma) \\
T(\langle i, \texttt{running}, \sigma, \pi \rangle) &= \langle i, \texttt{finished}, \sigma, \pi \rangle && (2.3)\\
T(\langle i, \texttt{running}, \sigma, \pi \rangle) &= \langle i, \texttt{aborted}, \sigma, \pi \rangle && (2.4)\\
T(\langle i, \texttt{running}, \sigma, \pi \rangle) &= \langle i, \texttt{running}, \sigma, \pi' \rangle && (2.5)\\
&\quad \textit{where } \pi' \in P \wedge \pi \neq \pi' \rangle
\end{aligned}
$$

- *$\texttt{step}(s)$, the collection of steps of s is defined by:*

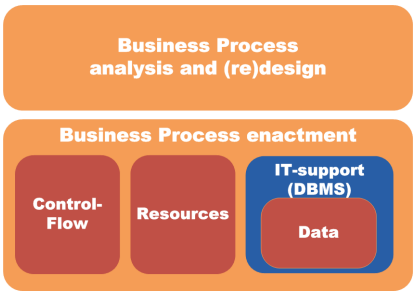
$$\texttt{Step(s)} = \{(E, E - \{e\} \cup \{T(e)\}) | e \in E \subseteq EXEC\}$$

Transitions 2.1, 2.3 and 2.4 are straightforward and give the basic state changes of executions, respectively start, completion and abortion. Transition 2.2 is where the execution progresses; it executes new activities. Transition 2.5 is a migration of an execution. A system in which

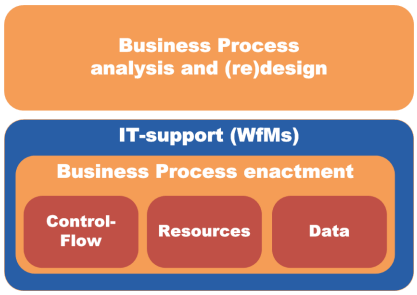
migrations take place from process π to π' is called a process migration from π to π'. Migration is in this way defined as binding a new process to an execution . Note that this is still a purely descriptive definition. We only state which transitions can potentially occur, not who performs these actions. Also we have not specified when these transitions are allowed and when they are not. It is the task of the meta process to define when migrations are allowed. In this definition no relation between the processes π and π' and the execution σ is forced. One could demand that executions are always compliant to their process but depending on the application such demands may not have to be so stringent. We leave this up to the specification of the meta process

2.3 IT-support for Business Processes

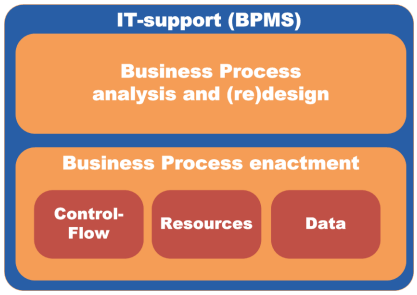
Instead of information systems that simply provide data storage capabilities, modern information systems within business are now aware of the process. To show the relevance of Process Aware Information Systems, it is interesting to put information systems in a historical perspective. Figure 2.6 shows the increasing scope of information systems used to support business operation, based on the description in [5].



(a) Database Management Systems, a data-driven approach to support business



(b) Workflow management systems focus on all three perspectives



(c) Business Process Management systems include support for all BPM practices

Figure 2.6: An overview of the history of Process Aware Information Systems, from old to new. The figures show an increasing scope; IT-support slowly begins to support all aspects of business operation and management.

A trend mentioned is the shift from data to processes [5]. The seventies and eighties were dominated by data-driven approaches. The focus of information technology was on storing and retrieving information and as a result data modeling was the starting point for building an information system. The modeling of business processes was often neglected and processes had to adapt to information technology. Management trends such as Business Process re

engineering illustrate the increased emphasis on processes. As a result, system engineers are resorting to a more process driven approach.

The last trend mentioned [5] is the shift from carefully planned designs to redesign and organic growth. Due to the omnipresence of the Internet and its standards, information systems change on-the-fly. Software development has become more dynamic and recognize not just the current processes but accommodate the Business Process Management practices that constantly change the processes involved.

2.3.1 Languages for Business Processes

A wide variety of languages to describe Business Processes exist. Languages can be divided into two categories, the execution languages and the modeling languages. A typical approach is to use a graphical notation to model the process, which may be translated to a textual language, usually in an XML-format for the purpose of execution. The purpose of the execution language is to describe precisely how Web service interaction must take place in order to implement the Business Process. Business Process execution languages thus bridge Business Processes and web-services. The execution language dictates the control-flow: the order in which activities must be executed, possible concurrent activities and may give alternative paths, or branches of the process. Also the data and resource perspectives may be explicitly described by the language. Examples of execution languages are WS-BPEL and YAWL, of modeling languages an example is BPML.

2.3.2 Business Process or Workflow?

In literature the field is split in Workflow and Workflow Management System (WfMs) approaches and the Business Management approach. The Business Process Management is the more recently emerged field and although definitions vary considerably, BPM is said to encompass Workflow Management.

Table 2.1: Comparison between Workflows, Business Processes and formalizations in this thesis

Abstraction level	**Workflow terminology**	**Business Process terminology**	**Thesis terminology**
Execution level	Work case	Business Process instance	Execution
Process level	Workflow	Business Process	Process
Meta process level		Dynamic change or process revision	Meta process

From table 2.1 we can conclude that the main difference between the field of workflows and Business Processes is that workflows do not explicitly define meta level concepts such as dynamic changes and process revision. Other difference are sometimes mentioned, such as that Business Processes tend to include possibly ongoing repetition and are more closely tied to Web-services whereas workflows model are used to model finite execution, but such difference seem more incidental limitations and are not generally assumed in either field.

2.3.3 Perspectives of a Business Process; control-flow, data and resources

Typically when we discuss Business Processes we we talk about the way activities are related, which activities must happen one after the other and which ones may happen in parallel. This is the control-flow perspective of the BP. However we may also consider what data elements activities use, the data perspective, and which resources they use up, the resource perspective. Figure 2.7 illustrates the different perspectives by example. In application fields these may be important considerations and demand planning considerations as work is conducted.

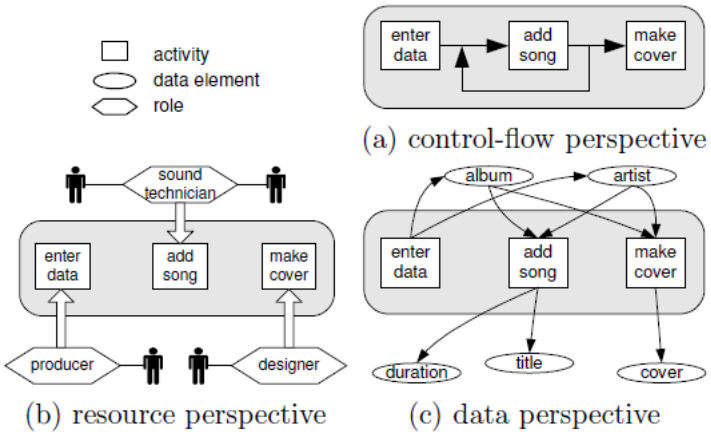


Figure 2.7: The three perspectives of a Business Process [9]

2.4 Terminology

In table 2.2 the terminology used throughout the paper is defined.

Table 2.2: Definitions related to Business Processes [10]

Term used	Alternative terms	Definition
Process		A formalization of a Business Process (definition 1).
Business Process	Workflow	A set of related work activities as conducted in a real-world business. We use this term when we refer to the business context.
Meta process level	Meta modeling	The level at which modeling of changes to Business Processes takes places. Instantiation, enforcement and migration strategies for processes are defined.
Process level	modeling level	The level at which modeling of Business Processes takes places. Control flow, data view and resource view are modeled without considering running instances.

Execution level	enactment/instance level	The level at which execution takes places and process instances are run by a process aware information system. Related data and resources are allocated when a process is run and actual business is performed.
Process level terminology		
Process model	Workflow model, procedure, schema	A model of one or possibly more Business Process using some formal or informal notation. Many modeling techniques can be applied here.
Activity	Task	An part of a process model that is considered atomic for the sake of abstraction. A unit of work that relates to a real-world job.
User		A potential participant in processes.
Role		An activity has roles to be fulfilled by some user type. Users perform a role in an activity.
Execution level terminology		
Execution	Process instance, Workcase	As processes are executed, a process instance is started for each occurrence (definition 2).
Process activity instance	Instance of activity	Each process instance follows the process and preforms activities as part of the process execution. The execution of an activity is instantiated as this happens.
Participant		A dynamic binding of a user to a role in the process.
History	Audit trail, execution trace	The history of a process instance is the sequence of activity instances performed in the past.
Stage		The stage of a process instance is the set of activity instances the process instance is performing at a given time.
Future goal set	Prescribed Future	The process instance has steps that the model prescribes. The future goal set describes the set of future activities that the process instance still has to perform to complete the process.

2.5 Business Process Management

We start by defining the terminology around which Business Process Management revolves. Next the practices of Business Process Management are briefly discussed, which will be relevant as we move to frameworks that support these practices. Before we can explain process migration, first the models of Business Processes themselves have to be explained. After this explanation, process migration is treated in detail. Relevant formal definitions related to evolutionary changes are given and the problems related to migration are specified. Next proposed techniques in other work are explained and compared. Finally the systems supporting BPM are treated.

2.5.1 Practices in Business Process Management

Business Process Management can be defined as supporting Business Processes using methods, techniques, and software to design, enact, control, and analyze operational processes involving humans, organizations, applications, documents and other sources of information [5]. This definition involves various practices that revolve around the management of processes. In this section these practices are briefly explained and their relation is clarified.

Process design

Describing processes in the organizations in a formal way is what process design is about. The process descriptions have to be an accurate reflection of the work conducted, have to be capable of handling sufficient flexibility encountered in practice and have to be precise enough to allow implementation. Process descriptions can be revealed by interviews, log analysis, higher-level businesses modeling and design methods. Exceptional flows have to be considered as well as possible optimizations and revisions. Process design is often supported by graphical tooling.

Process implementation

Process descriptions are abstract until actual users perform the process. At this time a process is actually executed and some sequence of steps (activities) is followed. Resources are consumed and people, applications and documents interact to generate a desired outcome. Binding resources to the execution can be automatically supported by a BPMS.

Process enactment

As processes are executed exceptions can occur and have to be handled. Also the actual choice of activities can deviate from the described process. Some choice has to be made on how the system should enact processes. Strict systems deny users from deviating, looser systems only log the event and do not enforce adherence to the process.

Diagnosis (or analysis)

Executed processes are typically logged and this information can be used to monitor the processes, detect errors and bottlenecks and discover new process variants.

Process revision

This subject will be revisited in section 3.1. As processes are diagnosed, this information can be used as feed-back for process design. What can happen is that a process changes over time or has to be specified in a different way to solve problems.

2.5.2 The Business Process Management life cycle

What is envisioned in Business Process Management is that the design, enactment, control and analysis of Business Processes is used in combination; one practice providing input for the next. This feedback loop is referred to as the Business Process Management cycle and when effective should lead to better, more efficient and suited processes and effective control and optimization of resources (human, organizational, applications, documents and other sources of information).

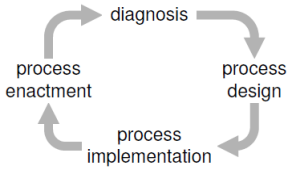


Figure 2.8: The Business Process Management life cycle

2.5.3 Web services as implementation of Business Processes

Worthwhile to mention is that the reason Business Process automation has become popular and applicable is that the activities are implementable as a set of web service invocations, allowing abstraction of implementation details and offering services by a description in WSDL. Business Process definitions are implemented as layer on top of the WSDL as shown in figure 2.9.

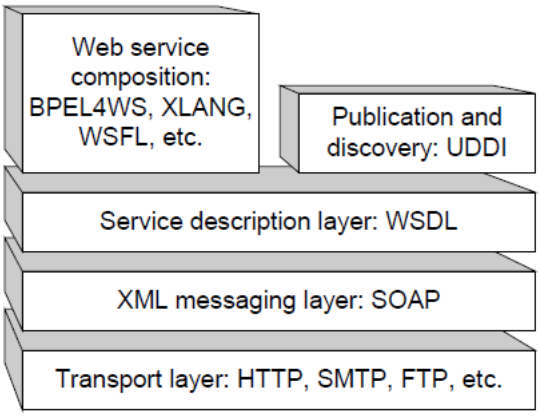


Figure 2.9: Overview of web services technology [11]

Chapter 3

Related work

The research related to this thesis is about formal modeling of Business Processes and modeling approaches that support Business Process Management and allow formalization of the constraints that these practices impose on the processes. We will closely examine the case in which a dynamic change to a process has to be handled and which aspects are interesting in this scenario. The related work on dynamic changes gives various approaches to dealing with change and some strategies to implement the change.

3.1 Variability Management

Both in the design of Business Processes as during the execution it can be beneficial to have some flexibility or variability. The notions flexibility and variability are closely related and used interchangeably. At the same time as introducing flexibility we must keep the process unchanged where it would otherwise harm the integrity of the process. Which process constraints must be retained and which ones can be relaxed depends on the business domain. To ensure that only the variation is allowed that is acceptable, the variability must be explicitly managed [7]. Various reasons for variability management are encountered in practice. A reason might be that many Business Processes might occur that have a high degree of similarity. To keep the processes manageable and to apply a redesign to all processes to which the new design could be applied, we can view the processes as variants of one another. In the definition of a process (definition 1) such variants would be various branches of an OR-gate. This form of variability management is classically described as modeling of variation points [12].

3.1.1 A taxonomy of Flexibility

In related research [6] four types of flexibility are distinguished, in table 3.1 these are described in detail. In this thesis the focus will be mainly on flexibility of the third category, and to lesser extent of the fourth. Process migration, the subject later in this thesis, provides flexibility by change as described in the table.

Table 3.1: A taxonomy of flexibility [6]

Flexibility technique	Description
1. Flexibility by design	Flexibility by design means having the expressive power in modeling a process to specify alternatives in the process model, or variants, such that users can select the most appropriate alternative at run-time for each process instance. This type of flexibility means that multiple execution alternatives are implicitly or explicitly specified in the process model.
2. Flexibility by underspecification	Leaving parts of a process model unspecified is what is meant by underspecification. These parts are later specified during execution of process instances. In this way, parts of the execution alternatives are left unspecified in the process model, and are specified later during the execution.
3. Flexibility by change	Flexibility by change is the ability to modify a process model at run-time, such that one or several of the currently running process instances are migrated to the new model. Change enables adding one or more execution alternatives during execution of process instances.
4. Flexibility by deviation	Flexibility by deviation is the ability to deviate at run-time from the execution alternatives specified in the process model, without changing the process model. Deviation enables users to 'ignore' execution alternatives prescribed by the process model by executing an alternative not prescribed in the model.

3.2 Modeling formalisms

Most of the Business Process description languages are based on some formal mathematical model or notation, although less formally specified semantics are used as well [5]. The advantage of the formally specified semantics of a process model are that properties such as correctness and expressiveness can be proven and the meaning of constructs is unambiguous. [13] gives an overview of the models used in various approaches. A very popular foundation for describing Business Processes is Π-calculus, originally a language for describing processes in concurrent systems. Since Business Processes consist of various concurrent processes, this approach is suitable. Another foundation is Petri nets, a graph based approach, that can be used to model transitions from activity to activity, thus defining the flow in a process model. A new concept is declarative; instead of explicitly modeling the process flow by defining the sequence of activities, constraints are defined on the flow using linear temporal logic. These three formalisms are explained in the following subsections.

3.2.1 Π-calculus

A way to formally model concurrent systems is to describe systems with algebraic expressions. The syntax of Π-calculus allows representation of processes, parallel composition of processes, synchronous communication between processes through channels, creation of fresh channels, replication of processes, and nondeterminism. Where a *process* is an abstraction of an independent thread of control. A *channel* is an abstraction of the communication link between two processes. Processes interact with each other by sending and receiving messages over channels [14].

Although Π-calculus is originally intended for concurrent programming, some variants have been proposed that allow direct translation into existing Business Process modeling languages, such as BPML and the newer WS-BPEL. Table 3.2 shows the syntax of such a variant. Π-calculus can be used to model any process, including how workflow works, for as it turns out, workflow itself is just a process. Its patterns can be constructed out of Π-Calculus primitives. Workflow management systems allow for the modeling of any workflow because they include workflow engines [15].

Table 3.2: webπ_ω-calculus; the formalism behind WS-BPEL [16]

Syntax rules	Description
P ::=	
0	**(nil)**
| P | P	**(parallel composition)**
| $\Sigma_i x_i(\tilde{u_i}).Pi$	**(alternative composition)**
| `if` $(x = y)$ `then` P `else` Q	**(conditional*)**
| $\bar{x}\langle\tilde{u}\rangle$	**(output)**
| $(x)P$	**(restriction*)**
| $!x(\tilde{u}).P$	**(lazy replication)**
| $\langle\!\langle P; P \rangle\!\rangle_x$	**(transaction*)**

* denotes an added construct with respect to the original Π-calculus

Variability in Π-calculus

Attempts have been made to include variability in Π-calculus based languages, but these attempts are limited to allow flexibility by design only. The way variability is included in VxBPEL is by introducing additional language constructs that explicitly model variation points [17].

3.2.2 Petri nets

Instead of describing Business Processes with algebraic expressions, petri nets are used to describe Business Processes using a special type of graph. In [18] petri nets are described. The petri net is built up out of three components; places, transitions and arcs. Places are nodes of the graph that may contain any number of tokens. Transitions are nodes of the graph that indicate connections between the places. The arcs are directed edges connecting a place and a transition. A place is referred to as an input place p of t if there is an edge from the place p to a transition t, vice versa a place can be called an output place p of t. The petri net operates by firing rules of the transitions.

- A transition t is said to be enabled iff each input place p of t contains at least one token.
- An enabled transition may fire. If transition t fires, then t consumes one token from each input place p of t and produces one token in each output place.

Figure 3.1 gives an example of a petri net in operation. Petri nets can be used as models for Business Processes as the structure of nodes and arcs describes the flow of a Business Process. The petri net also provides a model for execution as tokens can reflect process instances. Using graph representations can in some cases be a better way to express processes than the Π-calculus as petri nets are more powerful in expressing complex control-flow and in Π-calculus control flow must be described by nesting constructs [19].

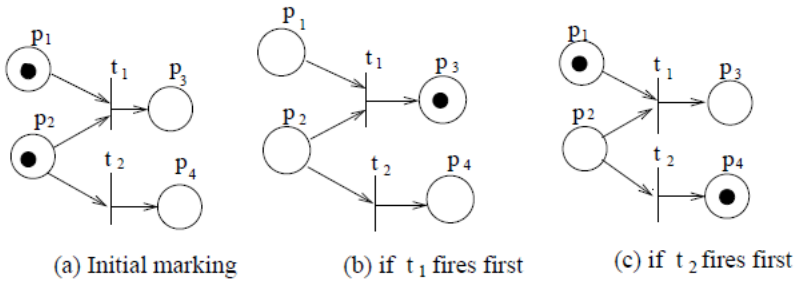


Figure 3.1: An example of a petri net [20]

Variability in Petri nets

The classic formalism of petri nets has been expanded in various ways to be better suited for describing Business Processes. Addition have been the distinction of colors for tokens, allowing the separation of different instance classes, and inheritance of petri nets, describing inheritance relations between petri nets [21]. For variability management inheritance of petri nets is convenient as it enables petri nets to describe process design variants in an elegant way [22].

3.2.3 Declarative modeling

The process can be either imperatively described or in a declarative fashion. Imperative means that the description explicitly states all the possible execution paths, with possible branching to allow multiple possibilities. In a declarative process model all possibilities are open by default and the model constrains the allowed execution by explicitly stating constraints on the full set of possibilities. Several types of logic formalism can be used in this approach. The approach tries to apply a variant of logic to a process model, sometimes a graph based model and sometimes a trace based model. The logic variants considered are usually based on First Order Logic, some examples of logics applied to Business Process modeling are Computational Tree Logic (CTL), Linear Temporal Logic (LTL) and Temporal Process Logic (TPL), a logic developed with Business Process modeling in mind. The various logic formalisms are briefly discussed.

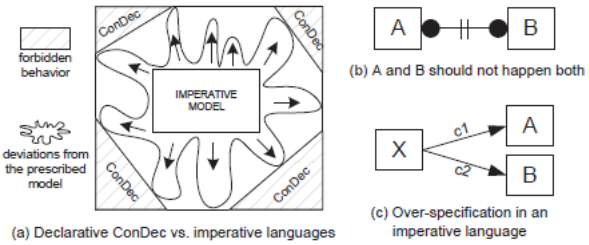


Figure 3.2: Declarative modeling versus imperative modeling [23]

Linear Temporal Logic (LTL)

[24] uses linear temporal logic to describe Business Processes. In LTL formula can use classical logical operators ($\neg$,$\wedge$ and $\vee$) and several additional temporal operators ($\bigcirc$, U, W, $\Box$ and $\Diamond$). The semantics of operators $\neg$,$\wedge$ and $\vee$ is the same as in the classical logic, while operators **U**, **W**, $\Box$ and $\Diamond$ have a special, temporal, semantics.

- Operator always ($\Box p$) specifies that p holds at every state in time

- Operator eventually ($\Diamond p$) specifies that p will hold at least once.
- Operator $next(p)$ specifies that p holds in the next state.
- Operator until ($p\mathbf{U}q$) specifies that there is a position where q holds and p holds in all preceding states.
- Operator weak until ($p\mathbf{W}q$) is similar to operator until ($\mathbf{U}$), but it does not require that q ever becomes true.

Computational Tree Logic (CTL)

This logic, again an extension of boolean logic, uses a branching tree as a model for time. Different sequences followed in time correspond to different paths followed in the time tree. This branching characteristic allows CTL to express statements on possibilities in the future, given that a past sequence is observed [25]. The operator $\mathbf{E}x$ describes that there is a path from the current state to the state x. The operator $\mathbf{A}x$ describes that all paths from the current state lead to the state x. This makes CTL slightly more expressive than LTL, but the branching time model makes that it as unsuitable for process modeling.

Temporal Process Logic (TPL)

This logic is a variant that is developed recently with Business Process Modeling in mind. It provides a limited set of operators that still allows more expressive power than found in LTL by expressing possible alternatives in the future, like CTL. In TPL the atomic variables are not positions in the tree, but logic variables that have a truth value corresponding with states, but are not exactly states, allowing the boolean operators to be expressed over atomic variables as well. This allows more freedom in expression, TPL introduces the following temporal operators on top of the standard boolean operators, $\rightarrow$, $\rightsquigarrow$ and $\rightrightarrows$. The temporal operators are informally described as [8]:

- $\rightarrow a$ means there is a state a in the process, and this state can always be achieved from the current state following the process model. In case of parallel splitting, at least one of the parallel branches must lead to the state a.
- $\rightrightarrows a$ means there is a state a in the process, and this state can always be achieved from the current state following the process model. In case of parallel splitting, all of the parallel branches must lead to the state a.
- $\rightsquigarrow a$ means there is a state a in the process, and this state can (but not always) be achieved from the current state following the process model.

3.2.4 Concluding remarks on modeling formalisms

Traditionally processes have been modeled imperatively. Imperative modeling means that all flows of the process are described explicitly. On these formalisms work has been dedicated to increase the flexibility by allowing variants of the process to be taken into account as well. The petri nets have traditionally been popular while Π-calculus has more recently become popular [19] as it is the formalism behind the popular web-service composition language WS-BPEL [16]. A new approach works the other way around, in declarative modeling all flows are implicitly allowed and only the hard restrictions are explicitly mentioned. This approach is inherently more flexible, although it has been noted [26] that in practice this approach is not always practical as non-experts find explicit flow graphs easier to understand.

