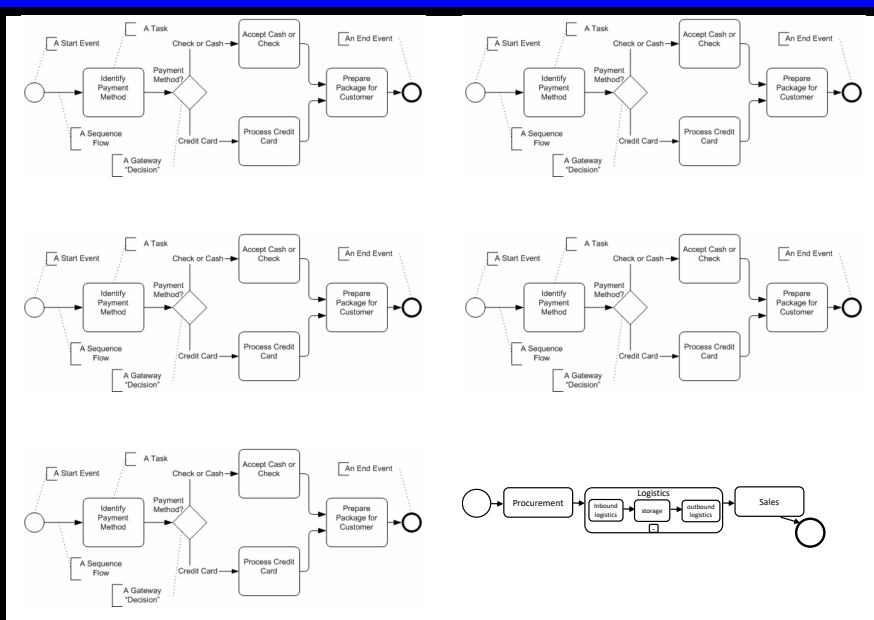
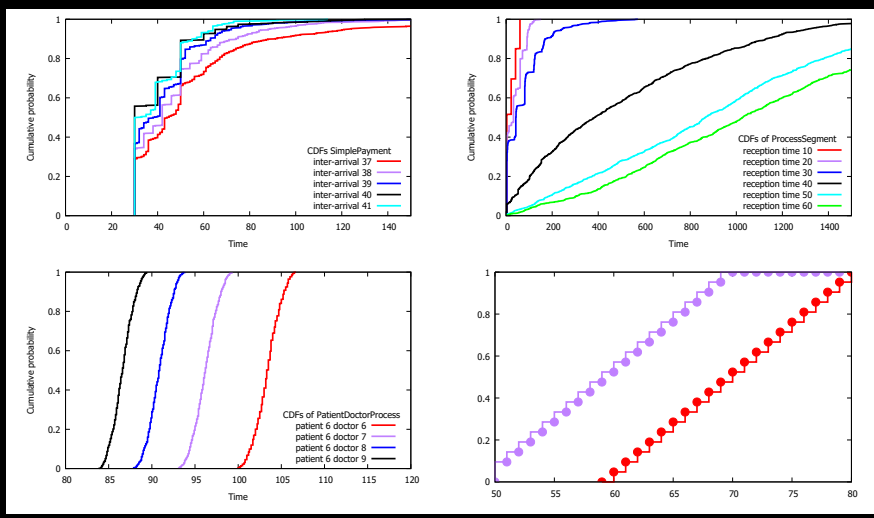
# Evaluating the Performance of Business processes using iDSL

**Written by Freek van den Berg**
**Supervised by Jozef Hooman and Patrick van Bommel**




Radboud Universiteit

## Abstract

Large scale globalization has required a business to be able to connect to other businesses in an automatically manner, often driven by the internet. To separate the business concerns from the software, a business process layer is usually added on top of the software that decouples the business from the software. A business process is a collection of related activities that are each performed by an assigned resource that possesses the right capabilities. Generally, the number of resources of a business is finite due to cost reasons. Consequently, the performance of a business, the rate at which business processes are performed, depends much on the way activities are assigned to resources.

In order to determine this business performance in general, we have selected six typical business processes from literature in the Business Process Model and Notation language and formalized them. After this, we automatically transformed each formal business process into an equivalent iDSL performance model using transformations in the Xtend language. iDSL is a performance evaluation language and toolset for service-oriented systems that presents its results visually. iDSL is based on process algebra language Modest under the hood. Executing an iDSL model then yields Cumulative Distribution Function graphs that convey for each latency the probability that the business process has ended after that time, for many design variations. In case of nondeterminism, iDSL returns an upper and lower bound latency.

Each performance outcome is linked to the original business process for validation. The performance outcomes display the behaviour we would expect from the business processes, which gives us confidence that they are correct. We also address ways to improve a business process. For instance, a resource with a high utilization is susceptible to be a bottle neck and might better have less activities assigned to it. For another example, extending the business with new resources might be a way to counteract bottle necks, albeit at a higher cost for the business.

In future work, we would like to use a performance language other than iDSL to assess generalizability of the approach, test the approach on more extensive business processes for extensibility, and put more emphasis on the correctness of the generated iDSL models.

# Contents


**1 Introduction** **5**

**2 Background on business processes** **11**
2.1 Definition of a supply chain . . . . . 11
2.2 Definition of a value chain . . . . . 11
2.3 Definition of a business . . . . . 11
2.4 Definition of a business activity . . . . . 11
2.5 Definition of a resource . . . . . 12
2.6 Definition of a business process . . . . . 12

**3 The business process model & notation BPMN** **13**
3.1 A simple process . . . . . 14
3.2 A segment of a process with more details . . . . . 17
3.3 A process with pools . . . . . 19
3.4 A segment of a process with lanes . . . . . 21
3.5 A high-level business process . . . . . 22
3.6 A hierarchical business process . . . . . 24

**4 Different performance evaluation approaches** **27**
4.1 Performance models . . . . . 27
4.1.1 Automata . . . . . 27
4.1.2 Queueing networks . . . . . 29
4.1.3 Petri Nets . . . . . 29
4.1.4 Dataflows . . . . . 30
4.2 Performance evaluation techniques . . . . . 30
4.2.1 Back-of-the-envelope . . . . . 30
4.2.2 Discrete-event simulation . . . . . 30
4.2.3 TA model checking . . . . . 31
4.2.4 PTA model checking . . . . . 32
4.3 Performance approaches . . . . . 32
4.3.1 Modest . . . . . 32
4.3.2 STORM . . . . . 33
4.3.3 PIPE2 . . . . . 33
4.3.4 POOSL . . . . . 33
4.3.5 UPPAAL . . . . . 33
4.3.6 PRISM . . . . . 34
4.3.7 iDSL . . . . . 34
4.4 Selection of performances model and approach . . . . . 34

**5 The performance evaluation approach iDSL** **36**
5.1 The conceptual model of IDSL . . . . . 36
5.2 An business-related IDSL instance . . . . . 37
5.2.1 A Business iDSL Process . . . . . 38
5.2.2 A Business iDSL Resource . . . . . 39
5.2.3 A Business iDSL System . . . . . 39
5.2.4 A Business iDSL Scenario . . . . . 39
5.2.5 A Business iDSL Measure . . . . . 40
5.2.6 A Business iDSL Study . . . . . 41
5.3 Transforming the iDSL language into Modest . . . . . 41
5.4 The iDSL performance results . . . . . 46

**6 Extending BPMN for a transformation to iDSL** **49**
6.1 A simple process . . . . . 49
6.2 A segment of a process with more details . . . . . 51
6.3 A process with pools . . . . . 52
6.4 A segment of a process with lanes . . . . . 54
6.5 A high-level business process . . . . . 56
6.6 A hierarchical business process . . . . . 58

**7 Performance evaluation results** **61**
7.1 A simple process . . . . . 62
7.2 A segment of a process with more details . . . . . 64
7.3 A process with pools . . . . . 66
7.4 A segment of a process with lanes . . . . . 68
7.5 A high-level business process . . . . . 70

**8 Conclusion** **74**

**A The regular and extended BPMN grammar** **76**
A.1 The regular BPMN grammar . . . . . 76
A.2 The extended BPMN grammar . . . . . 77

**B Graphviz code generator and code examples** **80**
B.1 The Graphviz code generator . . . . . 80
B.2 The Graphviz code examples per use case . . . . . 84
B.2.1 A simple process . . . . . 84
B.2.2 A segment of a process with more details . . . . . 84
B.2.3 A process with pools . . . . . 85
B.2.4 A segment of a process with lanes . . . . . 85
B.2.5 A high-level business process . . . . . 86
B.2.6 A hierarchical business process . . . . . 86

**C The iDSL code generator, grammar and code** **87**
C.1 The iDSL generator . . . . . 87
C.1.1 The high-level code of the iDSL generator . . . . . 87
C.1.2 From a BPMN "Business Process" to an iDSL "Process" 89
C.1.3 From a BPMN "Resource" to an iDSL "Resource" . . 93
C.1.4 From a BPMN "Mapping" to an iDSL "System" . . . 94
C.1.5 From a BPMN "Scenario" to an iDSL "Scenario" . . . 94
C.1.6 From a BPMN "Measure" to na iDSL "Measure" . . . 94
C.1.7 From a BPMN "Design Space" to an iDSL "Study" . 94
C.2 The iDSL grammar . . . . . 97
C.3 The iDSL code per use case . . . . . 100
C.3.1 A simple process . . . . . 100
C.3.2 A segment of a process with more details . . . . . 101
C.3.3 A process with pools . . . . . 102
C.3.4 A segment of a process with lanes . . . . . 104
C.3.5 A high-level business process . . . . . 106

**D Additional performance results** **107**
D.1 A segment of a process with more details . . . . . 107
D.2 A high-level business process . . . . . 108

# 1 Introduction

**Context.** In the last centuries, large-scale globalization has led to a worldwide interaction and integration between people, companies, and governments [7, 167]. Moreover, these interactions implied a growth of international trade. Particularly, the rise of the internet has led to an exponential grow of online shopping [168] and business-to-business services [25, 100]. Therefore, in order to be competitive nowadays, a business has to be efficient by automating as much as its operations possible [91] and easily connect to other businesses [166].

Traditionally, the way the business worked was stored in the software of a business [158], which has some major drawbacks: the way of working is hard to identify and communicate, and the technology change cycle of software is 6-10 years versus 1-2 years for a business strategy [34]. The latter implies that the business will continuously have to deal with a lot of legacy software.

More recently, a so-called business process layer has been used on top of the software that decouples the business processes from the software [28, 158]. A business process is a collection of related activities that are performed in a certain order [88, 162]. An activity can be implemented, among others, as a web service, a local computation, a database search or a user query [142]. Note that a web service activity can be used to connect a business to other businesses [124].

Managing business processes entails many activities [78], including: (i) design; (ii) modelling; (iii) execution; (iv) monitoring; (v) optimization; and (vi) re-engineering. In this paper, we address optimization [10, 155], which is originally concerned with retrieving performance information of the business process from the modeling and/or monitoring phase. On top of that, we predict this process performance information using state-of-the-art performance evaluation techniques, which enables us to evaluate different business process alternatives before actually implementing them.

**Research questions.** Given this context, we pose the following research question.

How to evaluate the performance of business processes?

We define six sub-questions that need to be answered first in order to answer this research question.