3.3 Dynamic change and process migration

In business, procedures may evolve, either voluntarily due to process improvement or involuntarily due to changes in work environment or operations. Procedural changes, performed in an ad hoc manner, can cause inefficiencies, inconsistencies, and catastrophic breakdowns within offices [27]. Such changes have requirements of their own, setting constraints on the way the change should be performed. What makes the requirement for supporting changes over time interesting is when the changes is said to be dynamic. Dynamic entails that the change is made in the midst of execution (i.e. while some executions are in progress).

3.3.1 What is change: specification criteria

In [28] an analysis of the possible occurrences of changes is given, shown in table 3.3. This is interesting as it provides classification of considerations for the change handling approaches that are examined in this section. Criterion 4 has led to different approaches to support for dynamic changes as we will later see. Depending on whether a change is seen as a re-linking to a new process model or schema or a change in structure, different solutions are proposed. Approaches often neglect aspects of the full spectrum of change criteria. Where criteria 1 to 5 more have to do with what a change is, criterion 6 has to do with how the change should be handled. In the next subsection we will treat this aspect.

Table 3.3: Criteria for the classification of dynamic changes [28]

Criterion	Explanation
1. What is the reason for change?	Change can be motivated either from a change in business context, changing legal context or changing technological context.
2. What is the effect of change?	The effect of a change can be momentary, affecting just one instance (change by deviation) or affect the process in its entirety (evolutionary change).
3. Which perspectives are affected?	Any of the process perspectives can be subject to change.
4. What kind of change?	A process can be either structurally extended reduced or partially replaced, or a re-link may take place, in which the instances are linked to a new process.
5. When are changes allowed?	Change can be allowed at entry time or on-the-fly. For dynamic changes on-the-fly adaptation of changes is examined.
6. What to do with existing cases?	Changes are critical when they occur while there are cases being handled. Therefore, it is important to decide what to do with these cases (work-in-progress).

Business Process version management

When defining change, what distinguishes a new version of a process has to be carefully defined [29]. Some approaches define change patterns [30] to clearly define version transitions. A new version can be defined by set of change patterns applied to the old model. Then solutions for handling each change pattern are defined, which combined can facilitate more complex changes.

3.3.2 How is change handled: different strategies

Supporting evolutionary changes in a PAIS is one of the areas of Business Process variability management. The challenge is to propagate the change to process instances at run-time.

Basic migration strategies

Four basic options for treating running process instances are identified in [6] as well as in [31] under a slightly different terminology, as summarized in table 3.4. This refers to criterion 6 of table 3.3.

Table 3.4: The basic migration strategies as described in [6, 28]

Strategy	Description
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 model.
Detour	For momentary changes it is often wise to allow a temporary detour such that the unexpected situation can be cleansed.

Advanced process migration

Of these options to migrate, the transfer option is the most challenging and interesting case, as it involves various non-trivial issues. Various advanced solutions have been proposed for migration of process instances to a new process model [32, 33, 34]. The process instances must be verified to meet the criteria of the new version of the model and have all the required data. Another issue involved is to either migrate all process instances in the same way or allow some variation. This can be beneficial as for instances migration might not be possible and some different solution has to be found. Figure 3.3 shows how different sub-sets of the population might be treated differently during a migration. In the adaptation to a dynamic change we have the choice to determine the policy for instances either globally, per sub-set or individually.

Compliancy of the execution

Transferring instances to the new version of the process model is of the options in table 3.4 the most beneficial to individual instances. As the new process version is introduced with good reason, it is in general advantageous to move as many instances as possible to the new process. However a transfer may not always be possible as the activities already executed by instances may contradict the new process model. Whether the path of executed activities is also allowed in the new process model is what is defined by the notion of compliance (definition 5).

Research has been done to relax the notion of compliance [26], dividing process instances not in an all-or-nothing way, but several compliance related classes. This way instances can

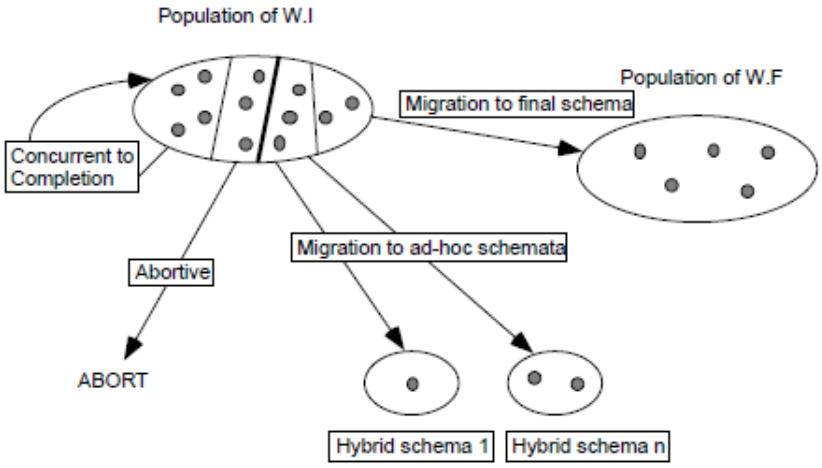


Figure 3.3: Different migration strategies for different partitions of the population of process instances [35]. W.I. is the original process with its executions shown inside, W.F. is the new process version.

still benefit from migration also if these are not strictly compliant. Compliance checking [36] is a research topic in itself and will not be treated in detail in this thesis.

Dealing with incompliancy: Abortion, roll-back and compensation

When an instance is not compliant, one solution is to make it compliant by aborting the activities that are already performed and starting again from scratch. However aborting a task may be more complicated then just removing the related administration in the Business Process Management system. For example when an instance has performed an activity "enroll", he actually has to unenroll to undo the enrollment, just removing the "enroll" activity from the system is not correct with the real-world Business Process. What is usually done to solve this problem is to define a compensation task; a task that must be performed to undo the task afterwards. In reality compensation is not always possible and involves high-costs and delays. In [32] an optimal roll-back process is described; instead of restarting the process from scratch, the process is only partially undone and brought to a compliant state with a minimal number of aborted activities. The approach of roll-back does seem a bit outdated as is remarked in [33]. Roll-back to a consistent state will be a very costly undertaking and other ways to treat non-compliancy are often preferable.

3.3.3 Considering business context requirements: Semantic constraints

In many application domains, processes are subject to business level rules and policies stemming from domain specific requirements (e.g., standardization, legal regulations). To clearly distinguish between syntactic constraints and business level rules and policies, we refer to the latter as semantic constraints [37]. For a particular Business Process, semantic constraints may express various dependencies such as ordering and temporal relations between activities, incompatibilities, and existence dependencies. The question is how we can constrain and control the migration according to the specific application needs[37]. Taking into account semantic constraints choosing the best migration strategy is not a clear-cut choice. Also it becomes clear that the meta process level is also faced with its own semantic constraints and the basic migration strategies are not specific enough to cope with business requirements.

3.4 Meta modeling approaches: change specification

We have seen the approaches behind modeling at the process-level. Now we will examine approaches at the meta process level, in particular how are changes to processes characterized and modeled. Change in imperative models is hindered by the fact that an equivalent new state must be found in the new model, which is not always possible [27]. In an imperative process modeling context the problem of process migration one of matching a source and destination graph and determining equivalent states and change regions. For declarative models it is straightforward to transfer instances [24]. Instances for which the current trace satisfies the constraints of the new model, are mapped onto the new model. Hence the "dynamic change bug" described in [27] does not apply.

Table 3.5: Approaches for specification of dynamic changes

Approach	Modeling approach	Dynamic change problem description	Proposed solutions
Change operation primitives	Imperative	A dynamic change is a relative change of the process	■ define correct solutions for each atomic change operation, such as activity insertion/removal/parallelization ■ define change operator composition
Conditional migration rules	Imperative	A dynamic change means one process is exchanged by a different process schema	■ Verify specified conditions on all instances ■ Apply the specified replacement schema to the instance ■ back-track from the old schema, and apply new changes, using roll-back where possible to handle state inconsistencies
Dynamic constraint model application	Declarative	A dynamic change applies changes in the constraint set	■ Attempt to re-link all instances to the new constraint-set and apply the new constraints ■ check constraints on the instance for structural-compliancy ■ check constraints on the instance for state-compliancy ■ let user decide what to do with the different cases encountered and possibly which constraints to ignore

3.4.1 Change operation primitives

Various solutions describe the modeling of change primitives, based on the approach de-

scribed in [35]. The approach describes the Workflow Modification Language (WFML). The Workflow Modification Language describes each change to a Business Process as a set of consistency preserving primitives applied over the process, sometimes denoted as a Δ operation onto a process. The approach successfully maintains correctness in each step. Change primitives are divided into two categories, declaration primitives and flow primitives. Most change primitives do not cause correctness problems, except where the flow is affected in the past of instances. [27] refers to the problem of changes occurring to the past of an execution as the "dynamic change bug". In the approach the change itself is well-defined, the policy to handle the change is not defined in the primitives, instead the approach resorts to the basic migration strategies to accommodate the change.

3.4.2 Conditional migration rules

Another approach to specify changes to processes is by introducing a set of rules that govern the change. In the experimental framework eFLOW [38], bulk migration rules are introduced. These rules take the form of event triggers, if a condition is met, then a migration of some sorts is activated. Such conditional rules are found under different terminology in $\mathcal{ML} - \mathcal{DEWS}$ [39].

3.4.3 Dynamic constraint model application

In [24] the DECLARE framework is described, a fully declarative modeling framework for Business Processes. Although the framework does allow flexible modeling of processes, at the meta process level it has a fully static change procedure. In [40] the SEAflows framework is presented. The SeaFlows framework is a hybrid approach in the sense that allows imperative process modeling complemented by declarative constraints. Interesting about this project is that constraints can be expressed at a meta process level and later be instantiated to individual processes. Still these meta process level constraints do not say how the change is handled, but just give detailed specification conditions.

3.4.4 Comparison of approaches

It is one thing to describe what has to done during a dynamic change, all of the solutions provide a way to exactly specify the modification to the process. Another aspect is to be able to specify how the change should be implemented. This we will call the change policy, a more general term for the migration strategies described earlier. A change policy means a specification of the behavior of the system at run-time, it involves a specification of all the criteria of the change at hand. Which behavior of existing instances is demanded, how are new instantiations treated and which additional semantic constraints have to be considered. The basic migration strategies we view as simple cases, more elaborate change procedures are described later.

Table 3.6: Comparison of dynamic change specification approaches

Approach	Flexibility	Change policy expression	Introduced problems
Change operation primitives	**None**, as correctness of change operations is strictly maintained.	A basic migration strategy is applied, the policy is **not specified** by the change operations.	Change operations to the history are not always possible.

Conditional migration rules	With the selection of event conditions and actions **some** flexibility in migration is enabled.	Migration strategies are conditionally applied, but constraints are **not expressed** on the migration itself.	Specification of the conditions usually is limited to simple filter operations, increasing expressiveness is problematic.
Dynamic constraint model application	Constraints and fine-grained compliancy models allow **good** flexibility in migration options.	Constraints on the policy are **not expressed**, instead the user decides on-the-fly how the change is applied, either for specific or general cases.	Constraint management becomes a problem in itself, also the higher complexity in different compliancies makes understanding consequences hard.

Table 3.6 compares the different approaches. It is clear that although changes can be specified correctly and in detail, the dynamic constraint model allow most expressive power and control over migrations. The change policy is however not usually explicitly modeled by these approaches.

3.5 Process Aware Information Systems

The systems that implement Business Processes and support management are numerous, but few offer dynamic change support. A Process Aware Information System (PAIS) can either perform a combination of process enforcement and monitoring [41] or just perform monitoring of webservices [42] and do nothing about process violations. Approaches typically focus on imperative process modeling and process migration is sometimes a feature but not investigated in detail. Dynamic change is thus sometimes supported but only in a basic manner. The ADEPT2 framework is an exception that provides advanced migration support, differentiating between compliancy classes and providing advanced migration strategies that exploit properties of the instances. Strangely not much work on evolutionary changes is done for declarative frameworks, while these are regarded as more flexible in nature, especially on run-time variability [7]. Of declarative frameworks DECLARE is examined. The eFLOW framework is also examined, which is relevant as it features dynamic web-service composition.

3.5.1 The eFLOW framework

Although eFLOW is not a true PAIS as the other systems examined in this section, it is the only system that actually provides a working implementation for web service composition. Where the other systems do provide extensive Business Process support the actual automation of the processes is neglected, only implemented by some translation to an existing Business Process execution engine that is in turn capable of only handling static web-service composition. DECLARE for example can only enforce the project by interfacing with a YAWL engine, losing some of the flexibility because of this. The eFLOW framework however focuses on dynamic web-service composition and although it describes Business Processes at a low level, does provide some migration support in the form of ECA rules. Of the migration rules a condition is a defined as a predicate over service process data and service process execution state that identifies a subset of the running instances, while

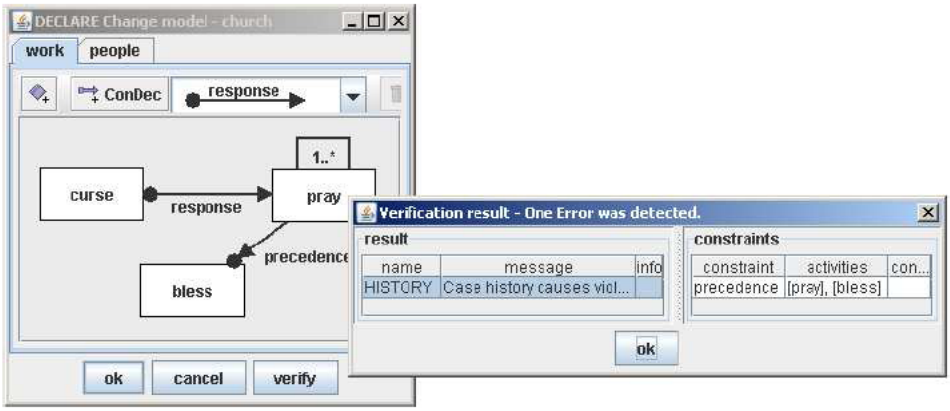
`<schema>` denotes the destination schema [38]. Instances whose state does not fulfill the migration condition will proceed with the same schema. An example of a migration rule is:

```
IF (guests>100) THEN MIGRATE
TO "Bonus_Ceremony_Service"
```

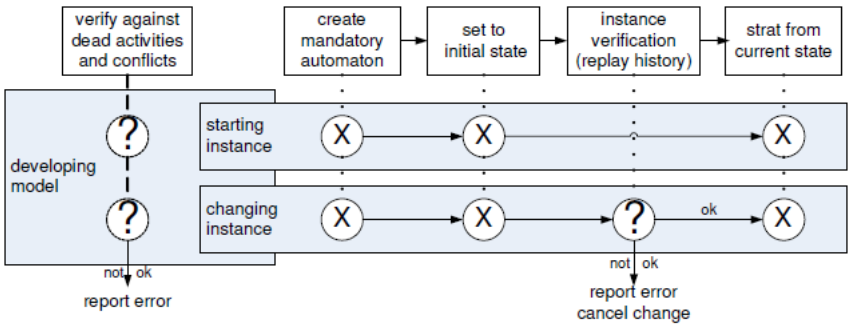
eFLOW uses a `migration manager` subsystem to implement process migration. The migration manager can be instructed by a user to perform a dynamic change. In this case the execution of instances is temporarily suspended while the migration manager does the necessary updates. The implementation does allow consistency rules to be formulated for process schema and application of migration rules.

3.5.2 Current migration support in DECLARE

Migrations in DECLARE can be initiated by the user, but the implementation is rather static. Process migration in DECLARE results in linking all newly defined constraints to all the instances in the original process [24]. If one of the instances fails to comply to the the new constraints a global error occurs and the change is canceled entirely. The results of such a cancellation are shown in figure 3.4(a). The procedure for handling a process migration is completely static but well defined. A meta process level description of the procedure can be seen in figure 3.4(b).



(a) Screenshot of a process migration cancellation



(b) The procedure for process migration, shown as abstract procedure

Figure 3.4: Process migration in the DECLARE framework illustrated [9]

3.5.3 Advanced migration strategies and optimizations in ADEPT2 framework

In order to deal with Business Process changes the ADEPT2 framework enables quick and efficient schema adaptations at the process type level (schema evolution) [43]. The implementation of ADEPT2 considers correctness of loops in the process, as well as data dependencies and uses a relaxed form of compliance, allowing traces to be different but still tolerable as important temporal relations are maintained none the less. The ADEPT2 offers the most comprehensive and advanced process migration support found in any of the frameworks examined.

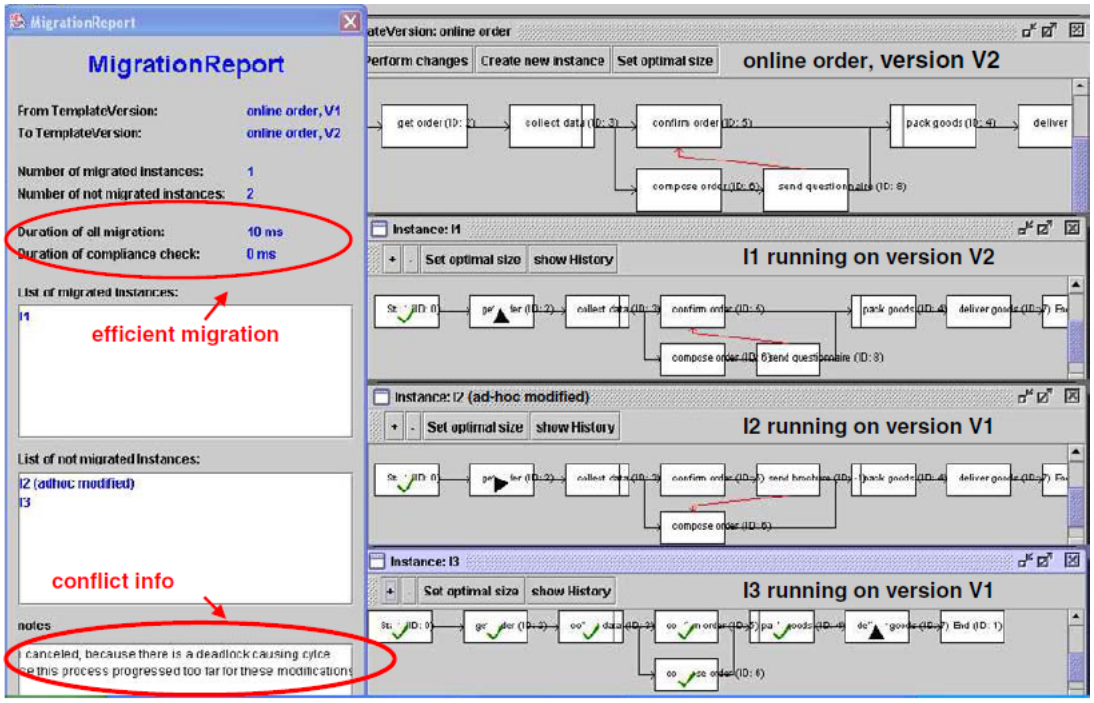


Figure 3.5: Process migration report in ADEPT2 prototype, showing instance Dependant migration policies [43]

Chapter 4

Problem statement

The process migration problem is perhaps best explained by providing some typical complex requirements faced in migration scenarios. The goal is to investigate a system that supports the verification of business constraints on not only processes but also on the procedures or policies that change processes.

4.1 Aim: A configurable solution for process migration

The aim of this research is to build a process aware information system in which a declarative process can be migrated at run-time to another process and the migration strategy or meta process can be defined declaratively. To implement processes the declarative language TPL is used. The whole migration process and the progress on the instance level should be trackable and manageable The meta process may include both instance specific constraints, such as there may be at most X instances in a certain state when an instance migrates as global constraints. Envisioned is that all hard-criteria, whether semantic, real-time, or resource or data dependent, can be added to the meta process definition eventually. And the user can plan his execution path within the set boundaries. Initially the aim is to support just the flow related constraints.

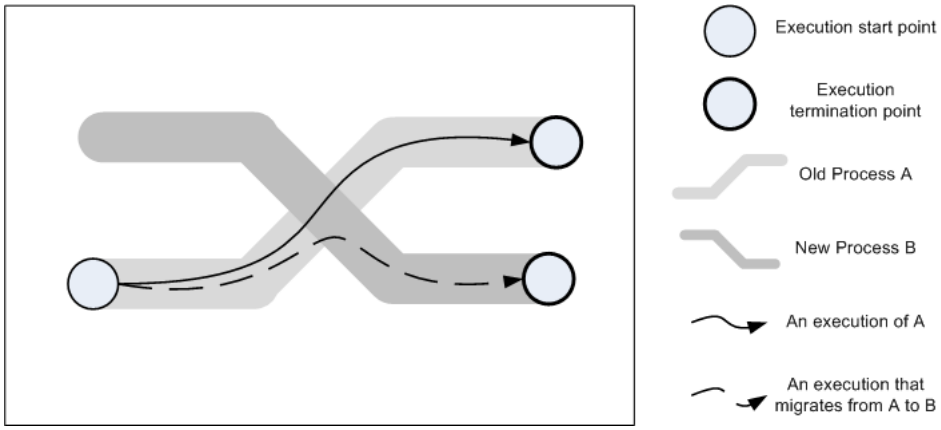


Figure 4.1: Abstracted view on process migration

Figure 4.1 shows migration in an abstracted way. It shows processes as areas constrained by their hard-criteria. Execution paths can be seen as traversals of the area. The figure shows one execution that follows the old process; conform the 'proceed' strategy, and another

diverting to the new process; conform the 'transfer' strategy. It is the meta process that constrains the allowed execution paths.

4.2 Reflection on the state-of-the-art of frameworks

For supporting repetitive procedural Business Processes organizations rely on Process Aware Information System. However since Business Processes are usually not static and may vary in practice, contemporary PAIS are unable to provide the flexibility or variability required. The DECLARE framework [9] has been proposed to address this issue by proposing a new way to model Business Processes that inherently allows flexibility and only specifies the constraints on the processes. Within a controlled process description users are given the flexibility to decide in which way they want to complete the process.

A problem that is still barely addressed in this approach is that process descriptions may need to change over time and these changes should be propagated to the running instances as well in a flexible way. Although dynamic change is supported, the state of instances may prevent the migration all together. In the DECLARE framework the behavior with respect to how dynamic changes should be performed is fixed, and cannot be configured. Although advanced strategies for handling migration have been proposed, these may or may not be beneficent depending on the application area, having its own set of requirements. In the ADEPT framework views of the population of instances are provided and some flexibility in the choice of a migration strategy is provided. In ADEPT some choices are presented to the user for migration implementations, which although are advanced are in turn non configurable The eFLOW framework takes another direction, instead of leaving choices for the change policies to the user on-the-fly, it allows rules to be specified that mange this to some extent. Unfortunately the rule specification is not approached formally and many other flexibility issues are neglected.

Important to realize is that in general there is no optimal strategy for process migration, instead the strategy and requirements depend on the business context in which the dynamic change has to take place. Some of the requirements may be hard-criteria, in the sense that some actions are forbidden. Other requirements may be soft-criteria in the sense that some actions may be preferred from the business context. For example when a new version is introduced a migration to the new version may be preferred, but it is not forbidden to remain following the old process. Such softer constraints will often be demanded on the user side, so the user will actually benefit from having flexibility in choosing the actions that within the hard-criteria is allowed. Handling dynamic changes in a running process, following such requirements from the business context, actually becomes a process in itself, what we call a meta process What is required to accommodate the hard-criteria requirements from the business is to allow flexibility in the specification of the meta process This research addresses the meta process specification in a declarative fashion.