**Q1** What is a business process?
**Q2** What formalisms exist for describing a business process?
**Q3** What does performance mean for a business process and why is it relevant?

**Q4** Which techniques for performance evaluation exist?
**Q5** Which technique is most suitable to be applied to business processes?
**Q6** How can this technique for performance evaluation be applied to business processes?

**Plan-of-approach.** We propose the following six subsequent steps to answer the aforementioned research questions, respectively.

**P1** Informally and intuitively describe what business processes are.
**P2** Select a suitable method for modeling business processes in a formal manner.
**P3** Provide examples of situations in which it is useful to know the performance of business processes.
**P4** Investigate the state-of-the-art of performance evaluation techniques.
**P5** Select a suitable performance evaluation technique to apply to business processes.
**P6** Apply the selected performance evaluation technique to the business process examples and interpret the results.

**A Domain-Specific Language approach.** A Domain-Specific Language approach (DSL) is proposed for specifying business processes, because it delivers an approach that is: (i) formal; (ii) consise; (iii) expressive; and, (iv) understandable by domain-expects in the business domain. Mature language workbenches that provide extensive DSL support are Eclipse Xtext [3, 4, 54] and Jetbrains MPS [112, 156]. We have realized the work performed in this thesis using the commonly used Eclipse Xtext tooling.

The way-of-working is a pipeline that is graphically depicted in Figure 1. The overarching goal is to go from an informal business process in Business Process Model and Notation (BPMN, [29, 38, 43, 108, 130]) as input towards a performance model in (iDSL, [145, 146, 150, 151]) format as output. This is accomplished by performing the following subsequent steps.

1. The approach starts with collecting informal BPMN instances (cf. Figure 1a) to gain insight in what BPMN instances are and in what variations they come. As will turn out, exemplary BPMN diagrams can be found in literature [163] that each highlight a certain aspect of an BPMN instance.
2. The informal BPMN instance is formalized in a manual way (cf. Figure 1b) by constructing a grammar in Xtext (cf. Figure 1e). The BPMN syntax (cf. Figure 1d) is derived from the grammar. The formal BPMN instance (cf. Figure 1c) has to adhere to the syntax. Also, grammar describes how a Ecore model, i.e., the core (meta-)model at the heart of Eclipse Modeling Framework (EMF), is derived from a textual notation.

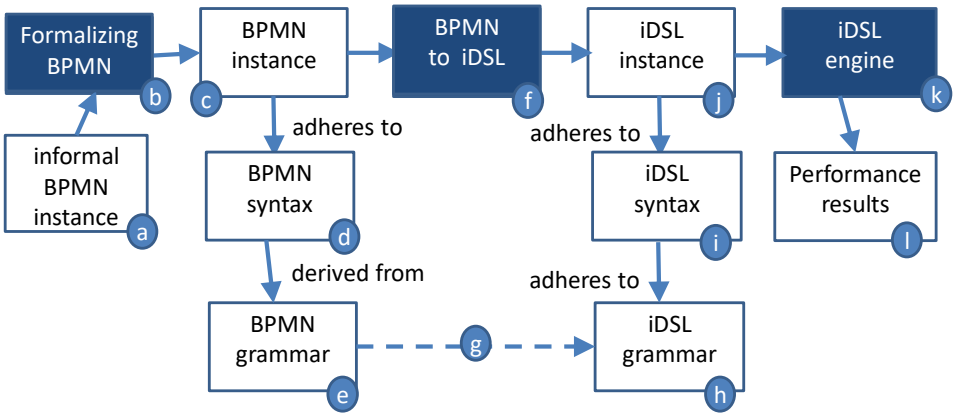



Figure 1: A Domain-Specific Language approach.

3. The performance language iDSL was developed outside the scope of this thesis. The iDSL grammar (cf. Figure 1h) is written in Xtext and yields a syntax (cf. Figure 1i) to which an iDSL instance (cf. Figure 1j) must adhere. The transformation from BPMN to iDSL (cf. Figure 1f) connects the concepts and relations of both grammars (cf. Figure 1g) by reading from the BPMN Ecore model and writing a textual iDSL instance. Concretely, this transformation is written in the Xtend language [4], a higher-level and more expressive version of Java that is transformed into Java under the hood.
4. Once an iDSL instance has been generated, it becomes possible to run it in the iDSL engine (cf. Figure 1k) and derive various performance results using the following three techniques. First, discrete-event simulation leads to latency breakdown charts, cumulative distribution graphs, and latency bar charts. Second, traditional model checking yields absolute latency bounds, i.e., an absolute minimum and maximum. Third and finally, probabilistic model checking leads to exact latency distributions. In case of nondeterminism, a minimum and maximum latency distribution is the result. Summarized, it is possible to derive many performance results (cf. Figure 1l) for a BPMN instance using this approach.

**Related work.** Besides the aforementioned iDSL, we mention two more DSL approaches that comprise a language and toolset. First, aDSL [144] comprises a DSL to specify a Cyber-Physical System in terms of its subsystems, parts, operation modes and requires. The adherence of requirements is then automatically derived for different designs.

Second, Quality and Resource Management Language (QRLM, [143]) comes with a DSL to specify heterogeneous hardware/software systems, their

composition and configurations. A QRML model then automatically transformed into a constraint satisfaction problem which is used to automatically derive valid configurations using the theorem solver Z3 [37].

**Business related concepts.** In the following, we introduce concepts that are related to a business. A *business* is often being referred to as "a legally recognized organization designed to provide goods and/or services to consumers or tertiary business in exchange for money" [110]. Hence, a business process can be seen as "dividing a business operation into distinct operations" [132], or "a collection of activities that takes one or more kinds of input and creates an output that is of value to the customer" [65]. Relatedly, a workflow includes the procedures, people and tools involved in each step of a business process [161].

**Business notation.** To specify and communicate business processes, several business process notations or formalisms can be used. We mention a few of them without attempting to be exhaustive. First, a Petri net [61, 123, 153] is a generic mathematical modeling language for describing distributed systems and, hence, very suitable for expressing business processes.

Second, a workflow [153, 154] comprises an arranged and repeatable pattern of a business activity. In a workflow, resources are processes that transform materials, provide services, or process information.

Third, Cognition enhanced Natural language Information Analysis Method (CogNIAM [22]) is a conceptual fact-based modelling method. It structures knowledge, gathered from people, documentation and software, using four dimensions, i.e., data, rules, processes and semantics.

Fourth, Unified Modelling Language (UML, [56]) is a generic language. However, UML provides extensions for business processes, e.g., Eriksson-Penker extensions [53, 165] provide a framework for UML business processing model extensions to be used by a system designer to model its business.

Fifth and finally, Business Process Model and Notation (BPMN, [29, 130, 142, 163]) can be used to specify business processes in a visual way. For this purpose, a BPMN diagram consists of flow objects, connecting objects, swim lanes and artifacts. Although BPMN shows the flow of messages and the association of data artifacts to activities, it is not a data flow diagram.

Business performance measurement (BPM, [138]) is the process of collecting, analyzing and/or reporting information regarding the performance of a business. Various models, systems and frameworks for BPM exist [1, 35, 82, 85, 93, 106]. Nevertheless, the balanced scorecard [83, 84] is regarded to be the most commonly-used strategic tool. It identifies four distinct perspec-

tives, viz., financial, customer, internal business processes, and learning and growth.

A performance metric measures the behavior, activities, and/or performance of a business. A metric should be valuable to stakeholders, including customers, shareholders and employees [127]. In this thesis, the focus will be on the metrics related to the internal business processes of the balanced scorecard, viz., the inventory, orders, resource allocation, cycle time and quality control [2], as defined in [149]. They are applied, as follows. First, the process latency can be used to determine whether a product will generally arrive in time for a customer. Second, the process jitter can be used to predict which proportion of costumers will receive their product late. Third, the process throughput can be used to estimate how many simultaneous customers a business can handle. Fourth and finally, resource utilizations and queue sizes can be used to identify internal bottlenecks.

**Performance evaluation.** We recognize two distinct categories of performance evaluation techniques. On one hand, there are tools using generalized Petri nets [101, 104] as their underlying model, e.g., Petri Net Editor 2 (PIPE2, [6, 46]) and TimeNET [60, 171]. On the other hand, tools use (a subset) of stochastic timed automata [71] as their underlying model, such as UPPAAL [19, 96], POOSL [48, 135], Storm [40] and Modest [68, 69].

**Structure.** The remainder of this thesis is structured (as graphically partially depicted in Figure 2), as follows. In Section 2, background on different business process models is provided. Section 3 contains details of the selected notation for business process models named BPMN, using typical business processses that are used as a running example. In Section 4, an overview of performance evaluation approaches is conveyed. In Section 5, background on the chosen performance evaluation approach called iDSL is provided. In Section 6, BPMN is extended to support a transformation into iDSL. The actual transformation is described in Appendix C. Section 7 conveys the performance evaluation results. They are applied to the typical business processes of Section 3. Finally, Section 8 concludes the paper.

Additionally, Appendix A contains the regular and extended BPMN grammar for Section 3 and 6, respectively. Appendix B presents the Graphviz code generator and generated code for the typical business processes of Section Section 3. Appendix C conveys the iDSL code generator illustrated by code for the typical business processes of Section 3 and, hence, provides the operation semantics of BPMN. Finally, Appendix D contains performance results in addition to the ones presented in Section 7.

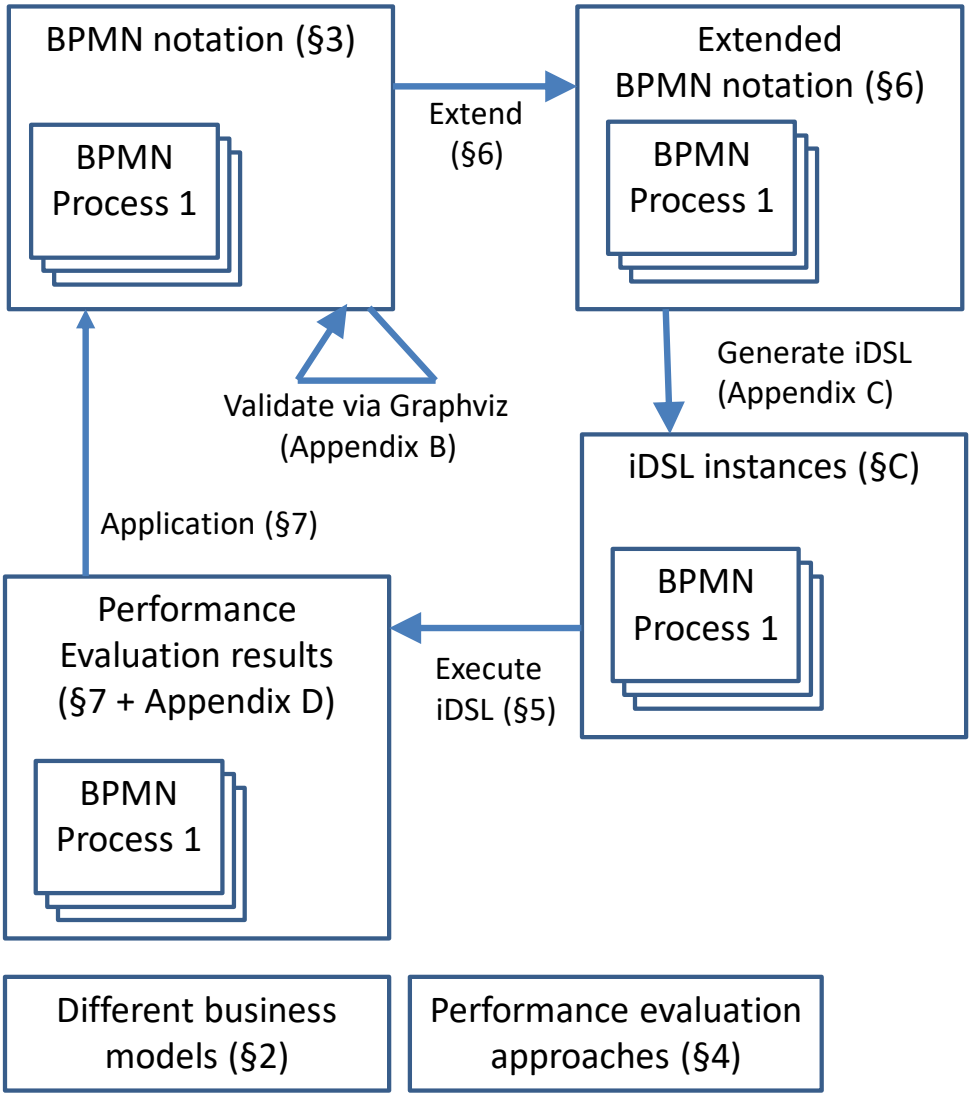



Figure 2: The structure of the main parts of this thesis.