Process migration is commonly approached as a problem of changing the structure of the process. The question then is whether it is allowed to swap activities A and B in the process, or whether these may be executed in parallel. The solutions for these structure changes are algorithmic in nature and do not offer much room for configurable choices. In this research the problem is approached not as one of process structure changing, but one of applying a different set of expressions on the fly. The expectation is that this approach by declarative modeling will be able to provide better flexibility.

4.3 The need for dynamic process change

In this section process migration is approached more from a business point of view. Example

stories are used to provide insight into the complexity of migration of processes. The stories are then used to categorize the requirements.

4.3.1 Some example stories from different domains

Three fictional examples are used to motivate that different business contexts result in different requirements for the migration. Different migration strategies will be required to implement these requirements. The stories also illustrate the complexity of potential domain requirements, far beyond the capabilities of current PAIS.

Example case 1: Shipping company introduces new billing procedure

The shipping company Widgets decides to do shipping and billing in parallel and to replace manual shipping and billing with computerized. Originally shipping had to be completed before billing the client. All employees of shipping and billing are required to follow a short course to learn to use the new software correctly. The course takes two weeks and all employees must be ready for fully computerized operation by September 1, 2011. A shipment must be registered either fully manually or entirely by the new computer system.

Exception: On Saturdays rushed shipments are handled even though billing does not work Saturdays. Manual billing of shipments of the Saturday must thus be handled on Monday. For computerized billing this restriction does not apply.

Example case 2: Car manufacturer switches production to convertibles

The car manufacturer Fuelhead & Sons discontinues one of their production cars completely in October 1, 2011. However a new convertible variant of this car is introduced, full production of this new car should be initiated on October 1st exactly. Before that time all cars of the old type that are due to be delivered before October 1st will still be produced as usual, any cars in production that are scheduled after this deadline are to be discontinued immediately and its components are sent to disassembly Only new car orders of the new variant are accepted as of now.

Exception: Sufficient resources must be invested in developing 8 cars to show-case the new variant by September 10, if necessary production some additional cars of the old variant must be canceled to accommodate the required production capacity.

Example case 3: Hospital offers new treatment

A hospital has a much lower risk operation as alternative for heart patients. Unfortunately only one surgeon is certified to lead the new operation. It is decided that a doctor will examine a subset of patients that particularly benefits from the new procedure to fit in the surgeon's schedule, mainly consisting of elderly and children. It is ruled that patients can participate in the new pre-operation examination while they are waiting for or being prepared for the old procedure. When selected the patients are operated on in the new way, otherwise they follow the traditional operation. Either way, patients must complete treatment in a 2 month time span. Of course the patients can only participate in the examination for the new treatment if they have not already received conventional treatment.

Exception: When approved by the director a deviating procedure may be used for patients as long as a special report is made and no more than 20 cases deviate annually.

4.3.2 Domain dependent requirements at the meta process level

If we look at the three stories described in the previous section we can see that process migration can involve more considerations than just correctness of the control-flow. Temporal constraints occur, some of which may have real-time deadlines, but also constraints on data, resources or specific occurrence relations. All of the constraints may be expressed over

the entire population of running instances, over individual instances or over specific subsets. Also we observe that some of the requirements expressed in the sample cases refer to either one of the processes and can thus be encapsulated at the process level. Many requirements are specific not to one of the processes involved but to the transition process involved, in other words conditions of the meta process are described. At the meta process level constraints may again be of any type expressed over any subset of instances. In a manufacturing context aborting process instances might be fully acceptable as the main concern is productivity, whereas in a hospital ceasing treatment of a patient is out of the question. From the stories we can see that the migration strategy depends on the domain requirements. The domain requirements or constraints can be divided into various categories, by type and scope, examples of different constraints and their categorization is provided in table 4.1.

Table 4.1: Examples of constraints extracted from the stories and placed in different categories

Constraint type	**Process level example**	**Meta process level example**
control-flow (logical)	"On Saturdays rushed shipments are handled even though billing does not work Saturdays." **(Ex1)**	"Only new car orders of the new variant are accepted as of now." **(Ex2)**
control-flow (temporal)	"The shipping company Widgets decides to do shipping and billing in parallel" **(Ex1)**	"Of course the patients can only participate in the examination for the new treatment if they have not already received conventional treatment" **(Ex3)**
real-time	"..all employees must be ready for fully computerized operation by September 1, 2011" **(Ex1)**	"The car manufacturer Fuelhead & Sons discontinues one of their production cars completely in October 1, 2011" **(Ex2)**
resource	"Unfortunately only one surgeon is certified to lead the new operation. " **(Ex1)**	"..subset of patients that particularly benefits from the new procedure to fit in the surgeon's schedule" **(Ex3)**
data	"A shipment must be registered either fully manually or entirely by the new computer system." **(Ex1)**	"When approved by the director a deviating procedure may be used for patients as long as a special report is made .." **(Ex3)**
ad-hoc process creation	Not applicable	"When approved by the director a deviating procedure may be used for patients .." **(Ex3)**
Constraint scope	**Process level example**	**Meta process level example**
(individual)	"The course takes two weeks.." **(Ex1)**	"Either way, patients must complete treatment in a 2 month time span." **(Ex3)**
(subset)	"Manual billing of shipments of the Saturday must thus be handled on Monday." **(Ex1)**	"It is decided that a doctor will examine a subset of patients that particularly benefits from the new procedure to fit in the surgeon's schedule." **(Ex3)**
(global)	"..all employees must be ready for fully computerized operation by September 1, 2011" **(Ex1)**	"The car manufacturer Fuelhead & Sons discontinues one of their production cars completely in October 1, 2011" **(Ex2)**

From the categorization of constraints in table 4.1 we can express requirements for any PAIS that is required to implement the constraints from the stories. For the aim of this research mainly the expression of the constraints of different types and scopes on the meta process level is of interest.

4.4 System context

In this section the problem is regarded from a software engineering perspective. As we are considering a system for process migration, the involved participants, system components and use case descriptions will need to be considered. Research will exclusively focus on the logic framework to support configurable process migration, still the essential other components and the relation of the logic framework to these components is important as those determine the requirements and interfaces of the framework. The interaction of related systems and actors is shown in the system context diagram in figure 4.2.

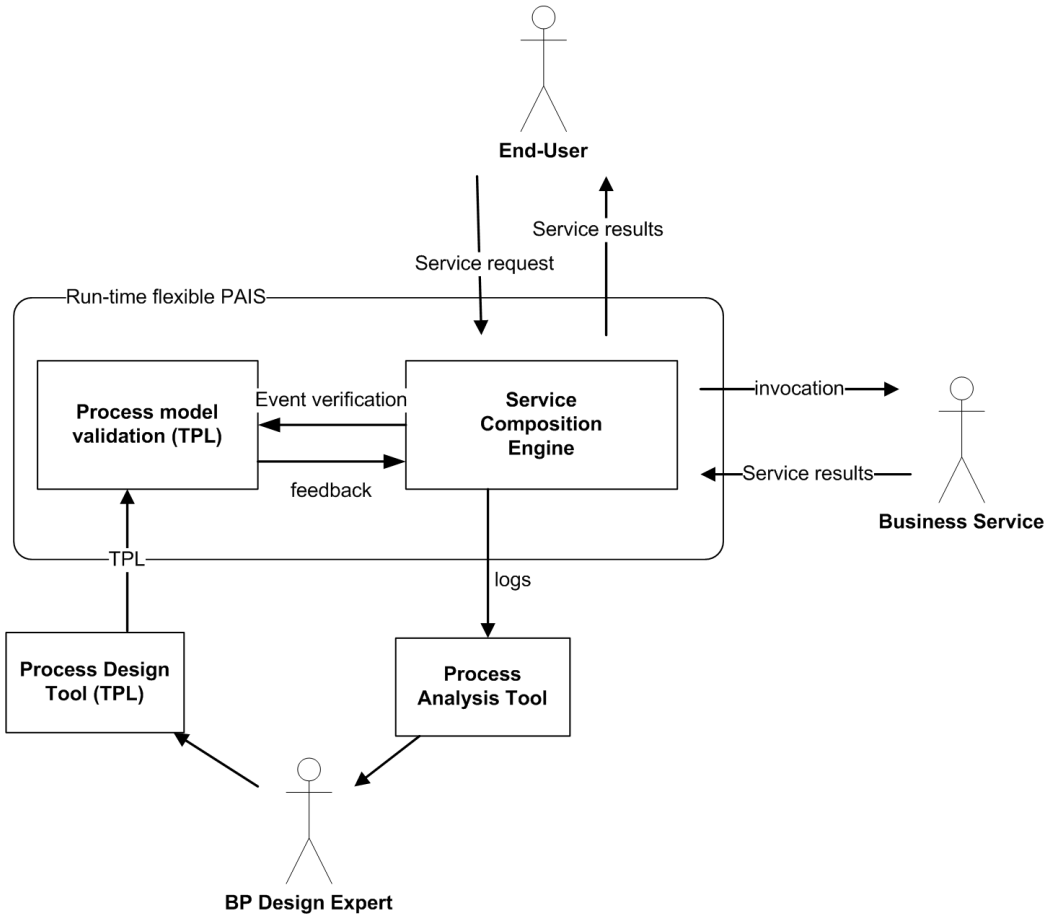


Figure 4.2: System Context of the TPL evaluation system

4.4.1 Actors

The actors depicted in figure 4.2 are described here.

End-User

The end-user is an active human participant in the business flow, possibly a client or employee. The goal of the end-user is to complete the business case he is involved in. In the case of example story 3 described in 4 this could be a patient in need of treatment.

Business Service

This user represents an invoked Web service to conduct the required tasks that correspond to the activities in a Business Process. The Web service should simply be seen as an interface to some automated agent, or a human participant. What the concrete Business Service user is depends on the business context. In the case of example story 1 described in 4 this could be the billing system being referenced to send an invoice for a placed order. The concrete goal of a Business Service depends on the domain obviously, but in general we can say it is to implement an activity participation by providing a service to some End-User(s).

BP Design Expert

A BP Design Expert is a human involved in BPM, in particular the (re)design of Business Processes. In the system context we enable the Design Expert to redesign processes while the system is in operation and to allow designed processes to take effect immediately in a fashion controlled by the BP Design Expert.

4.4.2 Involved systems

The systems depicted in figure 4.2 are described here.

Service Composition Engine

This system intermediates the interaction between users and services. It implements some form of service composition according to specifications provided by the Process Model Validation system. It needs to be aware of service providers and do record-keeping in order to provide information on process instances. Information on the execution behavior of users can be relayed to a Process Analysis Tool and to the Process Model Validation, which may use this information to determine the allowed actions of the execution participants. To summarize the Service Composition Engine enforces the Business Process according to information it gets from the Process Model Validation.

Process Model Validation (TPL)

This system is the system of interest for the thesis. It needs to make the decisions on which execution is allowed to do what. The model that provides the required information for this decision is a Temporal Process Logic (TPL) model. Also executions may be allowed to follow another process model while they are running, discontinue their work prematurely or deviate from the process. This behavior of execution is controlled by a provided TPL meta model.

Process Design Tool (TPL)

Temporal Process Logic (TPL) is designed to be a theoretically well-founded process language. For process design, even by an expert some other input format might be desired, possibly a visual representation and some automated form of support. The Process Design Tool provides this capability and is used to generate TPL models and meta models.

Process Analysis Tool

Running executions can be interesting to analyze if one is aiming to improve a process or decide a suitable migration strategy. A Process Analysis Tool provides analytics on the execution data provided by the Service Composition Engine.

4.4.3 Use Case description: process migration configuration

The flow of events in case the BP Design expert decides to change the policy on migrations is detailed in this section.

Table 4.2: Basic flow of process migration configuration

Step	Interaction	Actor
Preconditions:	- A set of executions **E** of a process model **P** is running - The process model **P** is already defined by the *BP Design Expert*	
1)	A new process model **P'** is defined	*BP Design Expert*
2)	A meta model is defined, describing constraints over the executions in E and any new executions of processes **P** and **P'**	*BP Design Expert*
3)	The enactment of the new meta model is confirmed	*BP Design Expert*
4)	The meta model changes the allowed behavior of executions in **E** and all new executions of **P** and **P'**	*System*

4.5 Research questions

The research questions for this thesis follow the research aim (section 4.1). Since the research aim is to investigate a configurable migration supporting PAIS, the research questions revolve around various aspects of the modeling and implementation of processes and meta processes.

Are process traces a sufficient description of run-time execution to evaluate TPL expressions?

In order to make migration decisions on instances the system needs to be aware of process instances as well as the processes. Various degrees of knowledge can be chosen with different results [44]. We need a model that provides the global state and history of the system at any moment in time. From this state representation process compliance and satisfaction should be verifiable.

Given a meta model formulated in TPL, how does the control over migrations compare to existing migration strategies and languages?

Existing approaches do have various ways to model what a modifications need to be performed during a dynamic change [24, 38, 27, **?**]. The policies describing how these changes should be handled however are currently defined only in the most basic way [6] and much expressive power to define constraints over the policies is not provided in current frameworks. The expectation is that TPL meta models as described in this thesis will provided more fine-grained control over dynamic changes.

Do *propagation*, *critical path length*, *compliancy* and *variability* notions provide discerning metrics for TPL (meta-)models?

Some of the metrics for processes are straightforward, others apply only to the process level or only to imperative approaches [34]. The metrics *propagation*, *execution length*, *compliancy coverage* and *uniformity* express some basic properties of models and meta models and might be of use to characterize these.

4.6 Research methodology

Our approach focuses specifically on declarative process modeling. In imperative process modeling the scenario of process migration has already been investigated deeply. In declarative approaches this scenario is implemented but in an inflexible way. Although a migration can be issued while instances are running, conficting instances just block the entire migration. One cannot specify a migration strategy or configure the migration in any way. The research goal is to use declarative modeling again to solve this inflexibility. To do so we need a declarative model of how instances behave with respect to processes; a declarative meta model First of we need a declarative language to model processes, for this we use the Temporal Process Logic or TPL for short [8]. Since TPL is designed for design-time variability, we will need to find a way to evaluate TPL formulas not over a process but over a running instance. Next the goal is to define meta models with TPL and evaluate these. When successful we should be able to use TPL-meta models to define a migration strategy. Finally we need to be able to quantify properties of the models and meta models that form a basis for designing meta models Since we will deal with the interpretation of logic expressions and are mainly interested in collection of metrics for different possibilities we have decided to build an experimental framework for TPL in Prolog. Prolog should be well suited for evaluating logic formulas resulting in a small code base of the implementation and searching a full problem space. The same prolog program can be used to generate test-data and verify specific executions.

4.6.1 TPL as a declarative process model

Although TPL was introduced in the related work, we need formal definitions of the semantics of the operators to be able to formalize the realization. The formal descriptions of operators are taken from [8].

Definition 7 (TPL operators)**.** *A process P can satisfy a TPL expression if there is a model $\mathcal{M} = \langle P, \nu \rangle$ where ν is a valuation function. The function $\nu(a)$ returns true if and only if a is executed in the current state of the model. Truth values are local to model states (i.e. activity nodes of the process). We use $a < b$ to denote that there is a path from a to b in the model. We use $a \ll b$ to denote that all paths from a to $\otimes$ in the model go through b.*

$$\mathcal{M}, x \models a \Leftrightarrow x \in \nu(a)$$
$$\mathcal{M}, x \models \rightarrow b \Leftrightarrow \forall \sigma \in \Omega_P : x \in \sigma \wedge \exists y \in \sigma : x <_\sigma y \wedge \mathcal{M}, y \models b$$
$$\mathcal{M}, x \models \rightsquigarrow b \Leftrightarrow \exists \sigma \in \Omega_P : x \in \sigma \wedge \exists y \in \sigma : x <_\sigma y \wedge \mathcal{M}, y \models b$$
$$\mathcal{M}, x \models \rightrightarrows b \Leftrightarrow \forall \sigma \in \Omega_P : x \in \sigma \wedge \exists y \in \sigma : x \ll_\sigma y \wedge \mathcal{M}, y \models b$$

The unary operators defined above have binary counter-parts as well, defined as:

$$a \rightarrow b \equiv a \wedge (\rightarrow b)$$
$$a \rightsquigarrow b \equiv a \wedge (\rightsquigarrow b)$$
$$a \rightrightarrows b \equiv a \wedge (\rightrightarrows b)$$

4.6.2 Model validation

To provide a flexible framework based on TPL we want to be able to determine if executions are allowed given a set of TPL expressions. For this to happen there must be a process P that satisfies the model and an execution graph E that is compliant to P. Validation of executions has been checked for FOL derivatives before. We follow the approach of [24], where LTL expressions are validated on traces. An important simplification made in [24] is that traces are limited to the finite domain, simplifying the automata required for validation. In our research we need formalization of traces and show correct validation of TPL expressions onto the traces.

4.6.3 Process quantification

Propagation is used as a notion in process migration literature, it describes how updates in process versions translate to the instance level. Structural similarity is a definition used in [34]. As is known within Business Process optimization, a high uniformity in processes, e.g. a low variability, can result in lower running costs, depending on the application field [45]. The metrics we will use are either based on BPM terminology (i.e. *Variability, Propagation, compliancy*) or on graph theory (i.e *critical path length*).

Chapter 5

Realization

To propose a solution for a configurable run-time migration framework we present an architectural view that follows the system context and requirements described above. From the architecture we go into detail on the TPL reasoning system, the subject of this thesis. We describe how the reasoning system operates and which models and algorithms lay at its foundation.

5.1 Envisioned framework architecture

Although the entire envisioned system as shown in this architectural view is not implemented and not all problems faced are the topic of this thesis. Still giving an overview of the entire architecture is important as it shows where a logic framework would fit into an operational PAIS and how this would solve the problem of having configurable migration models.

5.1.1 Dynamic process enactment and service composition

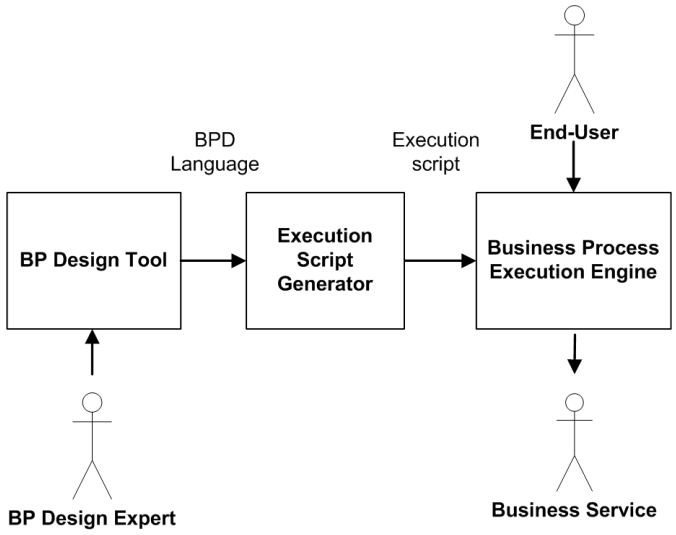
The foremost important architectural decision is to decide how the system manages the interaction of users with the business services that implement the activities within the Business Process.

Static service composition

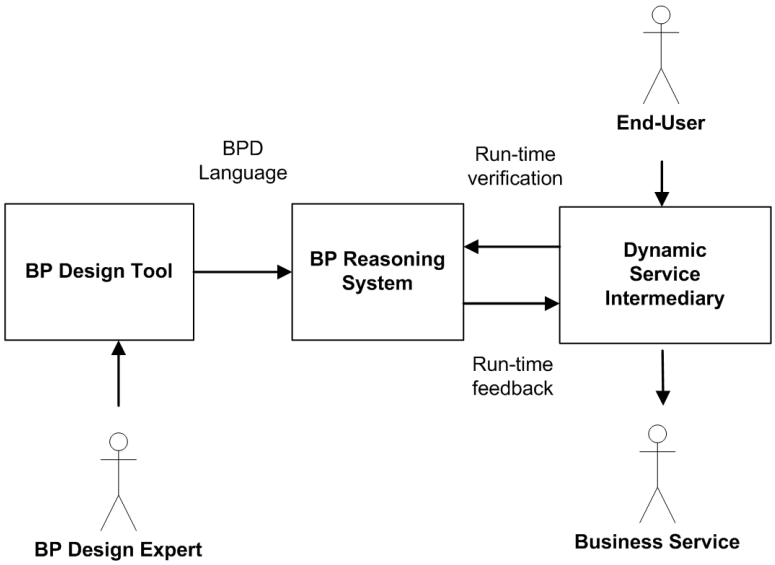
A typical architecture for service composition is to generate a script for service composition and use a web service execution engine to run that script. This is how for instance a BPEL engine operates. If changes are applied to the process definition the only possibility is to create a new execution script, and replace the current execution script by the new version. The execution engine can however not decide where it should continue in the new script or even if there is a corresponding state where it can resume. Instead the engine has to be restarted and progress is lost, thus process design is only supported compile-time and dynamic change is not supported. The static service composition scheme as used by classic PAIS is shown in figure 5.1(a).

Dynamic service composition

An alternative architecture for service composition is to verify the process when the web services are invoked. Each time the most up to date process description should be verified. This scheme as shown in figure 5.1(b) allows dynamic changes to processes to be handled



(a) Classic PAIS architecture with compile-time process design and run-time process enactment



(b) Alternative PAIS architecture with run-time process design and enactment

Figure 5.1: Different architectural schemes for service composition according to a process model specification.

without having to restart the system, we can say that this system supports run-time design updates and will be able to handle dynamic changes to processes.

5.1.2 System decomposition

Looking at the system context in figure 4.2 what is required to realize a PAIS that meets the demands is an integration of dynamic service composition and a TPL model validation subsystem. Figure 5.2 shows exactly that, the architecture of the proposed system consists of a Dynamic Service Intermediary system that handles on-the-fly web service selection. Process design can take place without requiring updates to the Dynamic Service Intermediary.

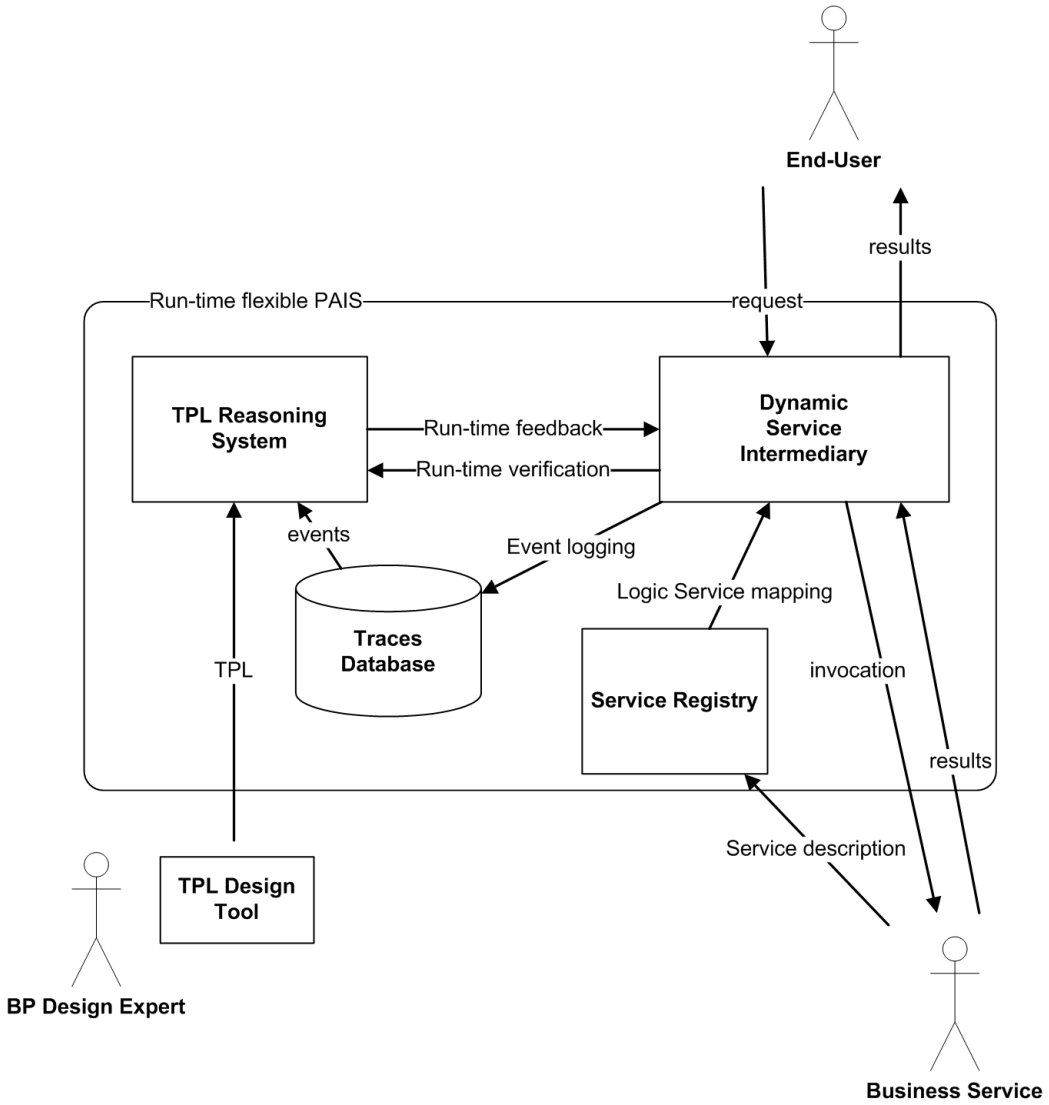


Figure 5.2: System architecture of the TPL evaluation system

Dynamic Service Intermediary

The Dynamic Service Intermediary functions as a brokering system for webservices. While it relays service invocations to selected Business Services the system knows about it also verified the actions with the TPL reasoning system and logs events.

Service Registry

The selected services have to be related to Business Processes by some mapping system. Active services can be registered dynamically so the system is aware of the services implementing activities in the Business Processes involved.

TPL Reasoning System

The TPL Reasoning system is responsible for validation of TPL models and meta models It decides which actions are restricted to the execution instances based on TPL sets that can be loaded and revised on-the-fly.

Traces Database

The event traces are stored in the Traces Database which can be accessed for any additional purposes, such as analysis of process execution and verification of processes by the TPL reasoning system.

5.2 Operation of the TPL Reasoning System

The TPL Reasoning system, as only part of the architecture described, will be actually realized. Its task is one of model validation. It needs to verify incoming events, provide a list of allowed steps to the user and provide an interface to manipulate the TPL models.

5.2.1 External interfaces

The external usage of the TPL Reasoning System can be separated into two parts, as shown in figure 5.3. The **TPLEventVerifier** interface provides actions for the Dynamic Service Intermediary and End-User side of the system, by providing event verification and providing lists of the allowed events for instances. Such a list of allowed events provide better usability than just giving rejection feedback if some disallowed events are performed. The **TPLModelRegistry** interface provides the BP Design Expert the means to modify TPL models.

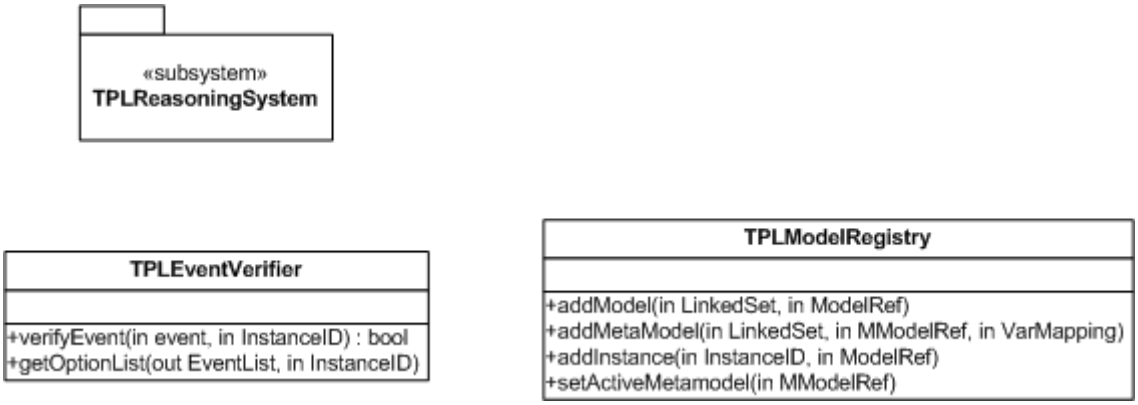


Figure 5.3: UML template for interaction with the TPL reasoning system

5.2.2 Internal operation

Internally we have chosen for a program logic implemented in an prototype fashion in Prolog. A listing of the Prolog implementation can be found in appending A. The program uses a typical design used in the logic programming paradigm, that of generate and test. The program generates possible executions and tests whether TPL-models apply. The implementation can be used both for testing possible execution as for control of externally determined executions. Where generation is convenient for creating lists of future options and for testing what-if scenarios, control of executions implements the verification and feedback interface to the dynamic service intermediary. Prolog allows these two use cases to be realized with the same implementation, as it support variable usage for both input as output. The task of the program is validating if an execution satisfies a TPL model and to verify if the complete set of executions validates a TPL meta model This is schematically

shown in figure 5.4. The same validation scheme is applied twice, one at the process level and once at the meta process level. The validation scheme is first to generate a trace and then to test if a TPL model satisfies this trace or not.

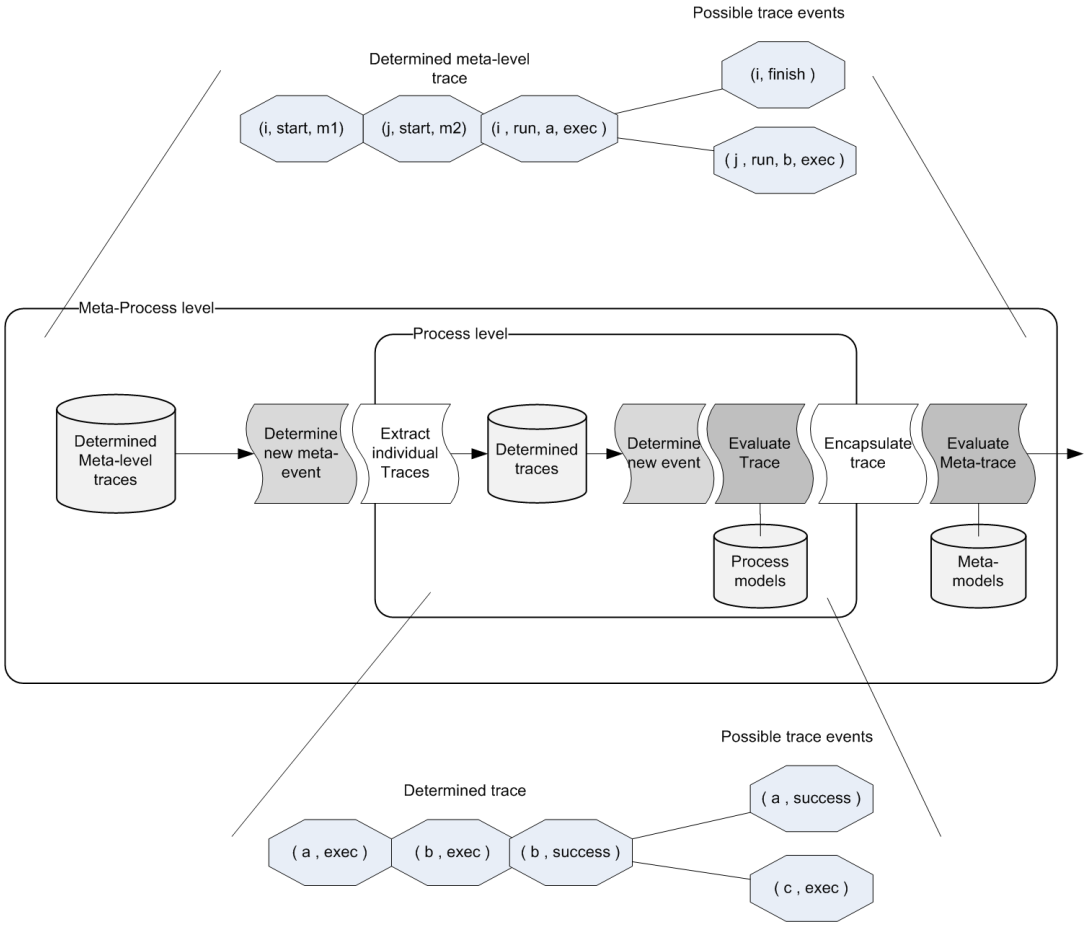


Figure 5.4: TPL reasoning system internal operation

5.3 The task of validating a TPL model

We define the task of validation as follows.

Definition 8 (Validation)**.** *Given a set of TPL-expressions E_{set} representing a process model and a trace t representing an execution. Determine if the execution of t satisfies all expressions in E_{set}. An execution graph $\sigma \in \Omega_P$ has a set of activity nodes A and edges T. The execution $\sigma = \langle A, T \rangle$ satisfies an expression $E \in E_{set}$ if and only if*

$$\exists\, \alpha \in A \;:\; \mathcal{M}, \alpha \models E$$

where σ can be obtained from the trace T by $\sigma = toExecutionGraph(T)$.

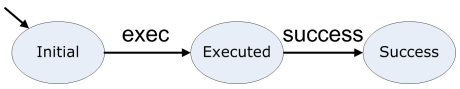
Definition 8 introduces traces, which are described in the next section. The function $toExecutionGraph(T)$ is described in algorithm 1.

5.4 Traces: a detailed description of run-time execution

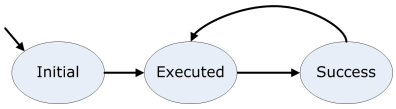
In a PAIS the actions taken by process instances are usually logged and on this log processes can be enforced. A process trace is a sequence of executions of activities. Process traces are typically encountered in run-time context instead of the execution graphs defined in chapter 2. Process traces as used in run-time context do have a correspondence with execution graphs as we will now show.

5.4.1 Modeling run-time execution

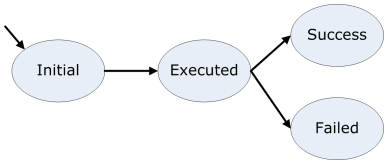
The execution of activities can be described by state transition diagrams as shown in figure 5.6. The most simple description given in the diagram just differentiates the begin and end events of activities. If we describe each activity in this form we immediately have a model for both parallel and sequential execution. Also we know if processes have already started, thus solving the "dangling state problem" encountered with process migration. If the success of activity A precedes the execution of an activity B, we have a sequential execution in which B follows A in time. If however the execution of one activity say C takes places after the execution of another, say D, but before the success of C we have parallel execution of C and D. We can expand this simple state diagram to allow re execution of activities and possibly failure of activities if we want, as shown in figure 5.6 as well. For simplicity we will assume the state machine diagram FSM_α shown in 5.5(a).



(a) The state machine used as a model of run-time execution: FSM_α



(b) Another state machine that models multiple executions of activities



(c) An alternative state machine that models failure of activities

Figure 5.5: Different simple state transition diagrams for activity execution

5.4.2 Creation of process traces

A trace consists of execution and success events for all the activities. Using Finite State Machine theory this can be formally described. We start with the automaton FSM_α over the alphabet $\Sigma = \{(\alpha, \texttt{exec}), (\alpha, \texttt{success})\}$ that accepts the event chain belonging to one activity α. The language accepted by FSM_α is a finite set that contains all intermediate states of execution of one activity. From FSM_α we can look at the traces of multiple activities, a set *Acts*. The set of possible execution traces for this set *Acts* is described by the language accepted by an automaton FSM_ρ. Where FSM_ρ is the composition of FSM_α machines for all $\alpha \in Acts$. This composition of automata is illustrated in figure 5.6. Note that since all activities are distinct the alphabets of the different FSM_αs are disjunct. This means that in the product of the automata is completely straightforward and every state in

the product is reachable. The language of FSM_ρ given a set $Acts$ is called $Span_{Acts}$, which contains each possible trace of the given activity set.

$$FSM_\rho(Acts) = \coprod_{a \in Acts} FSM_a$$

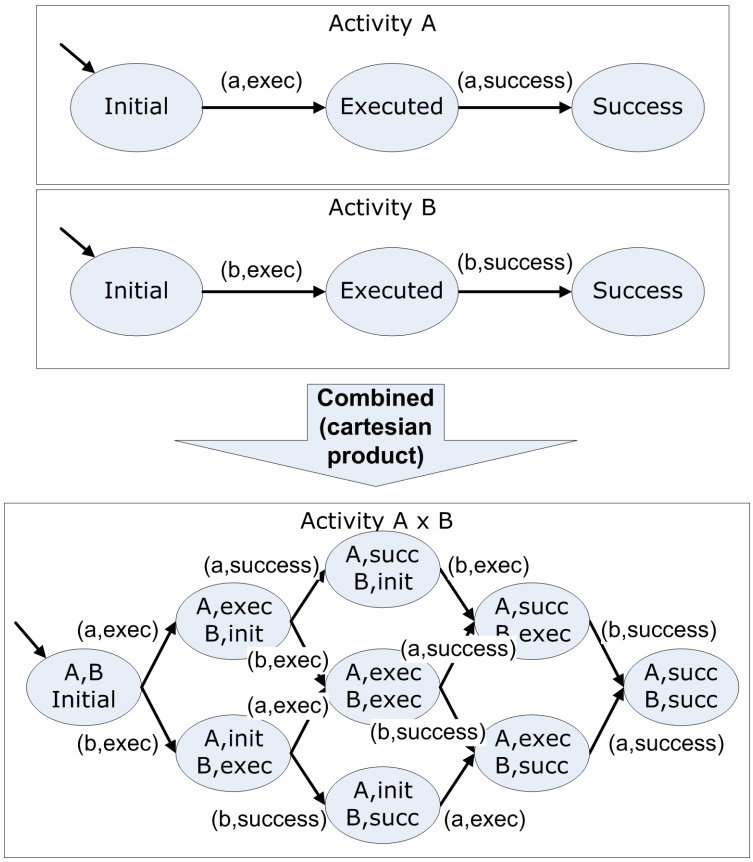


Figure 5.6: Example of FSM composition in the case of two activities

5.4.3 Traces related to execution paths

Suppose we have an activity p and q in an execution path and an arc from p to q. In this case we have sequential execution, meaning that p has finished before q begins. In the case of parallel execution this condition does not apply, and q may start before p has finished. In a trace $t \in Span_{Acts}$ we can say that q follows p if the sequential condition holds.

Definition 9 (sequential condition)**.** *If a trace t can be expressed in the following form, where $\{\alpha, \beta, \gamma\} \subset \Sigma^*$ are substrings of t:*

$$t = \alpha.(\mathtt{p}, \mathtt{success}).\beta.(\mathtt{q}, \mathtt{exec}).\gamma$$

We say q follows p. Equivalent to $q > p$ for the appropriate execution graph.

Note that traces are more fine-grained than execution paths, because of their differentiation between states of activities. This also means that the case of any parallel execution of

activities results in more than one possible trace for a given execution path, as events in the trace can be interleaved in various ways.

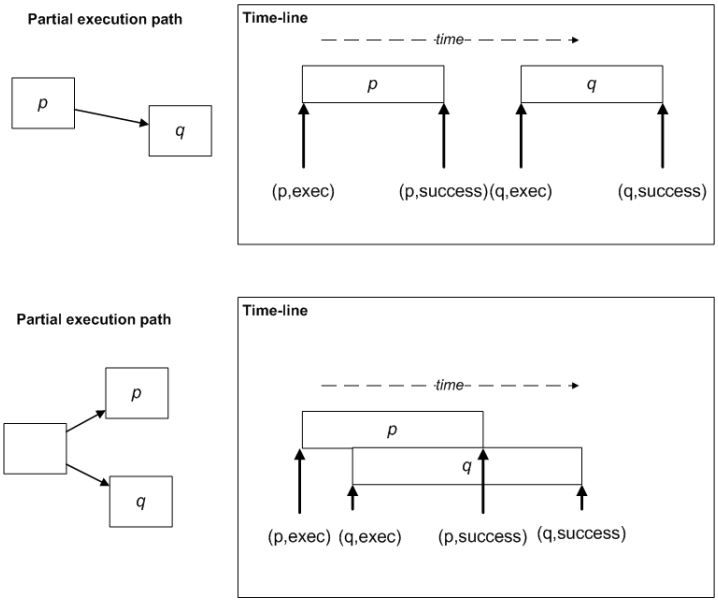


Figure 5.7: Upper part shows a sequential execution, below an example of parallel execution

Traces can be mapped to Execution Paths by construction. Given a trace $t \in Span_{Acts}$, we can create a unique $\sigma = \langle A, T \rangle$ in the following way. The set A is a subset of $Acts$ and consists of all activities that occur in t. For each pair $(x, y) \in t$, $x \in A$. For the transition set T we are interested in activities that happen sequentially.

Input: Trace string t
Output: Execution path $\sigma = \langle A, T \rangle$

$A := \{\odot\}$
$T := \varnothing$
repeat
  $(x, y) := head(t)$
  $t := tail(t)$
  if $y = success$ **then**
    $A := A \cup \{x\}$
  else
    for all $a \in A$ **do**
      $T := T \cup \{a \rightarrow x\}$
    end for
  end if
until $t = \epsilon$
$T :=$ *transitiveReduction(T)*

Algorithm 1: *toExecutionGraph* algorithm

The algorithm 1 uses the sequential condition to correctly add arcs for all sequentially executed activities. The algorithm processes all events from the trace in temporal order (i.e. first to last). Finished activities are stored in the set A, and from activities in this set arcs are created to all newly started activities. This does mean that the created path includes the full transitive closure of the execution path that is normally used, so we apply the *transitiveReduction* operation to get the desired execution path.

5.5 Validating TPL-expressions

The goal of the implementation is to apply the declarative models of TPL directly to the run-time process traces formulated in the previous section. In TPL we can differentiate two types of operators, the original logic operators we know from first order logic (FOL) and the added temporal operators. The logic operators and variables apply always to the current state of the model and may thus be simply evaluated onto the execution path or trace we have at the time. The temporal operators always have a part of the expression that should not be evaluated onto the current state but onto a state that follows. This strongly motivates a top-down approach in the semantic evaluation of expressions. The idea is that expressions when applied onto a state of execution given by a trace or its corresponding execution path yields a logical value and some subexpression(s) that should be evaluated in a later state. The sub-expressions do not need not be assessed at the given state, which can be exploited when evaluating the expressions top-down. The first task is splitting the TPL-expressions in a purely logic expression that concerns the current state and partial TPL-expressions that concerns the future. This can be achieved simply by considering the argument of the unary temporal-operators as future and for binary temporal operators taking the left-hand as present and right-hand as future.

5.5.1 Validation approach: step-by-step evaluation of partial expressions

Starting with an empty trace and a linked set of TPL-expressions, events can be progressively added to the trace as long as the set of expressions is updated accordingly. In order to provide a satisfying model for the linked set the linked set must be cleared entirely. Besides a set of expression that must be cleared, called the future goals, also a set of prohibited goals is maintained. Prohibited goals are goals that may never be fully cleared. The procedure of progressive evaluation is illustrated in figure 5.8. The steps are as follows.

1. Check whether the history trace t forms a stage transition with a new event e. This is the case when $stage(e) - stage(e.t) \neq \varnothing$. In the case of a stage transition, proceed with the other steps, otherwise accept the new event.

2a. The `Model State Expressions` are initialized with the linked set of TPL-expressions. In this step all expressions that apply to current states are removed from `Model State Expressions` and evaluated, this leads to added Future goals and possibly prohibited goals. In the case that an expression applies to a current state, evaluation must be successful, otherwise the trace is rejected.

2b. All `Future Goals` are evaluated. These should either have no effect or introduce replacing future and prohibited goals. The set of `Future Goals` must be empty to accept a trace.

2c. All `Prohibited Goals` are evaluated. These should either be false or introduce remaining future and prohibited goals. If a prohibited goal is evaluated to true and has no remaining future goals or prohibited goals to violate, the trace is rejected. The set of `Future Goals` may contain elements when a trace is accepted.

3. Finally the `Future Goals` and `Prohibited Goals` are merged with the results from step 2. If `Model State Expressions` and `Future Goals` are empty the trace is accepted as model of the linked set of TPL-expressions.

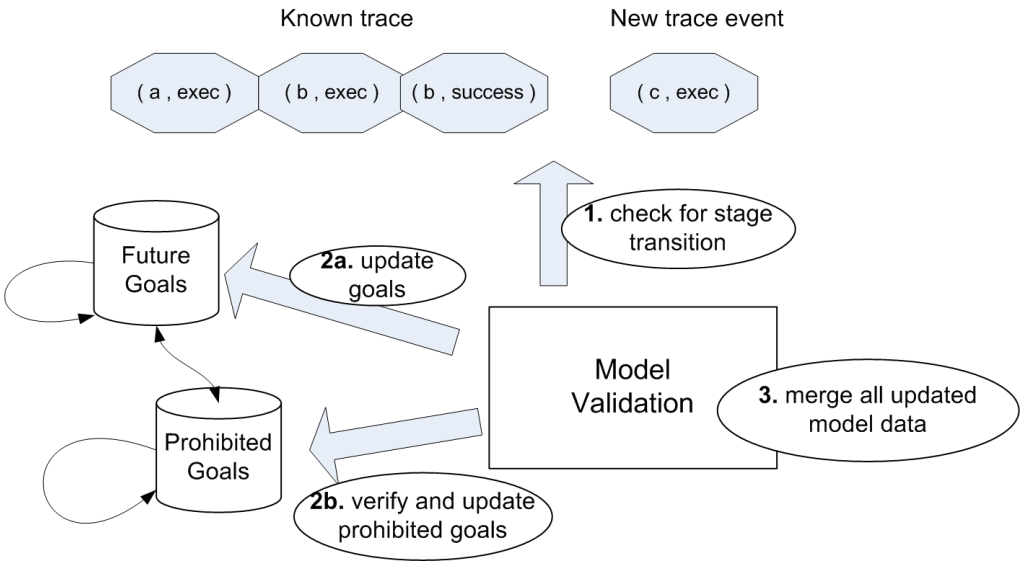


Figure 5.8: Informal schematic of model validation. When a trace causes a stage transition the set of expressions of the model is checked. This may result into new partial expressions entering the goal sets and possibly the creation of prohibited goals, by negation in the expressions.

Distinguishing steps of the execution: *stages* of an execution or trace

Having introduced the concept of top-down evaluation of TPL on the execution level, the first task is to define what is meant exactly by the current state of an execution. To avoid confusion with the many state transitions that are discussed in this thesis we call the current state of an execution its *stage*.

Definition 10 (Stage)**.** *For a given execution path its stage is simply the set of sink nodes (nodes with an outdegree of zero). This gives a set of concurrent activities that the execution is currently undertaking. TPL-variables are evaluated to true when their corresponding activity is true. Of a trace t the definition of a stage is the set of activities A for which:*

$$\forall\, a \in A \;:\; \neg\exists \langle b, s\rangle \in t : b \; follows \; a$$

In normal language, the stage set set A is the set of activities in the trace that have no sequential followers, so it only has recently executed activities in parallel.

The stage of an execution trace can be obtained by the stage algorithm, algorithm 2. Which traverses the trace and stores all concurrent executions along the way in a set Par. When an execution event is encountered all finished executions are removed from Par. Eventually the set Par is converted to the stage by removing the execution information.