# 2 Background on business processes

In this section, background on business processes is provided. We discuss the following concepts: Section 2.1 defines a supply chain, Section 2.2 a value chain, Section 2.3 a business, Section 2.4 a business activity, Section 2.5 a resource, and finally Section 2.6 a business process.

## 2.1 Definition of a supply chain

A *supply chain* is a system of organizations, people, activities, information, and resources involved in moving a product or service from supplier to customer. Its activities involve the transformation of natural resources, raw materials, and components into a consumer product [90]. Supply chains link so-called value chains [105], which are defined next.

## 2.2 Definition of a value chain

A *value chain* is a set of activities that a firm operating in a specific industry performs in order to deliver a valuable product or service for the market. The concept was first described by Michael Porter [116].

A value chain takes a process view of organizations, namely the idea of seeing a manufacturing or service organization as a system, made up of subsystems each with inputs, transformation processes and outputs. Inputs, transformation processes, and outputs involve the acquisition and consumption of resources, such as money, labor, materials and land. The way value chain activities are carried out influences costs and profits [116, 125].

## 2.3 Definition of a business

A *business* implements a value chain. In literature, a business has has been described as "a legally recognized organization designed to provide goods and/or services to consumers or tertiary business in exchange for money" [109]. Moreover, a business must use its resources as efficiently and effectively as possible to compete with other organizations.

## 2.4 Definition of a business activity

A business can be split up into multiple *business activities* using so-called differentiation steps, as follows. In case of horizontal differentiation, a business is split up in parallel activities, e.g., three processes for dealing blue, red and green bikes, respectively. In case of vertical differentiation, a business is split up in subsequent activities, e.g., the procurement, manufacturing and sales of bikes. Both types of steps can be applied recursively and many times, leading to smaller and more specific business activities [172]. Note

that this line of reasoning is similar to a value chain (of Section 2.2), which advocates the idea that a business consists of subsystems.

## 2.5 Definition of a resource

A business activity is performed by a *resource*. A resource can only perform a business activity when it has the right capabilities to perform it. In literature, resources are defined as "the inputs or the factors available to a company which helps to perform its operations or carry out its activities" [9]. Resources do not lead to productivity in isolation [9], hence, coordinating them is of vital importance. In order for a business to be competitive, i.e., being able to outperform its competitors, its individual resources have to be both valuable, rare, inimitable and non-substitutable [107, 131].

## 2.6 Definition of a business process

In literature, a *business process* has been described in different ways of which we mention three. First, Hammer et al. [65] state that "a business process is a collection of activities that takes one or more kinds of input and creates an output that is of value to the customer."

Second, Rummler et al. [126] define "a business process as a series of steps designed to produce a product or service. Most processes (...) are cross-functional, spanning the "white space" between the boxes on the organization chart. Some processes result in a product or service that is received by a customer external to the organization. We call these primary processes. Other processes produce products that are invisible to the external customer but essential to the effective management of the business. These are called support processes."

Third, Johansson [80] describes "a business as a set of linked activities that take an input and transform it to create an output. Ideally, the transformation that occurs in the process should add value to the input and create an output that is more useful and effective to the recipient either upstream or downstream." Shortly, all these three definitions convey that a business process comprises a set of business activities.

# 3 The business process model & notation BPMN

In the related work of the introduction, a number of business process notations have been introduced. In this thesis, we have selected Business Process and Model Notation (BPMN, [29, 164]) to be the notation for business processes, because: (i) making use of a visual language enables the communication of a wide variety of information and to different audiences [163]; and, (ii) BPMN is designed to cover many types of modeling and at different levels of abstraction [163]. Because of this, BPMN provides a large degree of freedom.

This degree of freedom makes it difficult to unambiguously define the operational semantics of BPMN. Nevertheless, multiple attempts have been made to make BPMN executable. For instance, BPMN has been translated into BPEL [111] as an intermediate step, but the conceptual mapping between both languages remains unclear [111]. For another instance, BPMN has been mapped to Petri Nets [43], which conveyed deficiencies in the BPMN specification. Finally, BPMN has been transformed into the functional language iTasks [98, 114], resulting into executable specifications of interactive work flow systems for the web [142].

In this section, we formalize the grammar of BPMN in an incremental way, i.e., we define the grammar in Xtext. For this purpose, we use five typical BPMN examples [163] (in Section 3.1 till 3.5). Moreover, a sixth BPMN example is included in Section 3.6, which is inspired by a business-related iDSL instance (as discussed in Section 5.2).

For each example, we provide: (i) the actual BPMN diagram from literature [163] with an informal description; (ii) the BPMN model instance in textual form; and, (iii) an automatically generated Graphviz [51, 63] visualization to validate whether the BPMN model instance contains all information of the actual BPMN diagram. We do not claim that it is easy to transform a BPMN diagram into a BPMN model instance. We do not intend to make the modeling of a BPMN model instance as easy possible here, but merely want to show the formalization step is possible. For instance, a graphical IDE could be implemented to make the modeling easier.

Furthermore, Section 6 extends the BPMN grammar to support a transformation into performance evaluation language iDSL. The BPMN grammar is presented in Table 37 of Appendix A and the Xtend code that transforms BPMN model instances into Graphviz code in Table 39 of Appendix B.

### 3.1 A simple process

The "A simple process" example is concerned with customers paying after they have made an order. In a real business, there can be thousands of customers going through this process simultaneously. In Figure 3, the BPMN diagram of the "A simple process" example is displayed [163, Figure 1]. It can be seen that a BPMN graph consists of boxes, i.e., activities, which are connected via arrows, i.e., flows that mandate in what order activities are executed. For instance, activity "AcceptCashOrCheck" precedes "PreparePackageForCumstomer" in Figure 3. This means activity "AcceptCashOrCheck" has to be completed first, before activity "PreparePackageForCumstomer" can start executing.

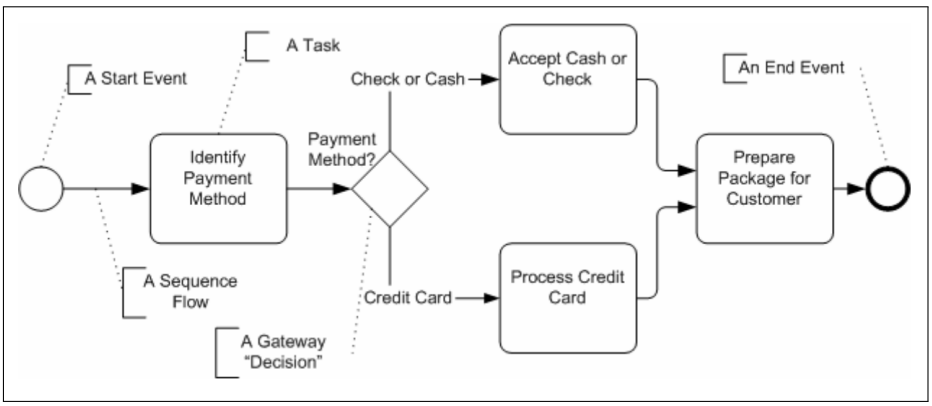



Figure 3: An informal BPMN diagram "A simple process"

On top of this, a choice is introduced in Figure 3, which is a selection that is made between two or more continuing flows after a preceding activity has finished executing. In this example, either activity "AcceptCashOrCheck" or activity "ProcessCreditCard" gets executed after the completion of activity "PaymentMethod", but not both.

Figure 3 is formalized by providing a grammar and BMPN model instance. For this purpose, Table 37 comprises the complete BPMN grammar, whereas Table 1 the comprises subset of this grammar (cf. Table 37, lines 4-8, 13-14 and 19-25) that is relevant in this section. Table 1a displays a value chain, which can recursively be composed using a variety of components. Here, we only concern:

1. the empty value chain (cf. Table 1c), denoted by ".", is a specific value chain that is used to terminate a value chain;
2. a business activity value chain (cf. Table 1b) is a value chain that contains a preceding business activity and comes in three flavours;
3. the alternate value chain (cf. Table 1d), denoted by "+", is used to split a value chain into multiple alternatives and consist of one business activity and a number of value chains;

Table 1: The grammar of regular BPMN regarding A simple process

(a) value chain

```
1. ValueChain:
2.  RepeatValueChain | EmptyValueChain | BusActValueChain;
```

(b) business activity value chain

```
1. BusActValueChain:
2.  ParallelValueChain | AlternateValueChain | SequentialVal-
    ueChain;
```

(c) empty value chain

```
1. EmptyValueChain:
2.  {EmptyValueChain} '.';
```

(d) alternate value chain

```
1. AlternateValueChain:
2.  ba=BusinessActivity '+' '(' vcs+=ValueChain+ ')';
```

(e) sequential value chain

```
1. SequentialValueChain:
2.  ba=BusinessActivity ';' vcs+=ValueChain;
```

(f) business activity

```
1. BusinessActivity: name=ID;
```

4. the sequential value chain (cf. Table 1e) denoted by ";", orders activities and consists of a business activity; and,
5. a business activity, representing an atomic activity, which solely uses a string for identification (cf. Table 1f)

Table 2 presents the BPMN model instance of "A simple process". It starts with the fixed keyword "Regular BPMN model" (line 1). Next, it comprises a sequential value chain for "IdentifyPaymentMethod" at its highest level of hierarchy (line 2), which is followed by an alternate value chain "PaymentMethod" (line 2) at a lower level. The alternate value chain has two options,

Table 2: The BPMN model instance "A simple process"

```
1. Regular BPMN model
2. IdentifyPaymentMethod; PaymentMethod + ( Accept-
   CashOrCheck; PreparePackageForCustomer;. ProcessCreditCard;
   PreparePackageForCustomer;. )
```

namely “AcceptCashOrCheck” and “ProcessCreditCard”, in line with two options in Figure 3. Either option continues in the same way, namely with an activity “PreparePackageForCustomer” that is succeeded by an empty value chain to terminate the process.

Note that the activity “PreparePackageForCustomer” is used twice in the BPMN model instance, because this makes the underlying grammar simpler by only having to allow tree structures opposed to graphs.

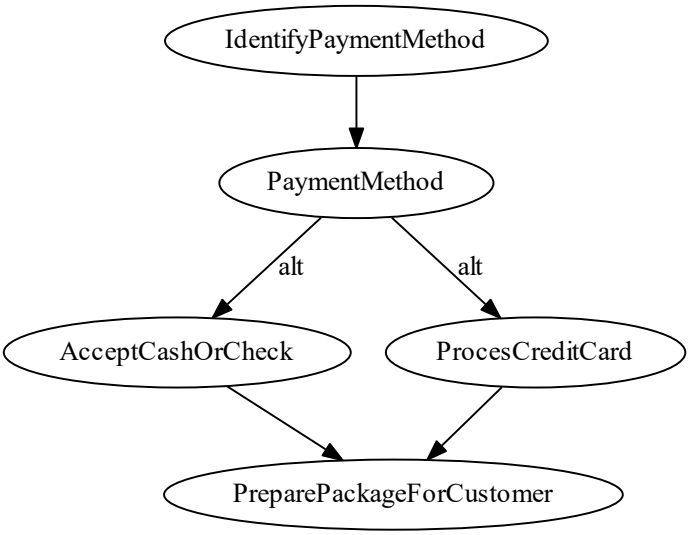



Figure 4: The BPMN diagram created using GraphViz “A simple process”

The grammar and BPMN model instance are validated by reconstructing the original BPMN graph of Figure 3 solely on the basis of the BPMN model instance of Table 2. For this purpose, GraphViz code (see Table 40) is automatically generated from Table 2 using a BPMN-to-Graphviz algorithm that has been created for this thesis (see Table 39 in Appendix B). The GraphViz code yields Figure 4 after executing GraphViz. Comparing Figure 3 with Figure 4 reveals that the structure of the former BPMN graph has indeed remained intact.

### 3.2 A segment of a process with more details

The "A segment of a process with more details" example addresses a business with a dynamic procurement facility. Namely, when a good needs to be purchased, a quote is send to a number of suppliers after which the cheapest is selected or no supplier. Also in this case, just like in the previous example, it is possible that many processes occur at the same time.

Figure 5 contains the BPMN diagram of the "A segment of a process with more details" example [163, Figure 2]. In extends the concepts of Section 3.1 with a repeat construct. This construct makes it possible to repeat a value chain of one or more activities for one or more times, followed by a value chain that is performed only once. For instance, the value chain consisting of activities "SendRFQ", "ReceiveQuote" and "AddQuote" may be executed one or more times in Figure 5 and is finally followed by activity "FindOptimalQuote", which is only performed once.

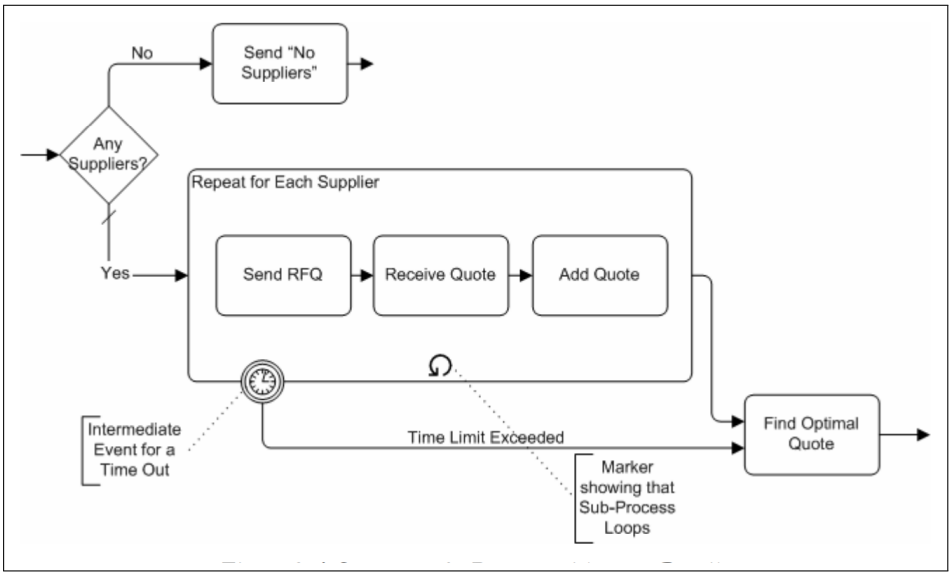



Figure 5: An informal BPMN diagram "A segment of a process with more details"

Figure 5 is formalized by extending the BPMN grammar with the so-called repeat & composition operator (see Table 37, lines 10-11 and Table 3). This operator consists of a value chain, an infix operator "><", and a second value chain. The first value chain is used to define both the repeating and composition part whereas the second value chain refers to the part that is only executed once afterwards. In this section both the repeating and composition functionalities of the operation are used.

The BPMN model instance as shown in Table 4 comprises an alternate value chain (cf. Table 1d) with a business activity "AnySuppliers" at its

Table 3: The grammar of regular BPMN: the repeat & composition operator

```
1.  RepeatValueChain:
2.  '('  rvc=BusActValueChain  ')'  '><'  '('
    vcs+=SequentialValueChain ')';
```

highest level (line 2). One alternative comprises the activity "SendNoSuppliers" (line 3) that is terminated via an empty value chain (line 3). The other alternative yields the repeat value chain, which comprises three repeating activities, i.e., "SendRFQ", "ReceiveQuote" and "AddQuote" (line 4), followed by a activity "FindOptimalQuote" (line 4). Activity "FindOptimalQuote" is executed only once and terminated via an empty value chain (line 4).

Table 4: The BPMN model instance "A simple process"

```
1.  Regular BPMN model
2.  AnySuppliers + (
3.  SendNoSuppliers;.
4.  (SendRFQ; ReceiveQuote; AddQuote;.)><(FindOptimalQuote;.)
    )
```

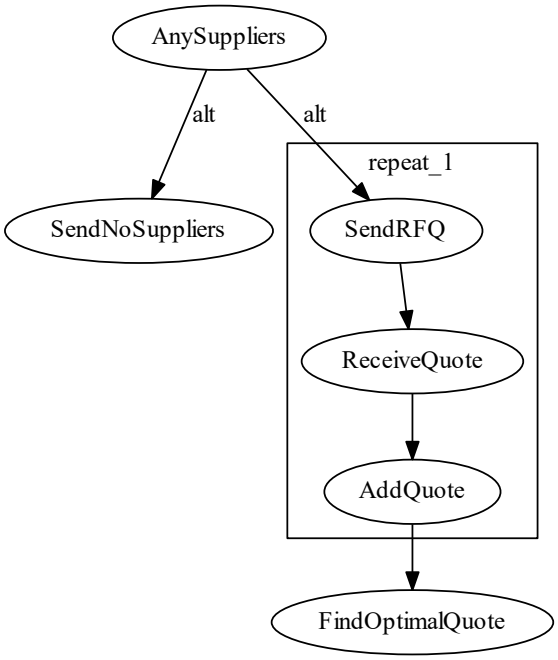



Figure 6: The BPMN diagram created using GraphViz "A segment of a process with more details"

The grammar and BPMN model instance are validated (as in Section 3.1). The original BPMN graph (see Figure 5) and automatically generated Graphviz code (see Table 40) and graph (see Figure 6) display a similar underlying structure. Note that the repeat activity has been represented by a cluster in GraphViz. For this purpose, the BPMN-to-Graphviz algorithm (of Table 39) uniquely numbers each cluster starting by 1. Therefore, the cluster is named "repeat_1".

### 3.3 A process with pools

The "A process with pools" example provides a back and forth interaction between a doctor's office (business) and patient (consumer). It is likely that many patients are in contact with the doctor's office at the same time and in different states of the process. In Figure 7, the BPMN diagram of the "A process with pools" example is revealed [163, Figure 3]. It extends Section 3.1 and 3.2 with pools, which uniquely assign each activity to a resource, also referred to as a lane in BPMN. However, in this section, pools and lanes will only be partially implemented in order to not have to be concerned with resources yet. That is, Section 6 provides an extension for resources. Consequently, only the sequence in which activities are performed will be considered but not the actual lane they are assigned to. For instance, activity "ReceiveDoctorRequest" succeeds activity "SendDoctorRequest" in Figure 7, and activity "SendAppt" succeeds activity "ReceiveDoctorRequest" in turn.

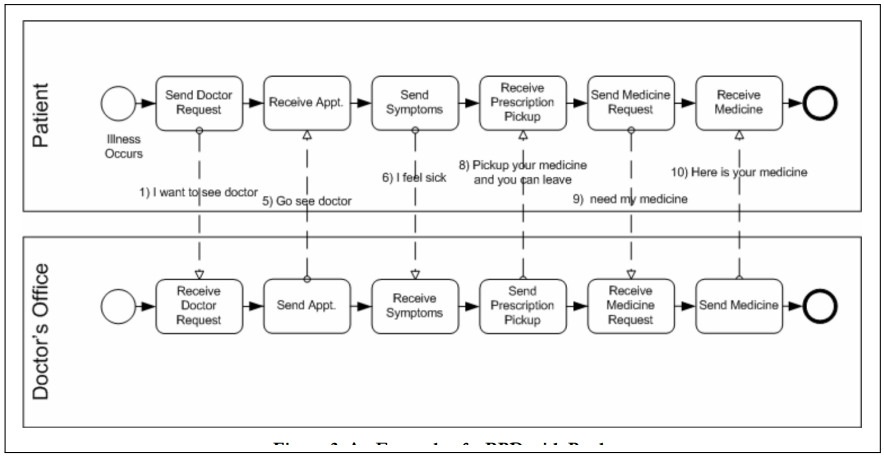



Figure 7: An informal BPMN diagram "A process with pools"

Figure 7 conveys two parallel value chains, viz., one for the "Patient" and another one concerning the "Doctor's Office". They synchronize back and forth via the dotted arrows, which means they can be modeled as a large sequence (see Table 5 and Figure 8), as follows. Activity "SendDoc-

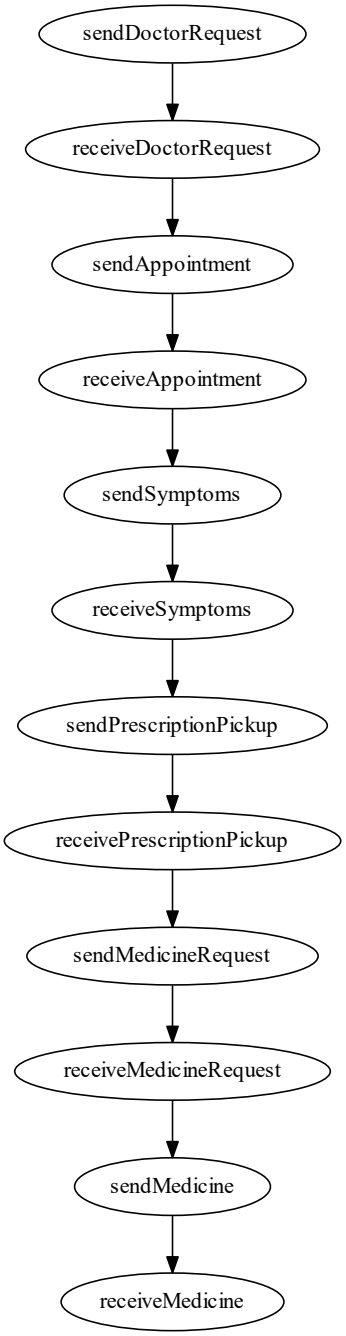



Figure 8: The BPMN diagram created using GraphViz “A process with pools”

torRequest" is performed first, followed by "ReceiveDoctorRequest" due to the first synchronization. Next "SendAppointment" is performed due to a regular sequence, followed by "ReceiveAppointment" due to the second synchronization. This pattern repeats itself a number of times and ends with final activities "SendMedicine" and "ReceiveMedicine".