1: **Input:** Trace string t
2: **Output:** set of activities S
3: $Par := \{(\odot, exec)\}$
4: $S := \varnothing$
5: **repeat**
6: $(x, y) := head(t)$
7: $t := tail(t)$
8: **if** $y = execs$ **then**
9: **for all** $(x1, success) \in Par$ **do**
10: $Par := Par - \{(x1, success)\}$
11: **end for**
12: **end if**
13: $Par := Par \cup \{(x, y)\}$
14: **until** $t = \epsilon$
15: **for all** $(x, y) \in Par$ **do**
16: $S := S \cup \{x)\}$
17: **end for**

Algorithm 2: Get the current stage of a trace: function *stage*

Stage transitions and stage division of a trace

We consider a stage changed, not just when the content of its set changes, because a stage may simply grow as more parallel activities are started, see figure 5.9, in which distinct stages are shown with rounded squares. Only when for an execution path e and its progression $E' = progress(E)$ the condition $stage(E) - stage(E') \neq \varnothing$ holds, we have a stage transition, and the function $next\_stage(E, E')$ holds true. In other words, $next\_stage(E, E')$ holds only when an activity in the stage of E has a successor in E': $\exists x \in E, y \in E' : x < y$. Algorithm 3 divides a trace into stages where-ever an event causes a stage transition. The algorithm simply iterates over the trace and determines stages in each step along the way, this is far from the optimal way, but the algorithm is shown for insight and correctness purposes.

1: **Input:** Trace string t
2: **Output:** a list of stages S_{list} (stages given as sets of activities)
3: $S_{list} := []$
4: $e := []$
5: **repeat**
6: $e' := e.head(t)$
7: $t := tail(t)$
8: **if** $next\_stage(e, e')$ **then**
9: $S := stage(e)$
10: $S_{list} := S_{list}.S$
11: **end if**
12: $e := e'$
13: **until** $t = \epsilon$

Algorithm 3: Divide a trace into separate stages: function *stage_division*

An example of stages to show its application for validation of TPL

Suppose a TPL-expression of the form $a \rightarrow (b \wedge c)$ is validated over an execution or trace in which an activity $x \in stage(E)$ in the execution graph E and $y \in stage(E')$ of a progressed graph E'=progress(E). Suppose we also know that $next\_stage(E, E')$ is true, $\mathcal{M}, x \models a$ and $\mathcal{M}, y \models b \wedge c$, we can use the stage transition to deduce that $\mathcal{M}, x \models a \rightarrow (b \wedge c)$. **Proof:**

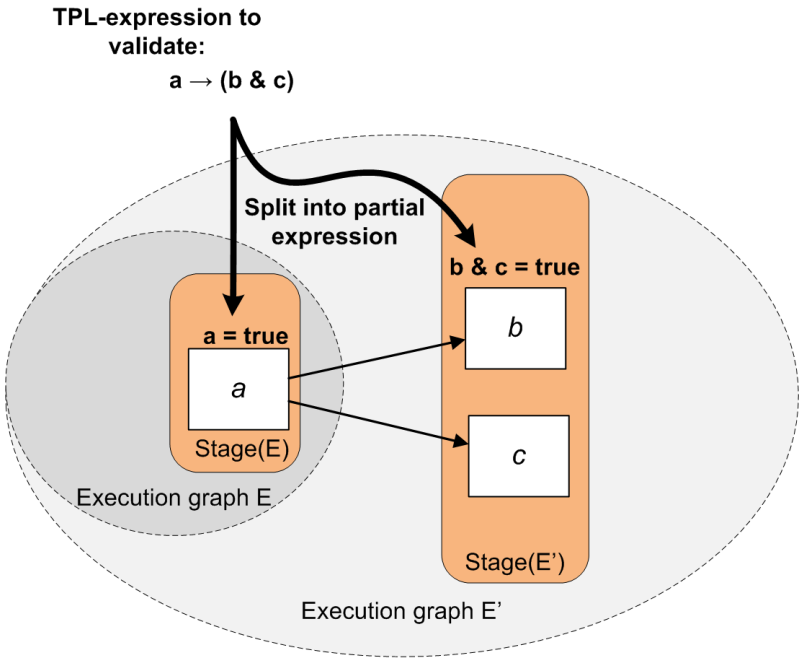


Figure 5.9: Example of the validation of a TPL-expression on an execution by validating partial expressions on consecutive steps of the execution. An execution E is shown that progresses to E'. On this execution $a \rightarrow (b \wedge c)$ is validated by validated a on E, and $b \wedge c$ on the progression E'.

Since $next\_stage(E, E')$ holds true, we know $\exists x \in stage(E), y \in stage(E') : x < y$. Another fact we can use is that if there is a $x \in stage(E)$ for which $\mathcal{M}, x \models a$, then actually $\forall i \in stage(E) : \mathcal{M}, i \models a$ as well, since the expression is valuated based on whether the activity is being executed at that moment, which it must be since the stage contains only parallel executions with x (we know this from the definition of a stage, definition 10). This means that there must exist an $i \in stage(E)$ for which $i > y$. And from $\mathcal{M}, i \models a$, $\mathcal{M}, y \models b \wedge c$ and $i > y$ we can apply the definition of TPL, to conclude that $\mathcal{M}, x \models a \rightarrow (b \wedge c)$.

5.5.2 Validation logic implementation

We will now describe the algorithm for validation in detail. We use a mathematical pseudo-code notation instead of actual prolog for readability and better understanding. The pseudo-code should be interpreted with a resolution mechanism to back-track over undetermined outcomes. At the top-level we start with a function $validate(E_{set}, t) \mapsto \{\texttt{true}, \texttt{false}\}$. $validate(E_{set}, t) = eval\_successive(E_{set}, \varnothing, S_{list})$ where $S_{list} = stage\_division(t)$ The function *validate* breaks the trace into stages, which the $eval\_successive(G, P, S_{list}) \mapsto \{\texttt{true}, \texttt{false}\}$ function handles in succession. This function terminates after the last stage from the list,

returning `true` only when the set of goals has been emptied by all the calls of *eval_stage*.

$$\begin{aligned}
eval_successive(G, P, []) &= \texttt{true} \text{ iff } G = \varnothing \\
eval_successive(G, P, []) &= \texttt{false} \text{ iff } G \neq \varnothing \\
eval_successive(G_N, P_N, [S|S_{Rem}]) &= eval_successive(G, P, S_{Rem}) \\
&\quad \text{iff } \langle G_N, P_N, \texttt{true}\rangle = eval_stage(S, G, P) \\
eval_successive(G_N, P_N, [S|S_{Rem}]) &= \texttt{false} \\
&\quad \text{iff } \langle G_N, P_N, \texttt{false}\rangle = eval_stage(S, G, P)
\end{aligned}$$

Next comes the more detailed $eval_stage$ function, which is responsible for updating the expression sets according to the stage encountered. It needs to process all goals from the goal set G and verify that the prohibited goals from P are not satisfied (at least not entirely).

$$\begin{aligned}
eval_stage(S, G, P) &= \langle G_N, P_N, R\rangle \\
&\quad \text{where } \forall g_x \in G : \langle G_x, P_x\rangle = eval_goal(g_x, S) \\
&\quad \text{and } \forall g_y \in P : \langle G_y, P_y\rangle = eval_goal(g_y, S) \\
&\quad \text{and } G_N = \bigcup G_x \text{ and } P_N = \bigcup P_x \cup \bigcup (G_y \wedge_a \neg_a P_y) \\
&\quad \text{if } \forall y : P_y \neq \varnothing \text{ , then } R = \texttt{true} \\
&\quad \text{else } R = \texttt{false}
\end{aligned}$$

Even more detailed is the function $eval\_goal(g, S) \mapsto \langle G_N, P_N\rangle$, almost at the lowest level this function discerns the truth value of a single expression. It tries to find a satisfying interpretion of the expression and provides the consequential partial expressions along with the result.

$$\begin{aligned}
eval_goal(g, S) &= \langle G_N, P_N\rangle \\
&\quad \text{if there is a positive evaluation } \langle G_N, P_N, \texttt{true}\rangle = eval(g, S) \\
&\quad \text{Any positive evaluation is chosen non-deterministically} \\
eval_goal(g, S) &= \langle \{g\}, \varnothing\rangle \\
&\quad \text{iff for all evaluations } \langle G_N, P_N, \texttt{false}\rangle = eval(g, S)
\end{aligned}$$

5.5.3 Recursive semantic evaluation of expressions

At the lowest level, the language constructs of TPL are interpreted by $eval(e, S) \mapsto \langle G_N, P_N, R\rangle$. The resulting implementation and the criteria the evaluation is based on are shown in table 5.1 for all language constructs of TPL.

Table 5.1: Evaluation of TPL expressions

TPL formula	**Satisfaction criteria**	**Logic program implementation (pseudo-code)**
e is a variable	There is an activity x in the execution for which $\mathcal{M}, x \models e \Leftrightarrow x \in \nu(e)$ We assume all variables map to the same activity symbol. Somewhere e must occur in the execution, in any stage.	$eval(e, S) = \langle \varnothing, \varnothing, \texttt{true}$ iff $e \in S$ $eval(e, S) = \langle e, \varnothing, \texttt{false}\rangle$ iff $e \notin S$
$\neg e$	$\mathcal{M}, x \models \neg e \Leftrightarrow \mathcal{M}, x \not\models e$ The expression e must not be satisfied by any activity x in the execution.	$eval(\neg e, S) = \langle P, G, \neg R\rangle$ where $\langle G, P, R\rangle = eval(e, S)$

TPL formula	**Satisfaction criterion**	**Logic program implementation (pseudo-code)**
$e1 \wedge e2$	$\mathcal{M}, x \models e1$ and $\mathcal{M}, x \models e2$ The expressions $e1$ and $e2$ must be satisfied at the same time in the execution.	$eval(e1 \wedge e2, S) = \langle G1 \cup G2, P1 \cup P2, true\rangle$ iff $R1 \wedge R2 = \texttt{true}$ $eval(e1 \wedge e2, S) = \langle \{e1 \wedge e2\}, \varnothing, false\rangle$ iff $R1 \wedge R2 = \texttt{false}$ where $eval(e1, S) = \langle G1, P1, R1\rangle$ and $eval(e2, S) = \langle G2, P2, R2\rangle$
$e1 \vee e2$	$\mathcal{M}, x \models e1$ or $\mathcal{M}, x \models e2$ Either of the expressions $e1$ and $e2$ must be satisfied.	$eval(e1 \vee e2, S) = eval(e1, S)$ or $eval(e1 \vee e2, S) = eval(e2, S)$
$\rightarrow e$	There is an activity y for which $\mathcal{M}, y \models e$ and $x > y$ where x is an activity that is currently executed.	$eval(\rightarrow e, S) = \langle \{e\}, \varnothing, R\rangle$ where $R \in \{\texttt{true}, \texttt{false}\}$
$e1 \rightarrow e2$	There are activities x and y in the execution such that $\mathcal{M}, x \models e1$ and $\mathcal{M}, y \models e2$ and $x > y$. Also the binary $\rightarrow$ operator is required to be right-associative for correct parsing.	$eval(e1 \rightarrow e2, S) = \langle \{e2\}, \varnothing, \texttt{true}\rangle$ iff $eval(e1, S) = \langle \varnothing, \varnothing, \texttt{true}\rangle$ $eval(e1 \rightarrow e2, S) = \langle \{e1 \rightarrow e2\}, \varnothing, \texttt{false}\rangle$ iff $eval(e1, S) = \langle \varnothing, \varnothing, \texttt{false}\rangle$
$\rightrightarrows e$	There are activities x and y in the execution such that $\mathcal{M}, y \models e$ and $x \gg y$. This condition means that for all activities x_c concurrent to x it is true that $x_c > y$.	$eval(\rightrightarrows e, S) = \langle \{\underset{S}{\rightrightarrows} e2\}, \varnothing, R\rangle$ iff $has\_merged(S) = \texttt{false}$ where $R \in \{\texttt{true}, \texttt{false}\}$ $eval(\rightrightarrows e, S) = \langle \{e\}, \varnothing, R\rangle$ iff $has\_merged(S) = \texttt{true}$ where $R \in \{\texttt{true}, \texttt{false}\}$ $eval(\underset{S2}{\rightrightarrows} e, S) = \langle \{\underset{S2}{\rightrightarrows} e\}, \varnothing, \texttt{true}\rangle$ iff $has\_merged(S2) = \texttt{false}$ $eval(\underset{S2}{\rightrightarrows} e, S) = \langle G, P, R\rangle$ iff $has\_merged(S2) = \texttt{true}$ where $eval(e, S) = \langle G, P, R\rangle$
$e1 \rightrightarrows e2$	There are activities x and y in the execution such that $\mathcal{M}, x \models e1$ and $\mathcal{M}, y \models e2$ and $x \gg y$. Also the binary $\rightrightarrows$ operator is required to be right-associative for correct parsing.	$eval(e1 \rightrightarrows e2, S) = \langle \{\underset{S}{\rightrightarrows} e2\}, \varnothing, true\rangle$ iff $eval(e1, S) = \langle \varnothing, \varnothing, \texttt{true}\rangle$ and $has\_merged(S) = \texttt{false}$ $eval(e1 \rightrightarrows e2, S) = \langle \{e2\}, \varnothing, \texttt{true}\rangle$ iff $eval(e1, S) = \langle \varnothing, \varnothing, \texttt{true}\rangle$ and $has\_merged(S) = \texttt{true}$ $eval(e1 \rightrightarrows e2, S) = \langle \{e1 \rightrightarrows e2\}, \varnothing, \texttt{false}\rangle$ iff $eval(e1, S) = \langle \varnothing, \varnothing, \texttt{false}\rangle$

TPL formula	Satisfaction criterion	Logic program implementation (pseudo-code)
$\rightsquigarrow e$	There is an execution for which there are activities x and y in the execution such that $\mathcal{M}, y \models e$ and $x > y$, but it is not necessarily this execution, possibly another branch in the process. Goals of this form are handled by preprocessing the goal sets, replacing $\rightsquigarrow$ by $\vee$ operators and merging goals.	$preprocess(G_1, G_2) = \forall g \in G_1 : g = (\rightsquigarrow e) \Rightarrow e \in G_L \wedge g \neq (\rightsquigarrow e) \Rightarrow g \in G_R$ and $combine\_by\_or(G_L, G_O)$ and $G_2 = G_O \cup G_R$
$e1 \rightsquigarrow e2$	There is an execution for which there are activities x and y in the execution such that $\mathcal{M}, x \models e1$ and $\mathcal{M}, y \models e2$ and $x > y$, but it is not necessarily this execution, possibly another branch in the process. Also the binary $\rightsquigarrow$ operator is required to be right-associative for correct parsing.	$eval(e1 \rightsquigarrow e2, S) = \langle\{\rightsquigarrow e2\}, \varnothing, \texttt{true}\rangle$ iff $eval(e1, S) = \langle\varnothing, \varnothing, \texttt{true}\rangle$ $eval(e1 \rightsquigarrow e2, S) = \langle\{e1 \rightsquigarrow e2\}, \varnothing, \texttt{false}\rangle$ iff $eval(e1, S) = \langle\varnothing, \varnothing, \texttt{false}\rangle$

5.5.4 Semantic evaluation challenges

Having shown the implementation of model validation, we will now examine some properties of TPL that complicate the validation process and have led to the implementation.

TPL expressions may lead to multiple future goals

The composition of operators in TPL-expressions may involve multiple sub-expressions that involve temporal operators. A simple example shown in figure 5.10 is $a \rightarrow b \wedge a \rightarrow c$. This statement is not equivalent to $a \rightarrow (b \wedge c)$. The first states that a state a is in the model and this state leads to a state where b is true and to a state where c is true. Not necessarily the same state, whereas in the second statement the state a does lead to the state in which $b \wedge c$ holds.

TPL variables are not the same as activities

Although for convenience we may use variables and activities interchangeably, the meaning of having a variable that is asserted to true, is not the same as saying one activity follows the other, as in the sequential execution definition. This makes TPL-expressions of this form counter-intuitive, as $a \rightarrow a$, may not seem to make much sense, but still has many ways to be fulfilled, as shown in figure 5.11. All that is required is a transition in a parallel branch. We just have a stage which contains the activity corresponding to the variable that was used, we can not claim that in a next stage the activity is completed. In the case of the stronger $\rightrightarrows$

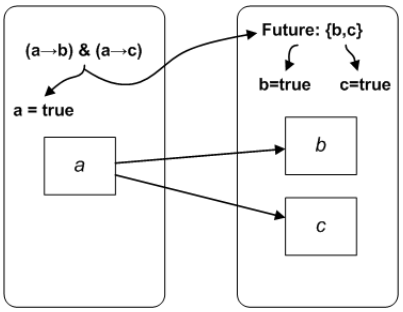


Figure 5.10: An example of an expression that leads to multiple partial goals for the future.

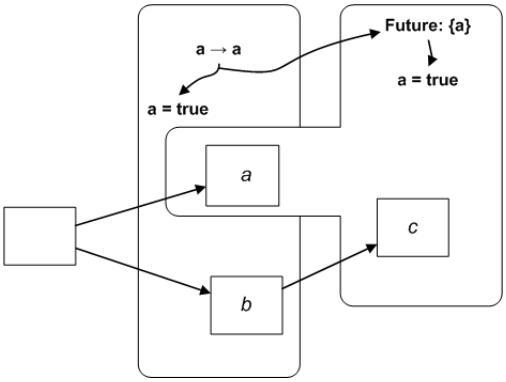


Figure 5.11: A counter-intuitive example of a TPL validation of $\mathcal{M}, b \models a \rightarrow a$

operator this claim does hold however, but then is more stringent than intuitively expected as all parallel branches must lead to the subsequent state.

Interpretation of negation introduces prohibited goals

The negation operator $\neg$ from FOL may simply be applied to variables and purely subexpressions, in which case the semantics are easily understood. $\mathcal{M}, a \models \neg b$ for example implies that when there is an activity a, the stage does not include activity b. However the negation may also be applied to subexpressions that include temporal operators, such as the example $a \wedge \neg \rightarrow b$, as shown in figure 5.12. In this construction there may not follow a state in which b holds, so the condition $b = false$ must be checked in every subsequent stage.

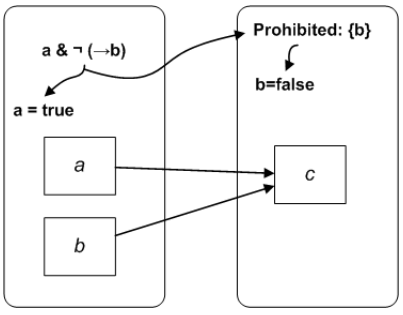


Figure 5.12: An example of prohibited goal validation.

Interpretation of $\vee$ causes branching

The $\vee$ operator in TPL makes future goals non-deterministic. In a given stage it is not certain which branch if any the execution may satisfy in the future. In the implementation

of the evaluation we must consider all branches until the outcome is determined. Figure 5.13 shows how the $\vee$-operator may lead to multiple future goal branches. Prolog's resolution mechanism already supports branching as it searches the solution-space, in imperative languages the branches have to be maintained in some explicit tree structure during evaluation.

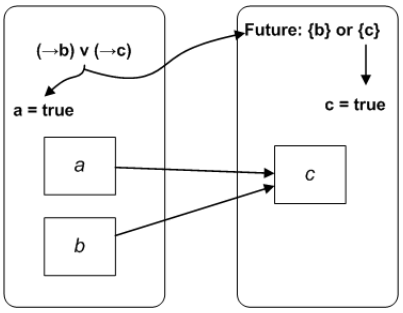


Figure 5.13: An example of branching in the validation, introduced by a single $\vee$ operator. At the first stage the branch that leads to a satisfying result cannot be determined.

Interpretation of $\rightsquigarrow$ means branching of the future goals

The $\rightsquigarrow$ operator in TPL also makes future goals non-deterministic. It states that an execution may either fulfill the goal given by the right hand of the operator or take another path indicated by an another part of the expression. We resolve this by regarding this problem as an conjunction of future goals, much like the $\vee$ operator, only this time applied to future goals. The conjunction of future goals is solved in a preprocessing step of the evaluation.

Supporting the $\rightrightarrows$ operator

The $\rightrightarrows$ implies that of a current state x all execution paths must lead to the state y satisfied by the right handed argument. All activities in the stage y must follow the activities in x. In a trace t this means that all activities that are currently active in a state x must succeed before y. This can be solved by making the activities that must finish explicit in the goal, by introducing a language construct $succeed(S)$ where S is a list of the activities currently in execution.

5.6 Towards configurable process meta models

Thus far process traces and the machines used to generate process traces have been treated and we have explained how TPL-expression can be evaluated over a trace by determining stages and stage transitions. The next goal is to apply the scheme of generation and evaluation again, but this time on a higher level. On the process level the process is a composition of activities and the flow of activities was controlled by a TPL-model. On the meta process level the meta process is a composition of executions and we control the flow of process models by a TPL-meta model In this section we show how meta process traces can be generated and also verified. We also show how TPL-expressions can be translated to the meta process level and introduce a more convenient choice of variables to express useful meta models.

5.6.1 Meta process level trace generation

On the process level we introduced $FSM_\rho(Acts)$ over the alphabet $\Sigma_\rho = \bigcup_{a \in Acts} \{(a, \texttt{exec}), (\texttt{a}, \texttt{success})\}$. On the meta process level we will have execution instances $i \in I$ that can be in a state $t \in \{\texttt{new}, \texttt{running}, \texttt{finished}, \texttt{aborted}\}$ as recapped from chapter 2 following some process p. To model the meta process level process we use the FSM_ρ and relabel the transition edges using the function $wrap\ \Sigma_\rho \mapsto \Sigma_{run}$, obtaining a machine FSM_{run} over a set activities $Acts$, an instance i and a process $\pi \in P$.

$$wrap(\langle a, state \rangle) = (i, \texttt{run}, \langle \texttt{a}, \texttt{state} \rangle, \pi)$$

Note the similarity to transitions of the PAIS dynamic system described in chapter 2. The edges of the automaton FSM_{run} correspond to transition 2.2 of the transition function T in chapter 2.2.2.1. From this automaton, we can continue to add the remaining transitions forming the automaton model of a process instance FSM_{inst}. The state machine for one instance is build up from relabeled activity span machines FSM_ρ for each process identifier in the set P.