Table 5: The BPMN model instance "A process with pools"

| | |
|---|---|
| 1. | Regular BPMN model |
| 2. | sendDoctorRequest; receiveDoctorRequest; sendAppointment; receiveAppointment; sendSymptoms; receiveSymptoms; sendPrescriptionPickup; receivePrescriptionPickup; sendMedicineRequest; receiveMedicineRequest; sendMedicine; receiveMedicine;. |

The BPMN instance (see Table 5) comprises a sequence of sequential value chains of which the last one is terminated by a empty value chain. All these concepts have already been introduced in Section 3.1. Therefore, the BPMN grammar of Table 37 does not need to be extended here.

The grammar and BPMN model instance are validated for completeness by reconstructing the original BPMN graph of Figure 7. For this purpose, the automatically generated GraphViz code (see Table 42) yields Figure 8 after executing GraphViz. We compare Figure 7 with 8. As anticipated, the pool and lanes are missing in Figure 8, simply because they have not been taken into account for a start. Instead, Figure 8 displays a sequence of twelve activities whose order resembles the activities of Figure 7.

### 3.4 A segment of a process with lanes

The "A segment of a process with lanes" example is about a web server that forwards one incoming request into two outgoing requests. The main purpose of this example is to show how parallelism is being dealt with and the web server is merely an example. Figure 9 conveys the BPMN diagram of the "A segment of a process with lanes" example [163, Figure 4]. In line with Section 3.3, it contains a pool with lanes albeit used slightly differently. Namely, two lanes are used to express parallelism, i.e., activity "PreparePO" and activity "ApproveRequest" execute in parallel after activity "DispatchToApprover" has finished. Just like in Section 3.3, pools and lanes are only partially implemented in order to not have to be concerned with resources.

Figure 9 is formalized by providing a BPMN grammar and a BPMN model instance. The BPMN grammar is extended with a parallel value chain as conveyed in Table 37 (line 16-17) and Table 6. It consists of a business activity, an operator "||", and a number of value chains. Each of the value chains get executed in parallel.

The BPMN model instance (as shown in Table 7) displays a parallel value chain, which has activity "DispatchToApprover" (line 2). Furthermore, it

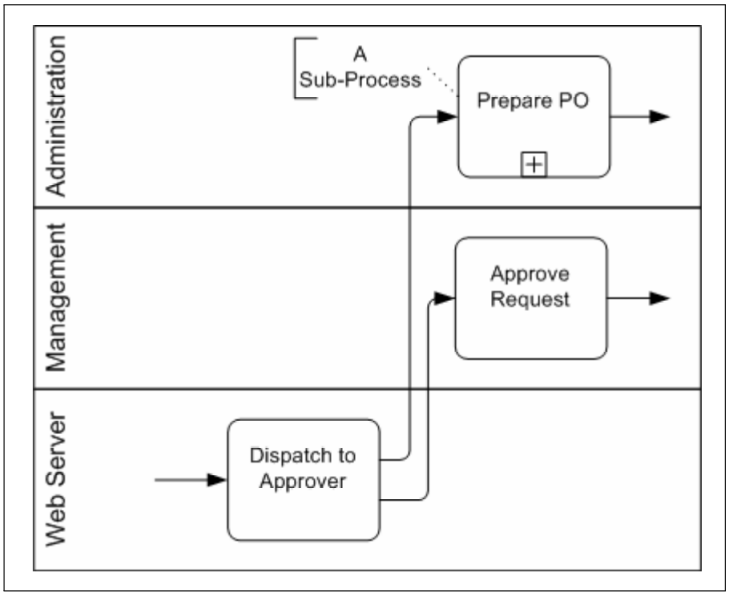



Figure 9: An informal BPMN diagram "A segment of a process with lanes"

Table 6: The grammar of regular BPMN: the parallel operator

```
1.  ParallelValueChain:
2.   ba=BusinessActivity '||' '(' vcs+=ValueChain+ ')';
```

comprises two value chains, namely activity "PreparePO" as well as activity "ApproveRequest" (both at line 2), which are both terminated by an empty value chain.

Table 7: The BPMN model instance "A segment of a process with lanes"

```
1.  Regular BPMN model
2.  DispatchToApprover || ( PreparePO;. ApproveRequest;. )
```

For validation, the BPMN diagram of Figure 9 and the generated Graph-Viz of Figure 10 are compared. Just like in Section 3.3, the pool and lanes are missing due to not taking them into account. Nevertheless, the three activities and the order in which they execute are present.

### 3.5 A high-level business process

The "A high-level business process" example illustrates what can happen when in a sequence of activities one or more can abort at anytime. For instance, when a customer visits a web store, loads items in his shopping basket, but decides not to complete the transaction and pay after all.

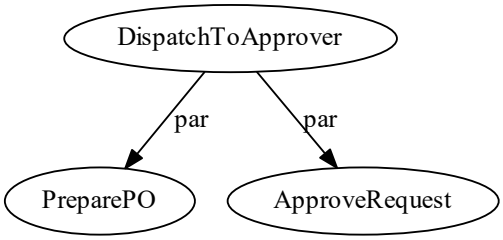



Figure 10: The BPMN diagram created using GraphViz "A segment of a process with lanes"

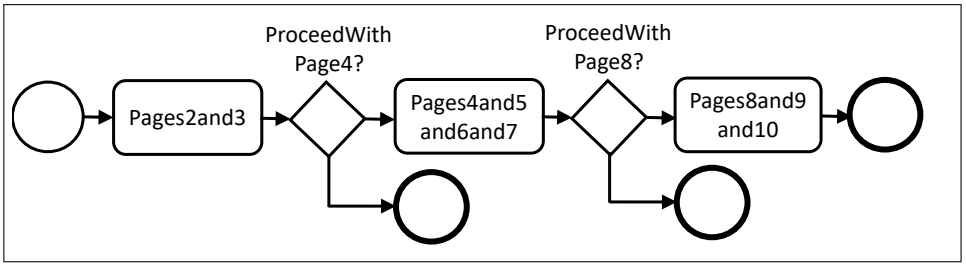



Figure 11: An informal BPMN diagram "A high-level business process"

In Figure 11, the BPMN diagram of the "A high-level business process" example is presented. It is inspired by the BPMN diagram from literature [163, Figure 7], but simplified to make it easier to explain without loosing the essence of abortion. Namely, activity "Page4", activity "Page5and6" and activity "Page7" have been merged into a new activity, which is named "Page4and5and6and7". Furthermore, activity "Page8", activity "Page9" and activity "Page10" have been merged into a new activity named "Page8and9and10".

The simplified BPMN diagram of Figure 11 reveals a process that can abruptly abort after the first and second step, e.g., due to not meeting a functional requirement. For this purpose, the alternate value chain construct of Section 3.1 is used twice to indicate a binary choice between continuing the process or aborting it. To accommodate abortion, a new business activity named "abort" is introduced that represents a business activity in which no actual activity takes place.

Figure 11 is formalized by providing a grammar and BPMN model instance. The BPMN grammar as can be found in is extended with the aforementioned business activity named "abort" cf. Table 37 (line 25) and Table 8).

Table 8: The grammar of regular BPMN: business activity

| | |
|---|---|
| 1. | BusinessActivity: name=ID; |

Table 9: The BPMN model instance "A high-level business process"

| | |
|---|---|
| 1. | Regular BPMN model |
| 2. | pages2and3 + ( abort;. pages4and5and6and7 + ( abort;. pages8and9and10;. )) |

The BPMN model instance (as conveyed in Table 9) consist of a sequential value chain that is alternated at two points using an alternative sequential value chain (line 2). Each time, one of the alternatives is an abort business activity that indicates the process aborts.

The original BPMN diagram of Figure 11 and the generated GraphViz of Figure 12 are compared for validation. The structures are the same, but it can be seen that Graphviz elegantly merges the both dummy business activities into one activity, simply because they have the same name.

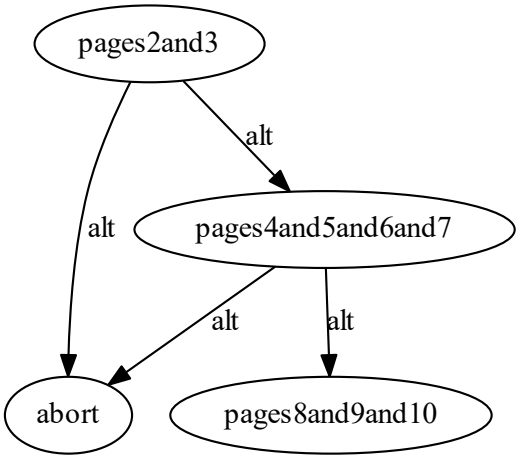


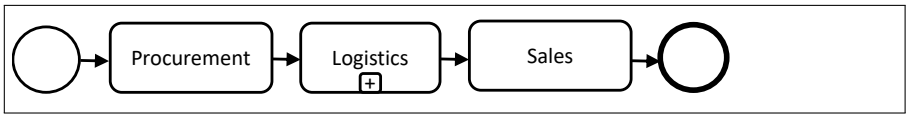

Figure 12: The BPMN diagram created using GraphViz "A high-level business process"

### 3.6 A hierarchical business process

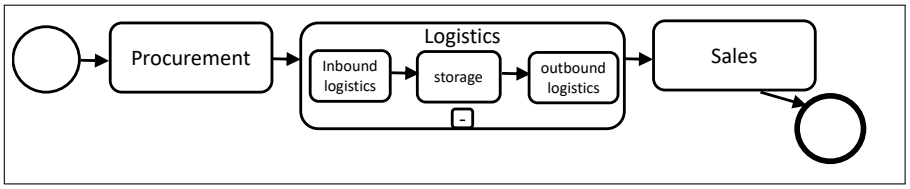
The "A hierarchical business process" example conveys how a decomposition and a composition are modeled in BPMN. A composition is useful, among

others, to: (i) zoom into details; (ii) hide complexity; (iii) reuse parts of a model; and, (iv) group and label parts of a model.

Figure 13 displays the BPMN of the "A hierarchical business process" example. The figure is based on a business-related IDSL instance that includes an hierarchy (see Section 5.2 till 5.4 for more information). The example comprises subsequent activities "Procurement", "Logistics" and "Sales" at it highest level (see Figure 13a). Furthermore, activity "Logistics" can be decomposed in subactivities "Inboud_Logistics", "Storage" and "Outbound_Logistics" (as Figure 13b conveys).