- $FSM_{run}(Acts, i, \rho) = \langle \Sigma_r, S_r, sr_0, \delta_r, F_r \rangle$ with variables $Acts$, the set of activities, i, the instance identifier, and $\rho \in P$, a process.
- $FSM_{inst}(Acts, i, P) = \langle \Sigma_i, S_i, si_0, \delta_i, F_i \rangle$ with variables i, the instance identifier, and P, a set of processes.
- $\Sigma_i = \Sigma_r \cup \bigcup_{\rho \in P} \{\langle i, \texttt{start}, \rho \rangle, \langle \texttt{i}, \texttt{migrate}, \rho \rangle, \langle \texttt{i}, \texttt{finish}, \rho \rangle, \langle \texttt{i}, \texttt{abort}, \rho \rangle\}$
- $S_i = \bigcup_{\rho \in P} S_{run}(Acts, i, \rho) \cup \{si_0, s_{aborted}, s_{finished}\}$
- $\delta_i = \bigcup_{\rho \in P} \delta_r(Acts, i, \rho) \cup \delta_s \cup \delta_m \cup \delta_a \cup \delta_f$
- $\forall_{\rho \in P} si_0 \underset{\delta_s \langle i, \texttt{start}, \rho \rangle}{\longrightarrow} s_0(Acts, i, \rho)$
- $\forall_{\rho, \phi \in P} \forall_{s \in S_r} \rho \neq \phi \wedge \langle s(Acts, i, \rho) \underset{\delta_m \langle i, \texttt{migrate}, \phi \rangle}{\longrightarrow} s(Acts, i, \phi)$
- $\forall_{\rho \in P} \forall_{s \in S_r} s \underset{\delta_f \langle i, \texttt{finish}, \rho \rangle}{\longrightarrow} s_{finished}$
- $\forall_{\rho \in P} \forall_{s \in S_r} s \underset{\delta_f \langle i, \texttt{abort}, \rho \rangle}{\longrightarrow} s_{aborted}$

As with the composition of FSM_ρ from FSM_α using a Cartesian product of FSMs we can now compose the meta process execution state machine FSM_{meta} from the composition of FSM_{inst} by a product of the state machines for all execution instances of a set E. The resulting language is the meta process trace language $Span_{Meta}$. A meta model is simply an accepted subset of $Span_{Meta}$ by the meta model evaluation.

$$FSM_{meta}(Acts, E, P) = \prod_{i \in E} FSM_{inst}(Acts, i, P)$$

5.6.2 TPL-meta model constructs

Since on the meta process level we have defined traces corresponding to executions, just as on the process-level, we can again evaluate linked sets of TPL-expressions. In this section we first show the limitations of using a straightforward TPL-model on the meta model In TPL, expressions have variables that are asserted for a given set of activities being executed. $\mathcal{M}, a \models\!\!\rightarrow b$ is an example. When the execution is in a certain state, the truth value of the variable depends on whether a certain activity is being executed. For the meta model we model the execution of process instances, so we can analogue to process-level models choose variables for each process instance and create an appropriate TPL linked set. Suppose we

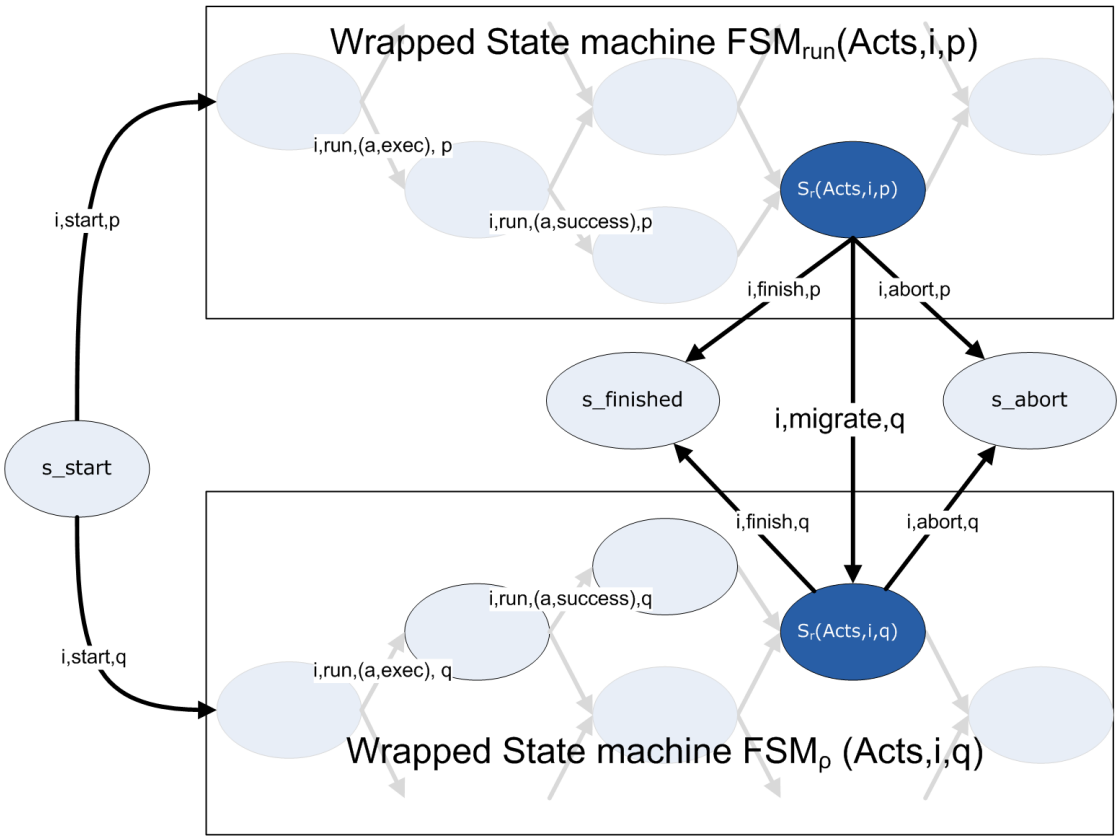


Figure 5.14: An illustration of the construction of FSM_{meta}. It shows how different state machines of executions, are wrapped with an execution and a process identifier. Migration transitions are added that simply go to a state from the execution level machine wrapped with a different process identifier. Other meta process level transitions are added as well to define starting, finishing and aborting the execution.

have an instance i and j, we can make for example an expression $\mathcal{M}, i \models\!\!\rightarrow j$. meta process level models of this form have some serious limitations:

(1) Variables correspond to individual instances, this is completely static. Introducing a new instance means revising the meta model

(2) Variables now only express the execution of instances, this is not interesting as it does not hold any qualitative information on the execution. In particular this is true for:

(2a) We have no expressive power to control which process or process version is executed by an instance.

(2b) We have no expressive power over process migration.

(2c) We have no expressive power over compliancy to a process.

Introducing dynamic predicates to refer to instances

To resolve limitation (1) we introduce dynamic predicates that can be used as 'wild card' for instances of the process, as a convention we will use upper-case symbols to denote dynamic predicates. An example of a TPL meta model expression then would be $\mathcal{M}, X \models\!\!\rightarrow Y$, we define the interpretation of this expression as follows. Given a set of execution instances E and an expression i of the form $\mathcal{M}, X \models T$ where X is a dynamic predicate and T is a TPL sub-expression containing a set of occurring dynamic predicates P, possibly having $X \in P$. This expression i is satisfied if a replacement set of conventional expressions I'

is satisfied where $\forall e \in E : \langle \mathcal{M}, e \models T' \rangle \in I'$. T' is obtained by taking T and applying a complete mapping $P \to E$ creating T', an expression not using dynamic predicates but actual instances bound to any known instance. Of course if $X \in P$ then its binding is already determined as e. The bindings may be selected in any way that allows satisfaction of the complete set I'. To summarize, a dynamic predicate in the left hand argument is interpreted for all instances, and dynamic predicates in the right-hand of the expression are bound to any one satisfying instance.

Introducing functors to obtain truth values

To resolve limitation (2) we introduce functors whose truth value depends on the argument. This allows fine-grained expression of qualitative aspects of executions. This does not change the semantics of TPL, it simply introduces different variables to evaluate truth values. The functors are simply boolean variables that take a number of arguments from the domain. Functors do not raise the complexity of the language. The functors we introduce are listed in table 5.2.

Table 5.2: TPL meta model functors

Functor	**Semantics**
process(X,P)	true iff instance X is running process model P
migrate(X)	true iff instance X is has just taken a migration transition
compliant(X)	true iff instance X can satisfy the model it is running from its current state
start(X)	the state where instance X starts to run a process
finish(X)	the state where instance X halts

Chapter 6

Evaluation and results

Having discussed the realization of a TPL model validation system that operates on process traces in the previous chapter, we now present the results of the actual prolog implementation. The program is mainly evaluated for its expressiveness. Therefore this chapter provides examples of models and meta models The example models create validation outcomes that are characterized by some metrics of which the results are given. We start by giving the process level results and then move on the meta process level results.

6.1 Results of trace validation with TPL

Trace validation has been successfully run on using the prolog implementation. The models themselves are not very interesting as such but will be used later by the meta models.

6.1.1 A readable trace notation

The traces presented in chapter 5 although being useful for formalization do not provide a reader friendly notation for presenting results. So for the sake of readability all execution tokens, of the form $\langle a, \texttt{exec}\rangle$, are replaced by `<a` and $\langle a, \texttt{success}\rangle$ tokens are replaced by `a>`. For example the trace $t = \{\langle a, \texttt{exec}\rangle, \langle a, \texttt{success}\rangle, \langle b, \texttt{exec}\rangle, \langle c, \texttt{exec}\rangle, \langle b, \texttt{success}\rangle, \langle c, \texttt{success}\rangle\}$ is depicted as:

```
<a a> <b <c b> c>
```

6.1.2 Process level metrics

A process can be characterized by some metrics. We propose the use of the following metrics shown in table 6.1. The notions of the metrics are often described, but usually not tied to quantification. Here we do show some simple way to quantify a process and use this to characterize and compare processes.

Table 6.1: Metrics at the process level

Metric	Definition
Critical path length	The critical path length is determined by the minimum number of stage transitions found in all traces that satisfy the model. This can be interpreted as determining the length of execution of the process. Execution length gives a metric that corresponds to how fast executions of the model can be finished, under the assumption that all activities correspond to one time unit and maximal parallelization is accomplished.
Compliant fraction	The fraction of traces that can be progressed to a trace that satisfies the model. We are determining which of all possible traces correspond to a partial execution of the process. The set of possible traces is the trace language Act_{span}, for just three activities this is already 271 possible traces. The complement are traces that violate expressions found in the model. The compliant fraction thus gives a metric for how restrictive a model is.
Variability	The number of different traces that satisfies the model. These different paths described variants of the execution.

6.1.3 Process simulation results

We have used TPL to describe some small but not entirely trivial cases of process models. We have noticed that even when just using three activities a, b and c a great deal of options are possible.

Table 6.2: Test set of Business Process descriptions modeled by TPL.

Model ID	TPL linked-set	Satisfying traces of the model	Critical path length	Variability	Compliant fraction
P1	$\mathcal{M}, a \models \rightrightarrows b$ $\mathcal{M}, b \models \rightrightarrows c$ $\mathcal{M}, c \models c$	<a a> <b b> <c c>	3	1	7 / 271
P2	$\mathcal{M}, a \models \rightrightarrows c$ $\mathcal{M}, c \models \rightrightarrows b$ $\mathcal{M}, b \models b$	<a a> <c c> <b b>	3	1	7 / 271
P3	$\mathcal{M}, a \models \rightrightarrows (b \wedge c)$ $\mathcal{M}, b \models b$ $\mathcal{M}, c \models c$	<a a> <b <c b> c> <a a> <b <c c> b> <a a> <c <b b> c> <a a> <c <b c> b>	2	4	15 / 271

$\mathbf{P_4}$	$\mathcal{M}, a \models \rightrightarrows b \wedge \rightrightarrows c$ $\mathcal{M}, b \models b$ $\mathcal{M}, c \models c$	<a a> <b b> <c c> <a a> <b <c b> c> <a a> <b <c c> b> <a a> <c <b b> c> <a a> <c <b c> b> <a a> <c c> <b b>	2	6	21 / 271

It can be seen from table 6.2 that the models $\mathbf{P_1}$ and $\mathbf{P_2}$ are highly restrictive, only having one satisfying trace. Whereas $\mathbf{P_3}$ is less restrictive, allowing parallel execution, and $\mathbf{P_4}$ being even less restrictive, allowing parallel execution but not forcing it. We can also use the same metrics to compare processes. Instead of determining compliant fractions on the full set of generated traces, we can use the compliant traces of one process and check compliancy with the other model.

Table 6.3: Compliancy overlap of processes

	$\mathbf{P_1}$	$\mathbf{P_2}$	$\mathbf{P_3}$	$\mathbf{P_4}$
$\mathbf{P_1}$	7/7 **(100%)**	3/7 **(43%)**	4/7 **(57%)**	7/7 **(100%)**
$\mathbf{P_2}$	3/7 **(43%)**	7/7 **(100%)**	4/7 **(57%)**	7/7 **(100%)**
$\mathbf{P_3}$	4/15 **(27%)**	4/15 **(27%)**	15/15 **(100%)**	15/15 **(100%)**
$\mathbf{P_4}$	7/21 **(33%)**	7/21 **(33%)**	15/21 **(71%)**	21/21 **(100%)**

In table 6.3 we show relative compliance, meaning that when an execution is compliant to one process it is also compliant to the other. The right-most column of $\mathbf{P_4}$ shows that this process accepts all traces compliant to any of the other processes. We can say that $\mathbf{P_4}$ contains all the processes $\mathbf{P_1}$ to $\mathbf{P_4}$. $\mathbf{P_1}$ and $\mathbf{P_2}$ show simply permutated values, as is expected since their model definition is also just an exchange of a with b.

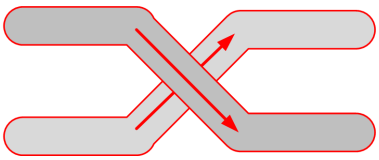
6.2 TPL meta model configuration and evaluation

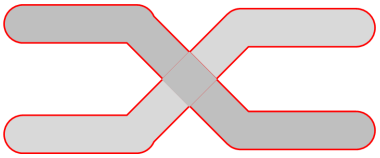
As meta models are linked to models it is hardly possible to provide metrics that are independent of the underlying processes. Also such metrics will not be very relevant in practice as the specific metrics that apply to the situation at hand will be most interesting. As stressed before meta models can not be optimized in general but instead constraints depend on the context of the problem.

6.2.1 Building meta models with TPL

First of all we have to invent some sensible meta models to use for testing the method. Here the basic migration strategies give some guidance, we can choose to have existing instances proceed as usual, or be transfered, or in extreme cases be restarted or even aborted. Table 6.4 shows a listing of possible meta models and their meaning. Note that this is not an exhaustive list.

Table 6.4: Listing of meta models, providing a short textual description, a graphic representation similar to figure 4.1 in chapter 4;old process is in light Grey, new process in darker Grey and areas excluded are in red.

ID	Meta model-description	Graphic meta model representation and appropriate TPL linked-set
$\mathbf{M_1}$	**no initialization of v2 and no transfer to v2**	 $\mathcal{M}, start(X) \models process(X, v1) \wedge \neg \rightarrow migrate(X)$ $\mathcal{M}, finish(X) \models completed(X)$
$\mathbf{M_2}$	**no initialization of v2 but optional transfer to v2**	 $\mathcal{M}, start(X) \models process(X, v1) \wedge$ $(\neg \rightarrow migrate(X)$ $\vee$ $\rightarrow (migrate(X) \wedge \neg \rightarrow migrate(X)))$ $\mathcal{M}, finish(X) \models completed(X)$
$\mathbf{M_3}$	**optional initialization of v2 and forced transfer to v2**	 $\mathcal{M}, start(X) \models process(X, v1) \rightarrow$ $(migrate(X) \wedge \neg \rightarrow migrate(X))$ $\vee$ $(process(X, v2) \wedge \neg \rightarrow migrate(X))$ $\mathcal{M}, finish(X) \models completed(X)$
$\mathbf{M_4}$	**optional initialization of v2 but no transfer to v2**	 $\mathcal{M}, start(X) \models \neg \rightarrow migrate(X)$ $\mathcal{M}, finish(X) \models completed(X)$

ID	Meta model-description	Graphic meta model representation and appropriate TPL linked-set
$\mathbf{M_5}$	**optional initialization of v2 and optional transfer to v2 (one-time only) with forced compliance**	 $\mathcal{M}, start(X) \models (process(X, v1) \rightarrow (migrate(X) \wedge \neg \rightarrow migrate(X)) \vee \neg \rightarrow migrate(X)) \wedge \neg \rightarrow \neg compliant(X)$ $\mathcal{M}, finish(X) \models completed(X)$
$\mathbf{M_6}$	**optional initialization of v2 and optional transfer to v2 (one-time only)**	 $\mathcal{M}, start(X) \models process(X, v1) \rightarrow (migrate(X) \wedge \neg \rightarrow migrate(X)) \vee \neg \rightarrow migrate(X)$ $\mathcal{M}, finish(X) \models completed(X)$

By controlling initializing of processes and migration by excluding it completely, allowing it or forcing it, already more fine-grained and advanced scenarios are possible than described in the basic migration strategies, even without considering aborting instances.

6.2.2 Temporary process incompliancy

In the meta models discussed thus far model execution is only constrained to reach a state where it completes the model, as forced by the statement $\mathcal{M}, finish(X) \models completed(X)$ found in all meta models The meta models do not force that the execution follows the model at all times, as we can see in the following example.

Suppose we take meta model $\mathbf{M_2}$ and have $v1 = \mathbf{P_1}$ and $v2 = \mathbf{P_1}$. A surprising allowed execution of this meta model is where an execution starts $\mathbf{P_1}$ and executes the trace `<a a> <c` which is not compliant to $\mathbf{P_1}$. We know `<a a> <c` is not compliant because it is not found as a prefix of any of the completion traces of $\mathbf{P_1}$, so there is no way the trace can be supplemented to a completion trace. This trace is at the meta process level still allowed because the instance can still migrate to $\mathbf{P_2}$ and satisfy that model by completing the trace to `<a a> <c c> <b b>`.

Whether this temporary violation of the model is allowed can be controlled by adding the TPL phrase $\neg \rightarrow \neg compliant(X)$ to the meta model meaning that there may not follow a state in which the instance X is not compliant to its model.

6.2.3 Meta model level metrics

At the meta process level we have already seen that we can specify again models and traces. We say a meta process level trace is meta-compliant if it satisfies the compliancy definition for its meta model Table 6.5 defines metrics that can be applied to meta models and relate to characteristics of the meta model $\mathbf{MC_{v1}}$ gives the acceptance of executions following the old process version, $\mathbf{MC_{v2}}$ that of executions starting with the new model. $\mathbf{MC_{full}}$ gives an indication of how accepting or complementary how strict a meta model is. **VAR** gives the variability at the meta process level, typically high when migrations are allowed at various positions. **PROP** gives the percentage of traces that is allowed to migrate. **PC** gives the percentage of traces that is forced to follow its process definition or complementary deviate temporarily. **L** gives an indication of how many steps an evenly divided population of executions would need to finish, taking best case routes, possibly by exploiting migrations.

Table 6.5: Metrics at the meta process level

Abbreviation	Definition	Calculation
$\mathbf{MC_{v1}}$	meta-compliant % of process executions of $v1$	Create a spanning set of all $v1$ compliant executions, check if the meta model is also compliant. Take the percentage of meta process level compliant traces on the spanning set.
$\mathbf{MC_{v2}}$	meta-compliant % of process executions of $v2$	Create a spanning set of all $v2$ compliant executions, check if the meta model is also compliant. Take the percentage of meta process level compliant traces on the spanning set.
$\mathbf{MC_{full}}$	meta-compliant % of process executions of all executions	Create a complete spanning set of executions, check if the meta model is compliant. Take the percentage of meta process level compliant traces on the spanning set.
VAR	The average allowed variability of meta-compliant traces in the future	Take the set of of all meta process level compliant traces. Count the number of completions (that validate the meta model). Calculate the percentage over the size of the original set.
PROP	% of meta-compliant traces allowed to migrate in the future	Take the set of of all meta process level compliant traces. If there is a completion that includes a migrate event, count it in. Calculate the percentage over the size of the original set.
PC	% of executions that is also compliant to the underlying processes at all times during execution.	Take the set of of all meta process level compliant traces. Verify if the execution is always compliant at the process level, then count it in. Calculate the percentage over the size of the original set.
L	The average critical path encountered in meta-compliant traces.	Take the set of of all meta process level compliant traces. For each instance trace, complete it and determine the critical path. Take the average for the entire set.

It should be noted that the metrics defined here are also Dependant of the processes. This is not an entirely bad property, as in practice meta models will have to be chosen with specific processes in mind anyway. But we will show characteristics of the meta models we defined to show general properties.

6.2.4 Meta model test data collection

To measure the effects of applying a meta model we apply the meta model to possible executions and calculate the metrics described in the previous section. A choice that has to be made is to choose test traces, we divide test traces into two groups, the well behaved meta process traces that correspond to an instance following the model. The second group is the complete set of possible meta process traces by taking the meta process trace for every trace in the full span of the activities. We take a meta process trace from the test-set as input and see how the meta model handles the meta process trace. The result can be one of two options. The first is the meta process trace is rejected by the meta model, resulting in a zero for compliancy and no resulting other metrics. The latter is that the meta process trace yields a non-empty set of completion traces. Of this set the minimum path length on process-level is determined, the process compliancy coverage is determined and the size of the completion set provides the variability.

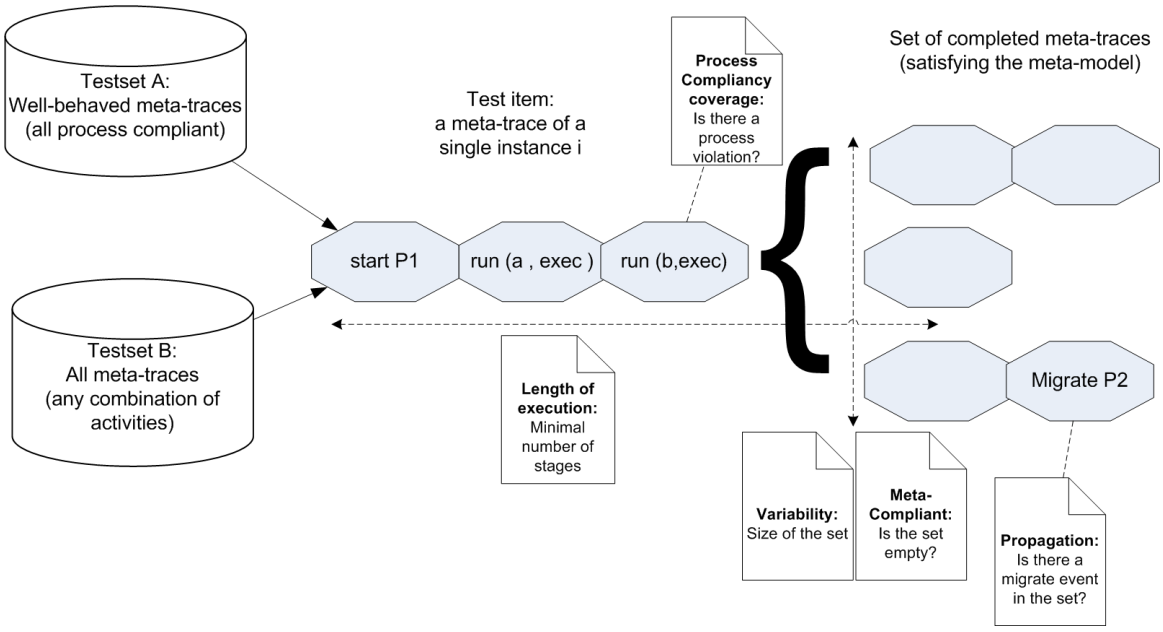


Figure 6.1: The figure shows how metrics are measured. Sets of traces are used to check if the system can finish the trace and fulfill the meta models On the set of these outcomes metrics are measured as shown in the figure.

6.2.5 Meta model test results

Table 6.6 shows the acquired measurements for the meta models we have defined in table 6.4. Table 6.6 actually provides empirical evidence to support properties of the defined meta models For meta models $\mathbf{M_1}$ and $\mathbf{M_2}$ we can see that indeed these do not allow instantiation of $v2$, as metric $\mathbf{MC_{v2}}$ always remains at 0%. Likewise we can verify that models $\mathbf{M_1}$ and $\mathbf{M_4}$ indeed do not allow migration, as metric **PROP** is always at 0%. Meta model $\mathbf{M_3}$ shows anomalies in $\mathbf{MC_{v1}}$, the only meta model that shows values below 100%. This actually correlates with the fact that $\mathbf{M_3}$ as only meta model forces migrations to $v2$. So should a trace be following $v1$ and have progressed beyond a point where it can migrate, it is rejected by $\mathbf{M_3}$. For scenarios 1 and 2 this can happen, but not for 3, since $\mathbf{P_4}$ completely contains $\mathbf{P_1}$ and migration is then always possible. In general we can see an increasing tendency of $\mathbf{MC_{full}}$ in order of indices of the meta models, indicating that the meta models are indeed in order of strictness. Whenever the metric **PC** is less than 100%, we allow temporary process incompliancies. Interestingly enough this can be prevented either by forcing compliancy in the meta model, or simply by forcing process completion and disabling migration.

Table 6.6: Measurements of the meta models applied to various model changing scenarios. The metrics are shown in fractions and percentages, relative to the variability of models and activity spanning sets.

Scenario 1: $\mathbf{P_1}$ changes to $\mathbf{P_2}$

	$\mathbf{MC_{v1}}$	$\mathbf{MC_{v2}}$	$\mathbf{MC_{full}}$	**VAR**	**PROP**	**PC**	**L**
$\mathbf{M_1}$	100%	0%	1%	1	0%	100%	3
$\mathbf{M_2}$	100%	0%	2%	3.2	64%	73%	3
$\mathbf{M_3}$	43%	100%	3%	2.5	50%	79%	3
$\mathbf{M_4}$	100%	100%	3%	1	0%	100%	3
$\mathbf{M_5}$	100%	100%	3%	1.4	21%	100%	3
$\mathbf{M_6}$	100%	100%	3%	2.3	39%	83%	3

Scenario 2: $\mathbf{P_1}$ changes to $\mathbf{P_3}$

	$\mathbf{MC_{v1}}$	$\mathbf{MC_{v2}}$	$\mathbf{MC_{full}}$	**VAR**	**PROP**	**PC**	**L**
$\mathbf{M_1}$	100%	0%	1%	1	0%	100%	3
$\mathbf{M_2}$	100%	0%	3%	6.6	83%	50%	2.2
$\mathbf{M_3}$	57%	100%	6%	4.7	50%	70%	2
$\mathbf{M_4}$	100%	100%	4%	1.6	0%	100%	2.3
$\mathbf{M_5}$	100%	100%	4%	3	18%	100%	2.1
$\mathbf{M_6}$	100%	100%	6%	4.5	45%	73%	2.1

Scenario 3: $\mathbf{P_1}$ changes to $\mathbf{P_4}$

	$\mathbf{MC_{v1}}$	$\mathbf{MC_{v2}}$	$\mathbf{MC_{full}}$	**VAR**	**PROP**	**PC**	**L**
$\mathbf{M_1}$	100%	0%	1%	1	0%	100%	3
$\mathbf{M_2}$	100%	0%	4%	8.3	100%	43%	2.3
$\mathbf{M_3}$	100%	100%	8%	5	50%	71%	2.3
$\mathbf{M_4}$	100%	100%	5%	1.8	0%	100%	2.5
$\mathbf{M_5}$	100%	100%	5%	4.1	25%	100%	2.3
$\mathbf{M_6}$	100%	100%	8%	5.2	50%	73%	2.3

Scenario 4: $\mathbf{P_4}$ changes to $\mathbf{P_1}$

	$\mathbf{MC_{v1}}$	$\mathbf{MC_{v2}}$	$\mathbf{MC_{full}}$	**VAR**	**PROP**	**PC**	**L**
$\mathbf{M_1}$	100%	0%	4%	2	0%	100%	2.3
$\mathbf{M_2}$	100%	0%	4%	3.3	33%	100%	2.3
$\mathbf{M_3}$	33%	100%	3%	2.5	50%	100%	3
$\mathbf{M_4}$	100%	100%	5%	1.8	0%	100%	2.5
$\mathbf{M_5}$	100%	100%	5%	2.8	25%	100%	2.5
$\mathbf{M_6}$	100%	100%	5%	2.8	50%	100%	2.5

Chapter 7

Discussion and future work

In our approach we have implemented process validation for declarative process models in TPL. Although similar solutions have been shown for other languages, such as LTL and CTL, this is the first time this has been conducted for a run-time execution model using TPL expressions. We have proposed meta process level modeling, again using TPL to describe policies with which to handle process migration. Thus we provide a solution for the migration configuration scenario presented in chapter 4.

7.1 Discussion of TPL's expressiveness for meta models

The main topic of the discussion will be to assess the application of TPL meta models In which respects are the meta models an improvement over existing models from the related work, and which areas still have to be examined. We show this by comparison to previously discussed related work.

7.1.1 TPL meta models in perspective of migration strategies

To place the results in perspective we give a comparison of the basic migration strategies mentioned in the related work (table 3.4) with the meta models introduced in table 6.4

Table 7.1: Comparison of meta models table 6.4 to basic migration strategies as described in [6, 28]

Strategy	Matching meta model(s)	Side notes
Forward recovery	None	Abortion was not studied in particular. $abort(X)$ phrases can replace or supplement $finish(X)$ phrases, leading to an implementation analogue to the meta models examined.
Backward recovery	None	Implementation work did not focus on compensation and roll-back, following the recent trend in the field.

Strategy	Matching meta model(s)	Side notes
Proceed	$\mathbf{M_1}$ and $\mathbf{M_4}$	Proceeding means following the process model that was assigned at the start, in TPL this translates to $\neg \rightarrow migrate(X)$. Still instantiation can be configured along-side this.
Transfer	$\mathbf{M_2}$,$\mathbf{M_3}$, $\mathbf{M_5}$ and $\mathbf{M_6}$	We have shown various ways to transfer instances, either mandatory or optionally. Also we can be strict or loose on process level compliance of the instances.
Detour	$\mathbf{M_2}$,$\mathbf{M_3}$ and $\mathbf{M_6}$	These meta models temporarily allow incompliant states of execution.

7.1.2 Comparison with existing frameworks on configurability

In table 7.2 we show how the proposed solution with the implementation of a **TPL reasoning system** compares to existing frameworks. The **eFLOW** does provide a mechanism to define rules on migrations, but it is very limited in expressive power and lacks a solid formalism. The TPL language defines migrations in a declarative way, allowing more flexibility and specification, not just of transfers, but behavior of executions in general.

Table 7.2: Comparison in configurability with existing frameworks

Framework	Change policy configuration	Rating
DECLARE	static migration procedure	very poor
ADEPT2	choice of predefined migration procedures, or manual transferring at instance level.	poor
eFLOW	Event-Condition-Action driven migration scripts	average
TPL reasoning system	Declarative meta models to configure change procedures	good

7.1.3 Separation of concerns between models and meta models

What the proposed solution does well is allow models to specify what needs to happen and meta models how executions should realize the models. Interestingly the same language operators are applied at both levels of abstraction.

7.1.4 Suggestions for alternative notations in TPL

Having experienced design of processes in TPL, a number of additions are suggested in this

section to simplify the notation of processes.

Temporal operators on history

TPL now has temporal operator that can be applied as binary operator, explicitly stating a condition for a current state and providing the consequence in the future. In the expression of models and meta models this can be a limitation, leading to complicated statements on the first occurring state $start(X)$. It might be more convenient to express that the state $finish(X)$ has had some conditions qualified in the past, without explicitly referring to whatever state that was. For this the reverse temporal operator $\leftarrow$ could be a handy addition to the language. Interpretation should be:

$$\mathcal{M}, x \models \leftarrow a \Leftrightarrow \forall \sigma \in \Omega_P : x \in \sigma \Rightarrow \exists y \in \sigma : y <_\sigma x \wedge \mathcal{M}, y \models a$$

Associative negative temporal operators

In meta models we often see the pattern $expr_1 \rightarrow (expr_2 \wedge \neg \rightarrow expr_3)$. A more intuitive notation would be simply to write $expr_1 \rightarrow (expr_2 \not\rightarrow expr_3)$, where the operator $\not\rightarrow$ is interpreted as $\neg \rightarrow$.

Associative persistent temporal operators

We have already seen the construction $\neg \rightarrow \neg expr$ used in meta models to demand that an expression $expr$ must follow at all times. This we may want to rewrite to $\overset{p}{\rightarrow}$.

7.1.5 Using TPL to express more constraint types

In the problem statement complicated constraints have been used to show examples of business demands involved in process migration scenarios. The solution presented in this thesis is still not sufficient to model all constraints from the exemplary stories. We will describe how support for these constraint types can be built into the proposed solution by adding language constructs.

Real-time constraints

To support real-time constraints we could introduce predicates of the form $before($`<timestamp>`$)$ and $before(X,$ `<timestamp>`$)$ which are evaluated to true if the time constraint is satisfied at that state in the validation process.

Data constraints

Supporting data constraints has to be implemented in the execution model, providing data usage relations to activities.

Resource constraints

Supporting resource constraints has to be implemented in the execution model, providing resource usage relations to activities.

Selection constraints

We might encounter constraints that apply to a selection of instances of the population. Selection criteria can be qualitative in nature in which case they can be implemented in TPL by introducing a variable that is true when the criterion is met. More problematic are constraints that govern a fixed number of instances, or have some other selection criterion that depends not on the instance itself but on conditions of the population. To model such criteria new functors need to be introduced that do not depend on dynamic instance predicates. An example can be a meta process level predicate $number_of_instances_bound(P, 1, 6)$ which is true when there are between 1 and 6 instances running process P at that state in the validation process.

7.2 Outlook and unresolved issues

For future work going towards an integrated system to support continuous process redesign and analysis would be a good direction. Although this research does provide a framework architecture, many of the subsystems remain unexplored. The actual implementation of dynamic service composition on this scale and in this setup is not investigated yet. Also the tooling support for TPL process design and analysis is lacking. Although we have provided some useful metrics, run-time measuring and application to an actual population of executions is not performed. Related issues and tasks for further research are explored below.

7.2.1 Ad hoc TPL process model creation

Although the solution describes ways in which a process can migrate between predefined models, a powerful addition would be to use reasoning on TPL models to create uniquely tailored process models as well during the execution. This would enable migration to intermediate solutions, further controlling the deviating flows. We can attempt for instance to construct process models that resemble a process as closely as possible but match the execution of a deviating instance. Perhaps process merging techniques as described in [8] can be applied for this purpose.

7.2.2 Integrated process development environment

The metrics and met-models we have shown in our results can be applied in a Business Process Development environment. Since the metrics reflect properties of the behavior of models it makes sense to apply these in the practices of redesign and analysis of Business Processes. A framework that provides a run-time view on the system and allows comprehensive analysis prognosis and suggestions for improvement would provide more flexible support of processes than now exists. To provide this more work on visualization and analysis of process models and on the population of executions needs to be done. Especially with the introduction of declarative models it is important to have good representations of what effects models and model changes will have.

7.2.3 Including data and resource perspectives

The data and resource perspectives have not been implemented in this research while it has been shown that these perspectives do impose constraints on dynamic changes. In fact in most of the related work these perspectives are also neglected. In [32] mention *Another intriguing direction is data integration for required services, since a replacement service may require data not needed by the original service.* This case is stated as possible future work and not answered in any way in the article. In [6] it is also mention that *" other perspectives addressing data, resources and applications used in a process are also subject to change."*

7.2.4 Hybrid approach: combining the declarative and imperative

An interesting approach is to combine both declarative and imperative approaches, as is done in the SeaFlows framework [37], suggested as well in [8]. Where declarative models are good at specifying specifying a large number of alternatives in a single statement, imperative models are easier to understand, visualize, and simpler models for highly restrictive control flows.

7.2.5 TPL model satisfaction and planning

The solution we have described models hard-constraints and leaves allowed executions open to choice by the user. Soft-constraints can be applied to plan a way in the open solution space. Planning is another but related research topic worth mentioning here.

Chapter 8

Conclusion

The aim of this research has been to investigate how a Process Aware Information System can be built such that it supports dynamic changes to processes and allows configuration of the migration policy. The migration policy defines how, if at all, the change is to be adopted. We have shown the use case for migration strategy and proposed a framework architecture that provides the required capabilities. We have further examined the crucial subsystem that implements the migration policy logic. For this reasoning system we have shown that TPL can be used to describe processes and the provide can provide an accurate model of the meta process level. This meta process level of the process we regard as the level at which we explicitly model execution of instances. At the meta process level it is defined which executions are present, how these executions behave and which processes are instantiated and so on. An implementation of the TPL reasoning system is provided, which implements TPL model validation. The algorithm for validation is derived and we show correctness of the algorithm by pointing out the criteria which must be met for all language constructs and how the implementation only validates the traces that meet the criteria. Next we have shown examples of TPL-models and meta process levels and shown how these can be applied in practice to get various results on controlling process migration. Regarding the research questions given in chapter 4 we now discuss the extent to which the research questions have been answered.

Are process traces a sufficient description of run-time execution to evaluate TPL expressions?

We have answered this question by showing that TPL model validation is possible on the level of traces and that traces correspond to an execution of a process. Depending on which amount of information is modeled by the trace and how traces are precisely generated, we can model any property of run-time execution using traces. We have shown that modeling of iterative execution, can-cancellation and possibly data and resource constraints is possible and TPL can be expressed over these traces. To limit the set of initial problems we have limited the implementation to a simple execution scheme. Investigation of more powerful execution models is not implemented, but traces can be used to describe these modifications.

Given a meta model formulated in TPL, how does the control over migrations compare to existing migration strategies and languages?

We have shown that just looking at proceed and transfer strategies, the provided TPL meta models provide a multitude of options to describe these strategies, we have explicitly shown 6 models corresponding to just the proceed and transfer strategies and this is not an exhaustive list. Although the other three strategies are not examined, it is expected that a similar situation can be derived for these strategies. It has been pointed out that TPL meta models are not limited to a description just of two versions of the process, but can be

used to describe any number of processes in various ways. Also when looking at the extent of control we have shown that with the TPL-meta models, we can constrain initialization, transfer and compliance to being strictly enforced, open or completely restricted using TPL constructs. Also we can build propositional combinations of the conditions in any way we want.

Do *propagation*, *execution length*, *compliancy* and *variability* notions provide discerning metrics for TPL (meta-)models?

We have shown metrics that can be obtained both at the process level and at the meta process level, based on *propagation*, *execution length*, *compliancy* and *variability* notions. The metrics do indeed reflect properties of the models used as we have pointed out for the models in our test set.

Having provided answers for each of the research questions and summarized the realization of our research aim we can say in conclusion that the proposed solution in this thesis provides a new approach to modeling dynamic change, allowing change policies to be defined from formal language constructs with fine-grained control over properties of migrations. This is a significant improvement over the current proposed solutions which although providing reasonable support to describe what has to change, only provide a small general list of strategies defined as adhoc-solutions. The solution we propose serves as another small step towards more flexible information systems to support Business Processes.

Bibliography

[1] Marco Sinnema Pavel Bulanov Chang-ai Sun, Rowan Rossing and Marco Aiello. Modeling and managing the variability of web service-based systems. *The Journal of Systems and Software*, (83):502–516, October 2009.

[2] Chang ai Sun and Marco Aiello. Towards variable service compositions using vxbpel. *Proceedings of the International Conference on Software Reuse (ICSR), Lecture Notes in Computer Science*, 5030:257–261, 2008.

[3] Chris Peltz. Web services orchestration, a review of emerging technologies, tools, and standards. *Tech. Rep., Hewlett-Packard Company.*, January 2003.

[4] S. Sadiq W. Bandara, M. Indulska and S. Chong. Major issues in business process management: an expert perspective. *In Proceedings of the Fifteenth European Conference on Information Systems*, pages 1240–1251, 2007.

[5] Wil M. P. Van Der Aalst, Arthur H. M. Ter Hofstede, and Mathias Weske. Business process management: a survey. In *Proceedings of the 2003 international conference on Business process management*, BPM'03, pages 1–12, Berlin, Heidelberg, 2003. Springer-Verlag.

[6] Nick Russell Nataliya Mulyar Helen Schonenberg, Ronny Mans and Wil van der Aalst. Process flexibility: A survey of contemporary approaches. *Lecture Notes in Business Information Processing*, 8:16–30, June 2008.

[7] M. Aiello, P. Bulanov, and H. Groefsema. Requirements and tools for variability management. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*, pages 245 –250, July 2010.

[8] Pavel Bulanov, Alexander Lazovik, and Marco Aiello. Business Process Customization using Process Merging Techniques. *to appear*, 2011.

[9] Maja Pesic. *Constraint-based Workflow Management Systems: Shifting Control to Users.* PhD thesis, Eindhoven, Technische Universiteit Eindhoven, 2008.

[10] Paulo Barthelmess and Jacques Wainer. Workflow systems: a few definitions and a few suggestions. In *Proceedings of conference on Organizational computing systems*, COCS '95, pages 138–147, New York, NY, USA, 1995. ACM.

[11] W.M.P. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, February 2003.

[12] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. Covamof: A framework for modeling variability in software product families. In Robert L. Nord, editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 25–27. Springer Berlin / Heidelberg, 2004.

[13] Bassam Atieh Rajabi and Sai Peck Lee. Change management in business process modeling survey. *Information Management and Engineering, International Conference on*, 0:37–41, 2009.

[14] Jeannette M. Wing. Faq on π-calculus, 2002.

[15] Howard Smith and Peter Fingar. Workflow is just a Pi process, November 2003.

[16] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, January 2007.

[17] Michiel Koning, Chang-ai Sun, Marco Sinnema, and Paris Avgeriou. Vxbpel: Supporting variability for web services in bpel. *Inf. Softw. Technol.*, 51:258–269, February 2009.

[18] W. M. P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[19] W. M. P. van der Aalst. Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype, 2004.

[20] Nabil R. Adam, Vijayalakshmi Atluri, and Wei-Kuang Huang. Modeling and analysis of workflows using petri nets. *J. Intell. Inf. Syst.*, 10:131–158, March 1998.

[21] W. M. P. van der Aalst and T. Basten. Inheritance of workflows: an approach to tackling problems related to change. *Theor. Comput. Sci.*, 270:125–203, January 2002.

[22] Li Xingyu, Hu Hao, and Lu Jian. Variability model and management for web service: A petri net based approach. In *Computer and Information Technology, 2008. ICCIT 2008. 11th International Conference on*, pages 7 –12, 2008.

[23] M. Pesic and W. van der Aalst. A declarative approach for flexible business processes management. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 169–180. Springer Berlin / Heidelberg, 2006.

[24] M. Pesic, M. Schonenberg, N. Sidorova, and W. van der Aalst. Constraint-based workflow models: Change made easy. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *Lecture Notes in Computer Science*, pages 77–94. Springer Berlin / Heidelberg, 2007.

[25] Andreas Speck, Sven Feja, Witt Sören, Elke Pulvermüller, and Marcel Schulz. Formalizing business process specifications. *ComSIS*, 8(2), May 2011.

[26] Stefanie Rinderle-Ma, Manfred Reichert, and Barbara Weber. Relaxed compliance notions in adaptive process management systems. In Qing Li, Stefano Spaccapietra, Eric Yu, and Antoni Oliv, editors, *Conceptual Modeling - ER 2008*, volume 5231 of *Lecture Notes in Computer Science*, pages 232–247. Springer Berlin / Heidelberg, 2008.

[27] Clarence Ellis, Karim Keddara, and Grzegorz Rozenberg. Dynamic change within workflow systems. In *Proceedings of conference on Organizational computing systems*, COCS '95, pages 10–21, New York, NY, USA, 1995. ACM.

[28] W. M. P. van der Aalst and S. Jablonski. Dealing with workflow change: identification of issues and solutions. *International Journal of Computer Systems Science and Engineering*, 15(5):267–276, September 2000.

[29] Sukho Kang Moonsoo Cho Dongsoo Kim, Namhun Lee and Minsoo Kim. A version management method for managing business process changes based on version-stamped business process change patterns. *International Journal of Innovative Computing, Information and Control*, 6(2):567–575, February 2010.

[30] Dongsoo Kim, Minsoo Kim, and Hoontae Kim. Dynamic business process management based on process change patterns. *Convergence Information Technology, International Conference on*, 0:1154–1161, 2007.

[31] S. Shazia, S. Olivera, M. Maria, and E. Orlowska. Managing change and time in dynamic workflow processes, March 1999.

[32] Mati Golani and Avigdor Gal. Flexible business process management using forward stepping and alternative paths. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 48–63. Springer Berlin / Heidelberg, 2005.

[33] Stefanie Rinderle-Ma and Reichert Manfred. Adjustment strategies for non-compliant process instances. Technical report, March 2009.

[34] Stefanie Rinderle-Ma and Manfred Reichert. Advanced migration strategies for adaptive process management systems. *E-Commerce Technology, IEEE International Conference on*, 0:56–63, 2010.

[35] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. Workflow evolution. In *Proceedings of the 15th International Conference on Conceptual Modeling*, ER '96, pages 438–455, London, UK, 1996. Springer-Verlag.

[36] Stefanie Rinderle-Ma, Peter Dadam, and Linh Thao Ly. Design and verication of instantiable compliance rule graphs in process-aware information systems. In *The 22nd International Conference on Advanced Information Systems Engineering (CAiSE'10)*, February 2010.

[37] Ly Linh Thao, Göser Kevin, Stefanie Rinderle-Ma, and Dadam Peter. Compliance of semantic constraints - a requirements analysis for process management systems. In *Proc. 1st Int'l Workshop on Governance, Risk and Compliance - Applications in Information Systems (GRCIS'08)*, 2008.

[38] Fabio Casati, Ski Ilnicki, Li-jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, CAiSE '00, pages 13–31, London, UK, 2000. Springer-Verlag.

[39] Clarence Ellis and Karim Keddara. Ml-dews: Modeling language to support dynamic evolution within workflow systems. *Comput. Supported Coop. Work*, 9:293–333, August 2000.

[40] Linh Ly, Stefanie Rinderle, and Peter Dadam. Semantic Correctness in Adaptive Process Management Systems. pages 193–208. 2006.

[41] W.M.P. van der Aalst and M. Pesic. Specifying and monitoring service flows: Making web services process-aware. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 11–55. Springer Berlin Heidelberg, 2007.

[42] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 193–202, New York, NY, USA, 2004. ACM.

[43] Manfred Reichert, Stefanie Rinderle, Ulrich Kreher, and Peter Dadam. Adaptive process management with adept2. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 1113–1114, Washington, DC, USA, 2005. IEEE Computer Society.

[44] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Correctness criteria for dynamic changes in workflow systems: a survey. *Data Knowl. Eng.*, 50:9–34, July 2004.

[45] Robert D. Klassen and Larry J. Menor. The process management triangle: An empirical investigation of process trade-offs. *Journal of Operations Management*, 25(5):1015 – 1034, 2007. Special Issue on Innovative Data Sources for emperically building and validating theories in Operations Management.