(a) with activity "Logistics" folded




(b) with activity "Logistics" unfolded

Figure 13: An informal BPMN diagram "A hierarchical business process"

Both Figure 13a and 13b are formalized by providing a grammar and BPMN model instance. No extensions are needed to the grammar, because the repeat and composition construct (of Section 3.2) provide composition functionality only whenever the repeated part, i.e., the left hand of the operator, is repeated exactly once. However, it is not possible to store the name of the composite activity, i.e., "logistics", as will turn out.

The BPMN model instance with "logistics" folded (see Table 10) has the subsequent activities "Procurement", "Logistics" and "Sales" (line 2). Besides that, the BPMN model instance with "logistics" unfolded (see Table 11) contains the three constituents of "logistics". That is, the subsequent activities "Inboundlogistics", "Storage", "Outboundlogistics" replace activity "logistics" (line 2).

The original BPMN diagrams of Figure 13 are compared with the generated GraphViz graphs of Figure 14 in twofold for validation. First, we

Table 10: The BPMN model instance "A hierarchical business process" (folded)

| | |
|---|---|
| 1. | Regular BPMN model |
| 2. | Procurement; (Logistics;.) >< (Sales;.) |

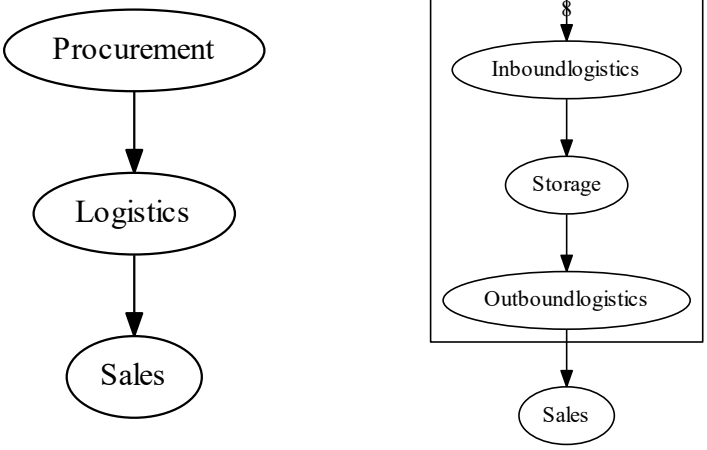
Table 11: The BPMN model instance “A hierarchical business process” (unfolded)

| | |
|---|---|
| 1. | Regular BPMN model |
| 2. | Procurement; (Inboundlogistics; Storage; Outboundlogistics;.) >< (Sales;.) |

consider the case in which activity “logistics” is folded; it can be seen that the BPMN diagram of Figure 13a and the GraphViz graph of Figure 14a both contain three subsequent activities with similar names.

However, it shows that the GraphViz generator (of Table 39) eliminates the cluster when it contains only one activity, i.e., “Logistics”. In this particular case, an empty cluster is created (as in Figure 45), which makes it easier to connect the preceding activity “Procurement” and succeeding activity ”Sales” to activity “Logistics” in the transformation.

Second, we consider the case in which activity “logistics” is unfolded; The BPMN diagram of Figure 13b and the GraphViz graph of Figure 14b both display three subsequent activities. However, the cluster in Figure 14b is unnamed. On the other hand, the second activity “Logistics” does decompose into three activities at a lower level, with similar names.




(a) with activity “logistics” folded (b) with activity “logistics” unfolded

Figure 14: The BPMN diagram created using GraphViz “A hierarchical business process”

# 4 Different performance evaluation approaches

In this section, different performance evaluation approaches are shed a light on. We sum up widespread underlying models in Section 4.1 and evaluation techniques that can be applied to them in Section 4.2. After this, different approaches are introduced in Section 4.3 that systematically apply evaluation techniques to underlying models. Finally, we select one approach from Section 4.3 in Section 4.4, which will be used in the remainder of this thesis.

## 4.1 Performance models

Here, we discuss different models that can be used to evaluate the performance. First, different Automata are discussed in Section 4.1.1. After this, Section 4.1.2 is about Queueing Networks and Section 4.1.3 covers Petri Nets. To conclude, Section 4.1.4 takes Dataflows into account.

### 4.1.1 Automata

In a paper titled "In the quantitative automata zoo" [71], twelve kinds of automata are proposed. The automata are classified using the following four dimensions [71, Section 1]:

First, **nondeterminism** with values: *none*, *discrete* and *continuous*, is either a fixed outcome (none) or a unquantified choice between two or more alternatives. Second, **jump target distributions** with values: *Dirac*, *discrete* and *continuous*, specify whether when jumping out of one automaton state, it is either possible to land in a single target state (Dirac) or in a probability distribution over target states. Third, **stochastic residence times** with values: *none*, *exponential* and *general*, can be used to include the notion of stochastic times spend in a state before going to the next state. Fourth and finally, **flows** with values: *none*, *clocks*, and *general*, enable the deterministic evolution of certain continuous values over time in models.

Table 12 displays for each of the twelve automata (vertically), their expressibility on each of the four dimensions (horizontally). Next, we discuss the twelve automata in a more detail and clarify how they relate to each other.

**Labelled Transition Systems (LTSs, [136])** consist of states that are connected by transitions. Transitions are labeled with elements from a given alphabet. There are a finite number of transitions from each state to other states. Hence, an LTS only accommodates discrete nondeterminism.

**Discrete-Time Markov Chains (DTMCs, [118, 129])** are the simplest state-transition model that include probabilistic choices, i.e., they only permit discrete jump target distributions.

| Model | Non-determinism | Jump target distributions | Stoch. Res. times | Flows |
|---|---|---|---|---|
| **SHA** | Continuous<br>Discrete<br>None | Continuous<br>Discrete<br>Dirac | General<br>Exponential<br>None | General<br>Clocks<br>None |
| **PHA** | Continuous<br>Discrete<br>None | Discrete<br>Dirac | None | General<br>Clocks<br>None |
| **STA** | Continuous<br>Discrete<br>None | Continuous<br>Discrete<br>Dirac | General<br>Exponential<br>None | Clocks<br>None |
| **HA** | Continuous<br>Discrete<br>None | Dirac | None | General<br>Clocks<br>None |
| **PTA** | Continuous<br>Discrete<br>None | Discrete<br>Dirac | None | Clocks<br>None |
| **MA** | Discrete<br>None | Discrete<br>Dirac | Exponential<br>None | None |
| **TA** | Continuous<br>Discrete<br>None | Diarc | None | Clocks<br>None |
| **MDP** | Discrete<br>None | Discrete<br>Dirac | None | None |
| **IMC** | Discrete<br>None | Dirac | Exponential<br>None | None |
| **LTS** | Discrete<br>None | Dirac | None | None |
| **DTMC** | None | Discrete<br>Dirac | None | None |
| **CTMC** | None | Dirac | Exponential<br>None | None |

Table 12: Twelve automata evaluated on four expressibility properties.

**Continuous-Time Markov Chains (CTMCs, [11, 12])** are the continuous-time equivalent of DTMC; they can be seen as stochastic processes that fulfil the Markov property interpreted in real-valued continuous time. The time spend in any state of a CTMC adheres to the exponential distribution, because it is the only memoryless continuous distribution. **Interactive Markov Chains (ICM, [74, 119])** integrate interactive processes and CTMCs. IMCs thereby allow for discrete nondeterminism in addition to CTMCs, which is considered essential to process algebra in general. On

top of that, **Markov Automata (MA [49, 50])** extend IMCs by adding probabilistic choices to immediate transitions.

**Markov Decision Processes (MDPs, [24])** combine the functionality of an LTS and a DTMC. That is, MDPs support discrete nondeterminism and discrete jump target distributions.

**Timed Automata (TA, [21, 76])** are used to model real-time systems [92, 99]. A TA extends a LTS with clocks, which are real variables that synchronously increase over time with rate 1. Clocks are used in guards to ensure that a certain edge can only be taken before a time bound or after a certain residence time. Moreover, **Probabilistic Timed Automata (PTA, [18, 95])** extend a TA with probabilistic choices, whereas **Stochastic Timed Automata (STA, [27])** extend a TA with continuous instead of discrete distributions.

**Hybrid Automata (HA, [8, 73])** are timed automata with the addition of general continuous variables. They are called hybrid because they combine continuous behaviour, viz., the evolution of the continuous variables over time, and discrete events in the form of location changes. A **Probabilistic Hybrid automata (PHA)** extends the features of a HA with a MDP, and a **Stochastic Hybrid Automata (SHA, [58, 64, 75])** combines a PHA and STA.

### 4.1.2 Queueing networks

Queueing networks (QN, [14, 59, 67, 72, 81, 137]) are networks of queues in which a number of queues are connected by customer routings. Queueing theory is the mathematical study of QN. QNs can be evaluated in different ways [59, 72], e.g., a QN can be transformed into a CTMC or a DTMC after which evaluation techniques for transition systems (to be discussed in Section 4.2) can be applied.

A Layered Queueing Network [57, 89, 137] is QN extension where the service time for each job at each service node is given by the response time of a QN at a lower level. Therefore, the service time of the latter QN may again be determined by further nested networks. A layered QN can be analyzed using Differential Equation Analysis [42, 159].

### 4.1.3 Petri Nets

Petri Nets [61, 123] describe distributed systems using places, transitions and arcs. They can be mathematically represented as a bipartite graph. Petri nets are either discrete, continuous or hybrid, and moreover related to discrete, continuous or hybrid automata (cf. Section 4.1.1).

Stochastic Petri nets [5, 81] extend regular Petri Nets by making transitions fire after a probabilistic delay which is determined by a random value. They can be transformed into a CTMC and a DTMC, which enables the application of certain evaluation techniques for transition systems (to be discussed in Section 4.2). Colored Petri nets [77, 121] extend regular Petri Nets by assigning a color to each token.

### 4.1.4 Dataflows

Dataflows come in many flavors, including Scenario Aware DataFlow (SADF, [134]) and Synchronous DataFlow For Free (SDF3, [133]). Dataflows allow the modeling of systems that operate in different modes (or scenarios). Dataflow models consist of actors that are connected via channels on which tokens flow with a certain rate. Dataflow models are popular because of their simplicity. However, they can be hard to analyze, especially when they are used to yield strict performance bounds.

## 4.2 Performance evaluation techniques

In the following, we describe how the performance models (as introduced Section 4.1) can be evaluated to obtain performance results, especially latencies. The focus will be on the automata of Section 4.1.1, because the other performance models can generally be mapped to them. Furthermore, the selection of an evaluation technique is closely related to the four dimensions (as introduced in Section 4.1.1). We discuss the four performance evaluation techniques (as presented in [152]).

### 4.2.1 Back-of-the-envelope

Back-of-the-envelope (BOTE, [152, Section 2.2.1]) analysis includes the transformation of a high-level performance model into a model that is simple enough to literally fit on the BOTE. Once this simple model has been obtained, the rules of mathematics are applied to it to quickly derive a solution. BOTE models can be very expressive, but there are no guidelines about obtaining such a model from a high-level performance model. In general, it might also be hard to express the system dynamics that, e.g., parallel processing and scheduling, bring about [16, 17].

Summarized, the BOTE technique yields basic estimates, which are easy to obtain but lacks accuracy. There is no general way of working.