Appendix A

Prolog implementation

```
%=====================================
%   TPL language syntactical constructs
%=====================================

:- op(950,yfx,>).     % implication
:- op(900,yfx,v).      % disjunction
:- op(850,yfx,&).      % conjunction
:- op(750, fy,~).       % negation
:- op(900,yfx,'<>').  % binary implication
:- op(800,xfy, '-->'). % tpl: leads to
:- op(800,xfy, '=>'). % tpl: leads
        % always to
:- op(800,xfy, '~>'). % tpl: may lead to

%unary tpl ops
:- op(800, fy, '-->').
:- op(800, fy, '=>').
:- op(800, fy, '~>').

:- dynamic testset/2.

%=====================================
%   Some example TPL models
%=====================================

%tpl_model(m1,a,a).
%tpl_model(m1,b,b).

%tpl_model(m2,a,-->b).
%tpl_model(m2,b,b).

%tpl_model(m3,a,a & ~(-->b)).
%tpl_model(m3,b,b).

%tpl_model(m4,a,a-->b).
%tpl_model(m4,b,b-->c).
%tpl_model(m4,c,c).

activity(a).
activity(b).
activity(c).


%=====================================
%   Activity state machine
%=====================================
act_state(init).
act_state(exec).
act_state(success).
act_state(fail).

act_state_transition(init,exec).
act_state_transition(exec,success).
%act_state_transition(exec,fail).
%allow multiple executions.
%act_state_transition(success,exec).
%act_state_transition(fail,exec).

event(A,S) :- activity(A), act_state(S).

%=====================================
%   Process execution
%=====================================
get_model_set(M,MS) :- setof(
  sat_state(A,Expr),
  tpl_model(M,A,Expr),MS).

follow_process(M,Tinit,Tlater) :-
  replay_trace(M,Tinit,States,MD),
  follow_process_h(Tinit,Tlater,States,MD).

follow_process_h(T,T,_,_).
follow_process_h(T1,T3,S1,MD1) :-
  process_step(T1,T2,S1,S2,MD1,MD2),
  follow_process_h(T2,T3,S2,MD2).

follow_process_step(M,Tinit,Tnext) :-
  replay_trace(M,Tinit,States,MD),
  process_step(Tinit,Tnext,States,
    S2,MD,MDNew),
  condition(underway,Tnext,S2,MDNew).
follow_process_cond(M,C,Tinit,Tfinal) :-
  replay_trace(M,Tinit,States,MD),
  process_trace_cond(C,States,MD,
    Tinit,Tfinal).

replay_trace(M,Tinit,States,MD) :-
  get_model_set(M,MSFull),
  initial_states(I),
  process_trace([],Tinit,I,States,
  model_data(MSFull,[],[]),MD).

process_trace(T,T,S,S,MD,MD).
process_trace(T1,Tf,S1,Sf,MD1,MDf) :-
  process_step(T1,T2,S1,S2,MD1,MD2),
  append(T1,X,Tf),length(X,N),N>0,
  process_trace(T2,Tf,S2,Sf,MD2,MDf).

process_trace_cond(C,S,MD,T,T) :-
  condition(C,T,S,MD).
process_trace_cond(C,S1,MD1,T1,Tf) :-
  process_step(T1,T2,S1,S2,MD1,MD2),
  process_trace_cond(C,S2,MD2,T2,Tf).


process_step(T1,T2,States1,States2,
                    ModelData1,ModelData2) :-
```

```
  event_step(T1,T2,States1,States2),
  check_model_data(T1,T2,
    ModelData1,ModelData2).

event_step(T1,T2,S1,S2) :-
  replace_state(A,X,S1,S2),
  append(T1,[event(A,X)],T2).

check_model_data(_,T2,M,M) :-
  \+new_stage(T2).
check_model_data(T1,T2,M1,M2) :-
  new_stage(T2),
  update_model_data(T1,M1,M2).

update_model_data(T,
    model_data(MExprs1,Fut1,PFut1),
    model_data(MExprs2,Fut2,PFut2)) :-
  update_model_exprs(T,MExprs1,MExprs2,
  AFut1,APFut1),
  pre_process_exprs(Fut1,PFut1,FFut1,FPFut1),
  update_futgoals(executed,T,FFut1,AFut2,
  APFut2),
  update_prohibgoals(executed,T,FPFut1,APFut3),
  append(AFut1, AFut2, Fut2),
  flatten([APFut1,APFut2,APFut3],PFut2).


%====================================
%   Process execution helper functions
%====================================
pre_process_exprs(Fut,FutP,FFut,FFutP) :-
  process_goal_list(Fut, NewFut,AddedFutP),
  process_goal_list(FutP, NewFutP,AddedFut),
  append(NewFut,AddedFut,FFut),
  append(NewFutP,AddedFutP,FFutP).

update_model_exprs(T,MS1,MS2,AddedFut,
         AddedFutP) :-
  update_sat_states_h(T,MS1,MS2,[],
     AddedFut,[],AddedFutP).

update_sat_states_h(_,[],[],Fut,Fut,FutP,FutP).
update_sat_states_h(T,[sat_state(A,Expr)|MSR],
      MSF,Fut1,Fut2,FutP1,FutP2) :-
    (eval_once(executed,T,A,AddedFutS,_,true),
    eval_once(executed,T,Expr,AddedFutX,
        AddedFutP,true))->
    ((AddedFutS=[]->AddedFut=AddedFutX;
    apply_to_goal(follow1_op,AddedFutS,
          AddedFutX,AddedFut)),
    update_sat_states_h(T,MSR,MSF,Fut1,FutM,
        FutP1,FutPM),
    append(AddedFut,FutM,Fut2),
    append(AddedFutP,FutPM,FutP2));
    (update_sat_states_h(T,MSR,MS2,
      Fut1,Fut2,FutP1,FutP2),
    MSF=[sat_state(A,Expr)|MS2]).

update_futgoals(_,_,[],[],[]).
update_futgoals(Mode,T,[G1|Fut1],Fut2,FutP2) :-
eval_once(Mode,T,G1,AddedFuts,AddedFutP,R),
    R==true,
update_futgoals(Mode,T,Fut1,FutR,FutP1),
append(AddedFuts,FutR,Fut2),
   append(AddedFutP,FutP1,FutP2).
update_futgoals(Mode,T,[G1|Fut1],
  [G1|Fut2],FutP) :-
eval_once(Mode,T,G1,_,_,R),R==false,
update_futgoals(Mode,T,Fut1,Fut2,FutP).

update_prohibgoals(_,_,[],[]).

%okay as long as the prohibited
% goals remain false.
update_prohibgoals(Mode,T,[G|PFut1],[G|PFut2]) :-
  eval_once(Mode,T,G,_,_,false),
  update_prohibgoals(Mode,T,PFut1,PFut2).

% okay as long as the prohibited goals have some
% future goals and their future
% and not their prohibited future are checked.
update_prohibgoals(Mode,T,[G|PFut1],PFut2) :-
  eval_once(Mode,T,G,AddedFut,AddedFutP,R),
  R==true,AddedFut\=[],
  update_prohibgoals(Mode,T,PFut1,PFutR),
  apply_to_goal(not_op,AddedFutP,AddedFutP_not),
  apply_to_goal(and_op,AddedFut, AddedFutP_not,
  NewAddedFutP),append(PFutR,NewAddedFutP,PFut2).


%====================================
%   Stage related helper functions
%====================================

% does last event of T2
% cause a stage transition?
new_stage(T2) :-
  append(T1,[_],T2),
  get_stage(T1,S1),
  get_stage(T2,S2),
  subtract(S1,S2,Diff),
  Diff\=[],!.

ended_during_latest_stage(T,A) :-
  get_latest_stage_events(T,Tlast),
  member(event(A,success),Tlast).
began_during_latest_stage(T,A) :-
  get_latest_stage_events(T,Tlast),
  member(event(A,exec),Tlast).

% returns partial trace T2 of T1
% which contains the sublist
% with the last stage events
get_latest_stage_events(T1,T2) :-
  get_latest_stage_events_h([],T1,[],T2).

get_latest_stage_events_h(_,[],T,T).
get_latest_stage_events_h
    (Tprev,[E|Trem],_,Curr_out) :-
  append(Tprev,[E],Tnext),
  new_stage(Tnext),
  get_latest_stage_events_h(Tnext,Trem,
    [E],Curr_out).
get_latest_stage_events_h(Tprev,[E|Trem],
    Curr_in,Curr_out) :-
  append(Tprev,[E],Tnext),
  \+new_stage(Tnext),
  append(Curr_in,[E],Curr_next),
  get_latest_stage_events_h(Tnext,Trem,
    Curr_next,Curr_out).

get_stage(Trace,Ssorted) :-
  get_stage_h(Trace,[],Pars),
  get_acts_from_trace(Pars,S),
  sort(S,Ssorted).
get_stage_exec(Trace,S) :-
  get_stage_h(Trace,[],Pars),
  remove_all_finished(Pars,Pars2),
  get_acts_from_trace(Pars2,S).

get_stage_h([],S,S).
get_stage_h([event(A,success)|R],
    Pars,S) :-
  select(event(A,_),Pars,ParsRem),
  get_stage_h(R,
```

```
    [event(A,success)|ParsRem],S).
get_stage_h([event(A,exec)|R],Pars,S) :-
  remove_all_finished(Pars,ParFilt),
  add_exec_event(A,ParFilt,ParFilt2),
  get_stage_h(R,ParFilt2,S).
get_stage_h([event(A,fail)|R],Pars,S) :-
  select(event(A,_),Pars,ParsRem),
  get_stage_h(R,ParsRem,S).

add_exec_event(A,L,[event(A,exec)|R]) :-
  select(event(A,exec),L,R).
add_exec_event(A,L,[event(A,exec)|L]) :-
  \+select(event(A,exec),L,_).

get_acts_from_trace([],[]).
get_acts_from_trace
      ([event(A,_)|Trem],[A|Arem]) :-
  get_acts_from_trace(Trem,Arem).

remove_all_finished([],[]).
remove_all_finished([event(A,exec)|Trem],
    [event(A,exec)|TfiltRem]) :-
  remove_all_finished(Trem,TfiltRem).
remove_all_finished(
    [event(_,State)|Trem],TfiltRem) :-
  State\=exec, remove_all_finished(Trem,TfiltRem).

last(L,E,R) :- last(L,E),select(E,L,R).

%====================================
%   Process conditions
%====================================

condition(completed,T,_,
    model_data(MExprs,Fut,PFut)) :-
  update_model_exprs(T,MExprs,[],[],_),
  pre_process_exprs(Fut,PFut,FFut,FPFut),
  update_futgoals(executed,T,FFut,[],_),
  update_prohibgoals(executed,T,FPFut,_).

condition(underway,T,S,MD) :-
  process_trace_cond(completed,S,MD,T,_),!.

%====================================
%   Administration of the state of the
%   set of activities
%====================================

initial_states(L) :-
  setof(state(A,init),activity(A),L).

replace_state(A,S2,[state(A,S1)|R],
     [state(A,S2)|R]) :-
  act_state_transition(S1,S2).
replace_state(A,S,
    [state(B,S2)|R],[state(B,S2)|R2]) :-
  replace_state(A,S,R,R2).

get_states(S,T) :-
  initial_states(I), reverse(T,Trev),
  get_states_h(Trev,I,S).

get_states_h(_,[],[]).
get_states_h(Trev,[state(A,_)|R],
    [state(A,S2)|R2]) :-
  first_occ_in_list(event(A,S2),Trev),
  !, get_states_h(Trev,R,R2).
get_states_h(Trev,[state(A,S)|R],
     [state(A,S)|R2]) :-
  get_states_h(Trev,R,R2).

first_occ_in_list(E,L) :- member(E,L),!.
```

```
%====================================
%   Semantic Evaluation
%====================================

eval_once(S,T,F,Fut,FutP,R) :-
  eval(S,T,F,Fut1,FutP1,true) *->
  (Fut=Fut1,FutP=FutP1,R=true);(Fut=[],FutP=[],R=false,!).

eval(executed,T,Formula,[],[],R) :-
   activity(Formula),
   (member(T,Formula)->R1=true;R1=false),
   (ended_during_latest_stage(T,Formula)
    ->R2=true;R2=false),
   ((R1=true;R2=true)->R=true;R=false).

eval(S,Trace,Formula1 & Formula2,
    FutGoal,ProhibGoal,R):-
   eval_once(S,Trace,Formula1,
    FutGoal1,ProhibGoal1,R1),
   eval_once(S,Trace,Formula2,
    FutGoal2,ProhibGoal2,R2),
   append(FutGoal1,FutGoal2,FutGoalNext),
   append(ProhibGoal1,
    ProhibGoal2,ProhibGoalNext),
   eval_and(R1,R2,R),
   switch(R,FutGoalNext,[],FutGoal),
   switch(R,ProhibGoalNext,[],ProhibGoal).

eval(S,Trace,Formula1 v Formula2,
    FutGoal,ProhibGoal,R):-
   eval_once(S,Trace,Formula1,
    FutGoal1,ProhibGoal1,R1),
   eval_once(S,Trace,Formula2,
    FutGoal2,ProhibGoal2,R2),
   eval_or(R1,R2,FutGoal1,ProhibGoal1,
    FutGoal2,ProhibGoal2,
    FutGoal,ProhibGoal,R).

eval(S,Trace,Formula1 > Formula2,FutGoal,
    ProhibGoal,R):-
   eval(S,(Formula1 v ~Formula2),Trace,
    FutGoal,ProhibGoal,R).

eval(S,Trace,~Formula,
    FutGoal,ProhibGoal,false):-
   eval(S,Trace,Formula,
    ProhibGoal,FutGoal,true).

eval(S,Trace,~Formula,
    FutGoal,ProhibGoal,true):-
  eval(S,Trace,Formula,
    ProhibGoal,FutGoal,false).

eval(_,_,--> Formula, [Formula],[],true).

eval(_,_,--> Formula, [Formula],[],false).

eval(S,Trace,Formula1 --> Formula2,
     [Formula2],ProhibGoal,R) :-
   eval(S,Trace,Formula1,[],ProhibGoal,R).

eval(S,Trace,Formula1 => Formula2,
    [=>Formula2],ProhibGoal,R) :-
   eval(S,Trace,Formula1,[],ProhibGoal,R).

eval(S,Trace,=> Formula, Fut,FutP,R) :-
   eval_merge(Trace,R1),
   eval(S,Trace,Formula,Fut,FutP,R2),
   eval_and(R1,R2,R),!.

eval(_,Trace,=> Formula, [Formula],[],R) :-
   eval_merge(Trace,R).
```

```
eval(S,Trace,Formula1 ~> Formula2,
     [~>Formula2],ProhibGoal,R) :-
   eval(S,Trace,Formula1,[],ProhibGoal,R).

eval(_,_,~> Formula, [~> Formula],[],_).

%=======================================
%   Semantic Evaluation (metamodel specific part)
%=======================================

%inst(ID,TRACE,PROCESS,STATUS)

eval(meta,step(_,I),completed(X),[],[],R) :-
   member(inst(X,ITrace,M,_),I),
   completed_model(M,ITrace)->R=true;R=false.

eval(meta,step(migrate(X),_),
   migrate(X),[],[],true).

eval(meta,step(E,_),migrate(X),[],[],false) :-
  E\=migrate(X).

eval(meta,step(_,I),process(X,V),[],[],R) :-
   process_mapping(V,P),member(inst(X,_,P,_),I)
    ->R=true;R=false.

eval(meta,step(_,I),satisfiable(X),
    [],[],R) :-
   member(inst(X,T,P,_),I),
  satisfiable_trace(P,T)->R=true;R=false.

%=====================================
%   Semantic Evaluation helper functions
%=====================================

%append actually used to
%remove the last stage elements here!
eval_merge(T,R) :-
  get_latest_stage_events(T,T2),
  append(Tprev,T2,T),
  get_stage_exec(Tprev,S),
  (S=[]->R=true;R=false).

completed_model(M,ITrace) :-
  replay_trace(M,ITrace,States,MD),
  condition(completed,ITrace,States,MD).

satisfiable_trace(M,ITrace) :-
  follow_process_cond(M,completed,ITrace,_),!.

%=====================================
%   Processing of optional futures
%=====================================

process_goal_list(Fut, NewFut,ChoicesPExpr) :-
  process_goal_list_h(Fut,Rem,
     Choices,ChoicesP),
  link_goals(or_op,Choices,ChoicesExpr),
  link_goals(or_op,ChoicesP,ChoicesPExpr),
  append(Rem,ChoicesExpr,NewFut).

process_goal_list_h([],[],[],[]).
process_goal_list_h([Expr|Fut], FutR,
     Choices, ChoicesP) :-
  process_goal_list_h(
    Fut,FutR1,Choices1,ChoicesP1),
  choice_list(Expr,L,LP,Rem),
  append(FutR1,Rem,FutR),
  append(Choices1,L,Choices),
  append(ChoicesP1,LP,ChoicesP).

choice_list(~> Formula, [Formula],[],[])
  :- !.

choice_list(~Formula, LP, L, FF) :-
  choice_list(Formula,L,LP,FF1),
  apply_to_goal(not_op,FF1,FF),!.
choice_list(F1 & F2, L, LP, FF) :-
  choice_list(F1,L1,LP1,FF1),
  choice_list(F2,L2,LP2,FF2),
  union(L1,L2,L), union(LP1,LP2,LP),
  apply_to_goal(and_op,FF1,FF2,FF),!.
choice_list(F1 v F2, L, LP, FF) :-
  choice_list(F1,L1,LP1,FF1),
  choice_list(F2,L2,LP2,FF2),
  union(L1,L2,L), union(LP1,LP2,LP),
  apply_to_goal(or_op,FF1,FF2,FF),!.
  choice_list(F,[],[],[F]).
%=====================================
%   Parsing helper functions (Omitted)
%=====================================

%=====================================
%   A stage division of a trace
%=====================================
get_path_length(T,N) :-
  get_stage_transitions(T,S),length(S,N).

get_stage_transitions(T,Sset) :-
  get_stage_transitions_h([],T,Sset).

get_stage_transitions_h(T,[],[S]) :-
  get_stage(T,S).
get_stage_transitions_h(T1,[E|R],[S|Sset]) :-
  append(T1,[E],T2),
  new_stage(T2),get_stage(T1,S),
  get_stage_transitions_h(T2,R,Sset).
get_stage_transitions_h(T1,[E|R],Sset) :-
  append(T1,[E],T2),\+new_stage(T2),
  get_stage_transitions_h(T2,R,Sset).

%=====================================
%=====================================
%
%   META MODELLING PART
%
%=====================================
%=====================================

instance(i).
%instance(j).

process_mapping(v1,p4).
process_mapping(v2,p1).

%=====================================
%   Some example TPL metamodels
%=====================================

meta_model(mm0,finish(X),completed(X)).

%no initialization of v2 and no transfer to v2
meta_model(m1, start(X), process(X,v1)
  & ~(-->migrate(X)) & ~(--> (~satisfiable(X))) ).
meta_model(m1, finish(X), completed(X) ).

%no initialization of v2 but optional transfer to v2
meta_model(m2, start(X), process(X,v1) &
  ( ~(-->migrate(X) ) v  ( -->( migrate(X)
  & ~ (--> migrate(X)) )) ) ).
meta_model(m2, finish(X), completed(X) ).

%optional initialization of v2
% and forced transfer to v2
meta_model(m3, start(X), process(X,v1)
  -->(migrate(X) &
  ~ (--> migrate(X))) v
```

```
  process(X,v2)
  & ~(--> migrate(X)) ).
meta_model(m3, finish(X), completed(X) ).

%optional initialization of v2
% but no transfer to v2
meta_model(m4, start(X), ~(-->migrate(X))).
meta_model(m4, finish(X), completed(X) ).

%optional initialization of v2 and
%optional transfer to v2
%with forced compliance
meta_model(m5, start(X),(process(X,v1)
  -->(migrate(X) & ~(-->migrate(X)) )
     v
   ~(--> migrate(X))) &
   ~(--> (~satisfiable(X))) ).
meta_model(m5, finish(X), completed(X) ).

%optional initialization of v2
% and optional transfer to v2
meta_model(m6, start(X),process(X,v1)
  -->(migrate(X) &
   ~(-->migrate(X)) )
  v ~(--> migrate(X)) ).
meta_model(m6, finish(X), completed(X) ).

%=====================================
%   metamodel execution
%=====================================
follow_metamodel(M,Tinit,Tlater) :-
  replay_mtrace(M,Tinit,States,MD),
  follow_metamodel_h(Tinit,Tlater,
    States,MD).

follow_metamodel_h(T,T,_,_).
follow_metamodel_h(T1,T3,S1,MD1) :-
  meta_step(T1,T2,S1,S2,MD1,MD2),
  follow_metamodel_h(T2,T3,S2,MD2).

follow_metamodel_step(M,Tinit,Tnext) :-
  replay_mtrace(M,Tinit,States,MD),
  meta_step(Tinit,Tnext,States,_,MD,_).

complete_metamodel(M,Tinit,Tfinal) :-
  replay_mtrace(M,Tinit,States,MD),
  complete_meta_run(States,MD,
    Tinit,Tfinal).

complete_meta_run(_,model_data([],[],_)
  ,T,T).
complete_meta_run(S1,MD1,T1,Tf) :-
  meta_step(T1,T2,S1,S2,MD1,MD2),
  complete_meta_run(S2,MD2,T2,Tf).

replay_mtrace(M,Tinit,States,MD) :-
  get_meta_model_set(M,MSFull),
  initial_i_states(I),
  meta_trace([],Tinit,I,States,
  model_data(MSFull,[],[]),MD).

meta_trace(T,T,S,S,MD,MD).
meta_trace(T1,Tf,S1,Sf,MD1,MDf) :-
  meta_step(T1,T2,S1,S2,MD1,MD2),
  append(T1,X,Tf),length(X,N),N>0,
  meta_trace(T2,Tf,S2,Sf,MD2,MDf).

%binds the free variable in A to any instance
bind_meta_model(M,A,Expr) :-
  meta_model(M,A,Expr),
  free_variables(A,L),
  instance(Free),L=[Free].

get_meta_model_set(M,MS) :-
  setof(meta_state(A,Expr),
  bind_meta_model(M,A,Expr),MS).

meta_step(T1,T2,I1,I2,MMD1,MMD2) :-
  i_event_step(E,P,I1,I2),
  append(T1,[e(E,P)],T2),
  update_meta_data(E,I2,MMD1,MMD2).

update_meta_data(E,I,model_data(MExprs1,Fut1,PFut1),
    model_data(MExprs2,Fut2,PFut2)) :-
  update_meta_exprs(E,I,MExprs1,MExprs2,AFut1,APFut1),
  pre_process_exprs(Fut1,PFut1,FFut1,FPFut1),
  update_futgoals(meta,step(E,I),FFut1,AFut2,APFut2),
  update_prohibgoals(meta,step(E,I),FPFut1,APFut3),
  append(AFut1, AFut2, Fut2),
  flatten([APFut1,APFut2,APFut3],PFut2).

update_meta_exprs(E,I,MS1,MS2,AddedFut,AddedFutP) :-
  update_meta_exprs_h(E,I,MS1,MS2,[],AddedFut,[],AddedFutP).

update_meta_exprs_h(_,_,[],[],Fut,Fut,FutP,FutP).
update_meta_exprs_h(E,I,[meta_state(E,Expr)|MSR],MS2,
    Fut1,Fut2,FutP1,FutP2) :-
  eval_once(meta,step(E,I),Expr,AddedFut,AddedFutP,true),
  update_meta_exprs_h(E,I,MSR,MS2,Fut1,FutM,FutP1,FutPM),
  append(AddedFut,FutM,Fut2),append(AddedFutP,FutPM,FutP2).
update_meta_exprs_h(E,I,[meta_state(A,Expr)|MSR],
    [meta_state(A,Expr)|MS2],Fut1,Fut2,FutP1,FutP2) :-
  E\=A, update_meta_exprs_h(E,I,MSR,MS2,Fut1,Fut2,
  FutP1,FutP2).

%=====================================
%   Meta Activity state machine
%=====================================

%inst(ID,TRACE,PROCESS,STATUS)
initial_i_states(L) :-
  setof(inst(I,[],none,new),instance(I),L).

i_event_step(start(X),P,I1,I2) :-
  select(inst(X,T,_,new),I1,IR),
  process_mapping(_,P),
  I2=[inst(X,T,P,running)|IR].
i_event_step(finish(X),P,I1,I2) :-
  select(inst(X,T,P,running),I1,IR),
  I2=[inst(X,T,P,done)|IR].

i_event_step(run(X,A,Xs),P1,I1,I2) :-
  member(inst(X,T1,P1,running),I1),
  get_states(S1,T1),
  replace_state(A,Xs,S1,_),
  append(T1,[event(A,Xs)],T2),
  replace_i_state(X,
    inst(X,T2,P1,running),I1,I2).
i_event_step(migrate(X),P2,I1,I2) :-
  select(inst(X,T,P1,running),I1,IR),
  process_mapping(_,P2),
  P1\=P2,
  I2=[inst(X,T,P2,running)|IR].

replace_i_state(X,Templ,
  [inst(X,_,_,_)|R],[Templ|R]).
replace_i_state(X,Templ,
  [inst(Y,Tr,Pr,St)|R],
  [state(Y,Tr,Pr,St)|R2]) :-
  X\=Y,replace_state(X,Templ,R,R2).

%=====================================
%   Meta trace filtering and printing (Omitted)
%=====================================

%=====================================
%  Measurement data collection (Omitted)
%
%=====================================
```