### 4.2.2 Discrete-event simulation

Discrete-event simulation (DES, [15, 55][152, Section 2.2.2]) is an evaluation technique in which a model is traversed in a random way; probabilistic choices are resolved using a random number generator [102]. Hence, DES

can be applied to a DTMC. Addtionally, nondeterministic choices are often resolved randomly as well, whereas nondeterministic time can be solved using an as early as possibly (ASAP) policy. This makes DES widely applicable, e.g., to LTS, MDP, TA, PTA and STA automata.

DES only gives a first impression about complex models. To compensate for this, multiple runs can be performed to yield performance numbers that are statistically more reliable, e.g., using confidence intervals. Also, the length of the individual runs can be increased for more accurate results. DES is sometimes being referred as statistical model checking [97]. However, one might argue that statistical model checking also includes algorithms in which multiple runs of discrete-event simulation are systematically performed, i.e., the output of one run is used to configure the next run.

In short, DES yields average behaviour and is relatively quickly to perform since only one of many possible paths is explored. It can be applied to many kinds of automata, although how to deal with nondeterministic choice and time is not straightforward. One has to perform multiple runs and apply statistics carefully for reliable results. DES is unable to detect absolute values, which are in many contexts the values of concern.

### 4.2.3 TA model checking

TA model checking [13, 32][152, Section 2.2.3], which is also known as traditional model checking, is an evaluation technique in which a TA model is traversed exhaustively. Consequently, it is prone to the so-called state explosion [30, 31] which means that the model can easily become too complex to apply this technique, e.g., due to time and memory constraints.

In line with what a TA automaton offers, TA model is tailored to deal with nondeterminism but unable to address probabilities. However, TA model checking can still be applied to a PTA and an STA by omitting the probabilities.

TA model checking is used to detect absolute behaviour. That is, the absolute minimum and maximum latency are returned when used in case of latencies. However, since probabilities are omitted in case of a PTA and STA, it is impossible to know how likely the bounds occur. For instance, the absolute behavior might be extremely rare, i.e., virtually impossible to accomplish in practice, and thus possibly not relevant.

Shortly, TA model checking yields absolute bounds opposed to DSE, but does not provide probabilities at which they occur when applied to a PTA or an STA. Opposed to back-of-the-envelope (of Section 4.2.1) computations and DES (of Section 4.2.2), TA model checking is prone to the state space explosion [33, 113].

### 4.2.4 PTA model checking

PTA model checking [40, 68, 117][152, Section 2.2.4] extends TA model checking with probabilities. This makes it possible to see the probabilities with which certain states are reached, but does make this technique inherently more complex. PTA model checking leads to two outcomes per property, viz., namely a minimum and maximum probability. The difference between these probabilities is the result of how nondeterminism is resolved. Hence, the larger the impact of nondeterminism is, the more these values differ. Due to the addition of probability compared to a TA model, PTA model checking is even more prone to the state space explosion [33, 113] than TA model checking.

Applied iteratively, PTA model checking has proven to be able to retrieve latency distributions [145, 146]. Moreover, a more efficient approach has been proposed [145] which comprises a pipeline of the aforementioned techniques, viz., applying back-of-the-envelop (of Section 4.2.1), DES (of Section 4.2.2), TA model checking (of Section 4.2.3), and PTA model checking (of Section 4.2.4) in sequence. The underlying idea is that lightweight computations that are performed in the beginning reduce the number of expensive computations that are needed in the end.

## 4.3 Performance approaches

In this section, a number of performance approaches are considered. Such an approach applies one or more different evaluation techniques (of Section 4.2) to one or more different performance models (of Section 4.1) in order to yield performance results. Besides this, a performance approach may provide convenience to a user, including an IDE to enter the language with syntax checking and input validation, automatic evaluation, and visualizations of the results. We consider seven widespread approaches, namely Modest, STORM, PIPE2, POOSL, UPPAAL, PRISM and iDSL, as follows.

### 4.3.1 Modest

The Modest Toolset [64, 68, 70, 103] supports the modelling and analysis of hybrid, real-time, distributed and stochastic systems. It provides a modular framework centered around the SHA formalism [64, 75]. Modest uses a network of SHAs, and therefore can be applied to all automata of Section 4.1.1.

For analysis, Modest is equipped with the following three tools: (i) MCSTA, a disk-based explicit-state model checker for STA, PTA and MDP; (ii) MODES, is a statistical model checker for SHA, STA, PTA and MDP; and, (iii), prohver, a safety model checker for SHA.

### 4.3.2 STORM

Storm [39, 40] is a tool for the analysis of systems involving random or probabilistic phenomena. Storm is built around discrete- and continuous-time Markov models, i.e., DTMCs, MDPs, CTMCs and MA. Consequently, Storm supports most of the aforementioned automata (of Section 4.1.1). Storm focuses on efficiency and modularity, since model checking is both a data- and compute-intense task.

### 4.3.3 PIPE2

Platform Independent Petri net Editor 2 (PIPE2, [6, 41, 45, 46]) is an open source, platform independent tool for creating and analyzing Petri nets. This includes Generalised Stochastic Petri nets. PIP2E uses the Petri-Net Markup Language (PNML, [115]) as its core language, which can conveniently be modelled using an extensive GUI and transformed into Generalized Stochastic Petri-Nets (of Section 4.1.3).

On top of DES, PIPE2 has various analysis tools, including: (i) a distributed semi-Markov response time analyzer named SMARTA [44]; and, (ii) a distributed Markovian passage time analyzer HYpergraph-based Distributed Response-time Analyser (HYDRA, [23, 44]). HYDRA can explore large state spaces using a parallel algorithm that is executed on a network of commodity PCs.

### 4.3.4 POOSL

The Parallel Object-Oriented Specification Language (POOSL, [48, 52, 135]) is a very expressive language which can be used to model concurrent hardware/software systems. It has a small set of powerful primitives and it has an unambiguous formal semantics. Using the Eclipse-based PooslIDE tool, a user can edit and debug POOSL models in the Eclipse environment.

POOSL models can be simulated with the Rotalumis [48] simulator at high-speed. For this purpose, the POOSL language is implicitly transformed to an LTS, following the probabilistic-nondeterministic Segala model [157]. POOSL does not provide any model checking support.

### 4.3.5 UPPAAL

UPPAAL [19, 20, 96, 139], which is jointly devloped by the Uppsala and Aalborg university, has orginally been used to analyze TA models. Recently, UPPAAL has been extended with a Statistical Model-Checker (SMC, [26, 36]), which: (i) enables the user to specify probability distributions that drive the timed behaviors; and, (ii) has an engine that uses statistical model checking to compute an estimate of a probability, compare a probability with a value, compare two probabilities without computing them.

The UPPAAL SMC is able to handle timed systems, which is in contrast to classical SMC model checkers [86, 128, 169]. The UPPAAL SMC also visualizes the results, e.g., probability distributions, evolution of the number of runs with timed bounds, and computation of expected values.

### 4.3.6 PRISM

PRISM [94, 117] is a probabilistic model checker, i.e., a tool for formal modelling and analysis of systems that exhibit random or probabilistic behaviour. PRISM models are state-based, simple, low-level, and modular. A PRISM model can represent various transition systems depending on which constructs are used, e.g., DTMC, CTMC, MDP and PTA automata (of Section 4.1.1).

PRISM models can be simulated using a DSE engine, but also be automatically evaluated for a wide array of quantitative properties, e.g., the property "what is the probability of a failure causing the system to shut down within 4 hours?" can be represented in PCTL temporal logics [66].

### 4.3.7 iDSL

iDSL [141, 150, 151] is a DSL and toolset for the performance evaluation of service-oriented systems. iDSL generally uses Modest models under the hood as its performance model. Via iterative algorithms, iDSL is capable to perform DES, TA model checking, PTA model checking and iterative PTA model checking to obtain results for many similar designs. Moreover, iDSL combines the aforementioned techniques to efficiently obtain results [145].

The iDSL language is a high-level, concise language [150] that is tailored to the service-oriented systems domain; the language is limitedly applicable but very easy to understand by people familiar with this domain.

To generate comprehensible results, iDSL automatically visualizes its performance outcomes using Graphviz [51, 63] and GNUplot [62, 120], yielding, among others, latency CDF plots, trade-off graphs and latency bar charts.

## 4.4 Selection of performances model and approach

In Section 4.3, we have introduced seven performance approaches that we have considered to use in this thesis, viz., Modest, STORM, PIPE2, POOSL, UPPAAL, PRISM and iDSL.

In the remainder of this thesis, we will use iDSL as the performance evaluation approach for the following six reasons. First, as developers of iDSL we have exclusive access to its source code, which makes bug-fixing, adjusting and augmenting iDSL to the specific needs of this thesis relatively easy. In constrast, this means iDSL is not widely supported and that the findings of this thesis are hard to check and hard to reproduce.

Second, iDSL is a high-level language. As a result, the transformation from another high-level language such as BPMN to iDSL is relatively easy. To ensure this, we have positively verified first that BPMN adheres to the service-oriented system paradigm that iDSL advocates.

Third, iDSL supports a number of performance evaluation techniques. Namely, iDSL supports probabilistic model checking, traditional model checking, discrete-event simulation and a back-of-the-envelope method as means of performance evaluation. Consequently, we are able to derive various performance results for BPMN models of different complexities. iDSL even combines all of the previous techniques to yield an efficient approach [145].

Fourth, iDSL supports a number of performance models that are closely related to the supported performance evaluation techniques. Namely, iDSL supports four kinds of performance models (as discussed in Section 4.1): (i) an STA can be analysed using discrete-event simulations [150]; (ii) a PTA using probabilistic model checking [146, 148]; (iii) a TA using traditional model checking [150]; and, (iv) a back-of-the-envelope model is analysed in a specific way.

Fifth, iDSL automatically generates visualizations which are likely to provide deeper insight in the BPMN models than just generating performance numbers only. In general, the aforementioned approaches (of Section 4.3) do not automatically generate visualizations or at all. Particularly when visualizations have a domain-specific character, it is hard for generic approaches to generate this visualizations one is looking for.

Sixth and finally, the iDSL language makes it possible to specify many design instances that differ on a few aspects. Each of these design instances can then be automatically evaluated for performance. Hence, this makes it possible to compare different business processes, i.e., iDSL is capable of generating aggregated visualizations in which different design instances are compared. This allows the system modeller to select the best one of them.

# 5 The performance evaluation approach iDSL

In Section 4.4, iDSL has been selected as the performance evaluation approach. Here, we discuss iDSL in further detail. Section 5.1 discusses the conceptual model of iDSL. which is illustrated in Section 5.2 by providing a typical iDSL instance. Additionally, the underlying grammar of iDSL is provided in Table 55 of Appendix C. After this, the iDSL instance is transformed into Modest in Section 5.3, which finally enables the computation of performance results as Section 5.4 conveys.

## 5.1 The conceptual model of IDSL

This section describes the conceptual model of iDSL [152, Section 4.2]. That is, the model a service-oriented system that comprises six related concepts as depicted in Figure 15.

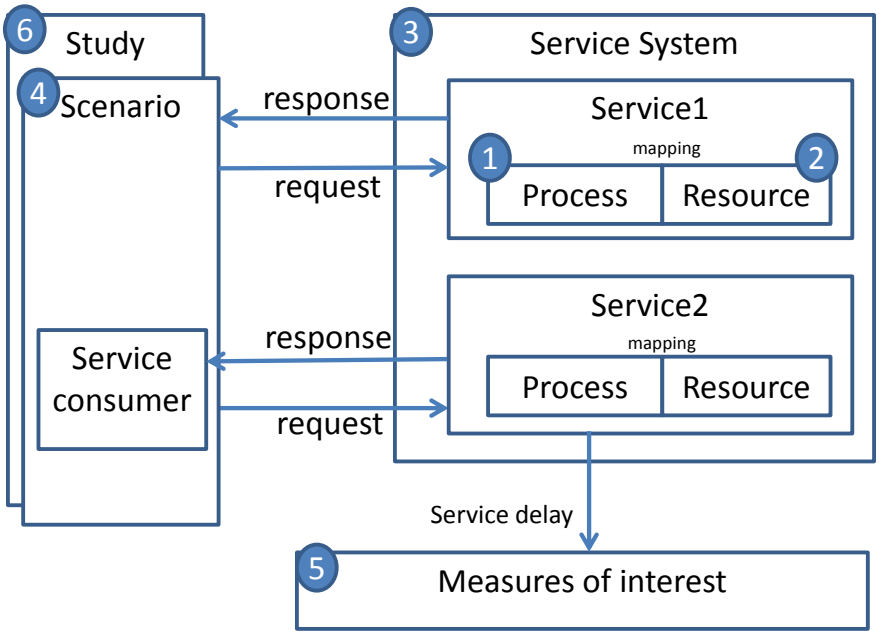



Figure 15: Conceptual model of a service system. External to the service system, measures of interest are obtained using scenarios.

A **service system**, as depicted in the upper right block, provides a service to one or more **service consumers** in its environment (exterior to the service system). For illustration, Figure 15 displays two services. Within these systems, a **service** is an autonomous, platform-independent entity that perform functions ranging from simple requests, e.g., a simple computation that can be performed on a calculator, to labour-intensive processes, e.g., performing logistics for a large amount of goods.

A **service consumer**, exterior to the system, sends a request for a specific service at a given time, after which the system responds with some delay. Besides providing the proper functionality, i.e., returning the right answers to requests, service systems often need to meet performance constraints, e.g., the system has to reply to a request within a certain time, *latency*. Service systems can be hard to analyze when they: (i) handle many service requests in parallel; (ii) for multiple kinds of services; and, (iii) in a real-time manner.

To meet the constraints, a **service** is implemented using one or more processes, one or more resources and a mapping, in accordance with the Y-chart philosophy [16, 17, 47, 87]. A **process** decomposes high-level service requests into atomic tasks, which are each assigned to resources through the **mapping** (from which we abstracted in the figure). Hence, the mapping forms the connection between a process and the resources it uses.

For simplicity, we assume that a **resource** is capable of performing one atomic task at time and at a certain rate. This is also known as the One-Time-One-Place-One-Person (OTOPOP, [122]) principle. When multiple services are invoked, resource usage may overlap, causing concurrency and making performance analysis more challenging.

A **scenario** consists of a number of systematically invoked service requests over time to observe the performance behaviour of the service system in specific circumstances. For instance, it can be valuable to know how a service system copes with many service requests for a relatively short amount of time. Service requests are assumed to be functionally independent of each other, i.e., service requests do not affect each other's functional outcomes, but may affect each other's performance implicitly due to concurrency.

A **study** evaluates a selection of systematically chosen scenarios to derive the system's underlying characteristics. Within a study, a **design space** efficiently describes many similar scenarios that only differ on a few aspects.

To conclude, a **measure of interest** defines which performance metric is interesting, given a system in a certain scenario. Measures are either external to the system, e.g., service throughputs, latencies and jitters, or internal to the system, e.g., resource queue sizes and utilizations.

## 5.2 An business-related IDSL instance

In this section, the performance approach iDSL is illustrated by providing an iDSL instance that is related to business. The iDSL instance is based on an existing image processing example of iDSL [152, Section 4.2]. To this end, the six concepts of iDSL (as revealed in Section 5.1) are implemented, as follows.

### 5.2.1 A Business iDSL Process

A process decomposes a service into a number of atomic tasks. In iDSL, this is implemented using a recursive data structure with layers of sub-processes and atomic tasks. At the lowest level of abstraction, the atomic tasks each specify a load. In a business context, this load can represent an amount of work, e.g., the required number of man-hours.

Table 13: The code of an iDSL process

```
1.  Section Process
2.  ProcessModel supply_chain_application
3.      seq supply_chain_seq {
4.          atom procurement load 50
5.          seq logistics {
6.              atom inbound_logistics load 44
7.              atom storage uniformLoad(80:140)
8.              atom outbound_logistics load 134
9.          }
10.         atom sales load 25
11.     }
```

Table 13 shows an example iDSL process, which combines hierarchies (curly brackets), sequential composition (*seq*) and atomic tasks (*atom*), as follows. At its highest level, it consists of a sequential task named " supply_chain_seq" (lines 3-11) that decomposes into the following three tasks. First, an atomic task "procurement" (line 4) with fixed load 50 represents acquiring goods from an external source. Second, a sequential task "logistics" (lines 5-9) is about moving goods. Third, atomic task "sales" (line 10) with fixed load 25 is concerned with passing of good to an external party for money.

At a lower level, the sequential task "logistics" consists of the following three atomic tasks. First, atomic task "inbound_logistics" (line 6) with load 44 encompasses transporting goods into a business. Second, atomic task "storage" (line 7) deals with putting the goods away in a warehouse. Its load is drawn, during model execution, from a uniform distribution on [80,140]. Third, 'outbound_logistics" (line 8) with load 134 consists of transporting goods out of a business.

In addition to what has just been conveyed, iDSL also provides process algebraic constructs for parallelism (*par*), nondeterministic choice (*alt*) and probabilistic choice (*palt*), as well as a mutual exclusion (*mutex*) to permit at most one process instance at a time to enter a certain process part. Furthermore, one can inject existing measurements into an iDSL model to make it more realistic [147, 151]. This is also known as model calibration.

### 5.2.2 A Business iDSL Resource

An iDSL resource is defined as recursive structure consisting of *decomp* and *atom* constructs, and a binary relation which defines how resources are connected. The *decomp* construct is used to create decomposable resources, whereas the *atom* construct is used to specify atomic resources that have a rate; the load they can process per time unit, e.g., the number of goods stored per hour. Resources need to be connected to perform operations in sequence for one process.

Table 14: The code of an iDSL resource

```
1.  Section Resource
2.  ResourceModel supply_chain_resource
3.      decomp supply_chain_decomp {
4.          atom logisticians rate 2
5.          atom marketing 5
6.      }
7.      connections { (logisticians,marketing) }
```

Table 14 shows an example iDSL resource that contains a decomposition "supply_chain_decomp" (lines 3-6). The decomposition comprises two connected atomic resources (line 7), viz., resource "logisticians" (line 4) with rate 2 and resource "marketing" (line 5) with rate 5.

### 5.2.3 A Business iDSL System

An iDSL system provides one or more services to its environment. Table 15) presents a system with one service. In line with the Y-chart philosophy [16, 17], the service "supply_chain_service" of Table 15 (lines 2-11) connects the already defined process of Table 13 and resource of Table 14, via a static mapping that assigns atomic tasks of the process to atomic resources.

Table 15 conveys that four atomic processes, viz., "procurement", "inbound_logistics", "storage" and "outbound_logistics", are performed by resource " logisticians" (lines 6-9). Contrarily, atomic resource "sales" is performed by resource "marketing" (line 10).

### 5.2.4 A Business iDSL Scenario

A scenario specifies a number of systematically-chosen service requests to a system. They are defined using one or more streams of requests. Table 16 conveys that two streams of requests both named "supply_chain_service" are defined (lines 3-6). Both streams have the same name, because they are both based on the same serivce "supply_chain_service" of the system (in Table 15). Each stream comprises an infinite number of requests and is defined using the starting time of the first and second request.

Table 15: The code of an iDSL system

```
1.  Section System
2.      Service supply_chain_service
3.          Process supply_chain_application
4.          Resource supply_chain_resource
5.          Mapping assign {
6.              (procurement,logisticians)
7.              (inbound_logistics,logisticians)
8.              (storage,logisticians)
9.              (outbound_logistics,logisticians)
10.             (sales,marketing)
11.         }
```

Table 16: The code of an iDSL scenario

```
1.  Section Scenario
2.      Scenario supply_chain_run
3.          ServiceRequest supply_chain_service
4.              at time 0, 400, ...
5.          ServiceRequest supply_chain_service
6.              at time dspace("offset"), dspace("offset")+400, ...
```

For the first stream (lines 3-4), the first and second requests are 0 and 400, respectively. Moreover, all inter-request times are assumed to be constant. We derive it is 400 for the first stream, viz., the elapsed time between the first and second request. Hence, the service requests take place at time 0, 400, 800, 1200, 1600, ...

For the second stream (lines 5-6), the request times are defined similarly. The inter-request times are again a constant 400. However, to influence the degree of concurrency between both streams, an initial offset is added to all the service requests of the second stream. This is accomplished by adding two *dspace* function calls, each with parameter "offset". The outcome of these function calls is constant within a design instance but may vary across design instances. This means that an iDSL model specifies multiple design instances, as will be explained further in the study (cf. Section 5.2.6).

### 5.2.5 A Business iDSL Measure

A measure defines what performance metric one would like to obtain, given a system in a certain scenario. Each measure includes specific techniques to obtain them. In Table 17, two measures are presented, as follows.

First, measure "ServiceResponseRimes" (lines 2-3) retrieves service response times, resource utilizations, and latency breakdowns in one go, via

Table 17: The code of an iDSL measure

```
1.  Section Measure
2.      Measure ServiceResponseTimes
3.          using 1 runs of 280 serviceRequests
4.      Measure ServiceResponseAbsoluteTimes
```

DES. It uses a given number of runs of a certain length (1 run of length 280 in the example).

Second, measure "ServiceResponseAbsoluteTimes" (line 4) yields absolute minimum and maximum response times, given a system and scenario. It is parameterless. Using this measure, the results are always obtained via traditional TA model checking. Hence, this measure is prone to the state-space explosion [33, 140] and might not always yield a result.

### 5.2.6 A Business iDSL Study

A study is a collection of systematically-selected scenarios to be analysed. In Table 18, one basis scenario is used that is parametrized using a so-called design space.

Table 18: The code of an iDSL study

```
1.  Section Study
2.      Scenario supply_chain_run DesignSpace
3.          ("offset" {"0" "20" "40" "80" "120" "160" "200" })
```

Previously it showed that the iDSL scenario (of Table 16, line 6) contains two *dspace* instances which both have parameter name "offset".

In Table 18, a study is defined which consist of a scenario (line 2) and a design space with one dimension "offset" (line 3). The two *dspace* instances refer to this dimension. We choose seven different values for the "offset" dimension, viz., 0, 20, 40, 80, 120, 160 and 200. This means that the study yields seven design instances.

## 5.3 Transforming the iDSL language into Modest

In Section 5.2, an iDSL instance has been specified by implementing the six concepts of Section 5.1. Executing the iDSL instance leads to a transformation (cf. [152, Section 4.3]) into the process algebra language Modest (of Section 4.3.1), which we describe concept by concept in this section.

**Process.** The iDSL process of Table 13 transforms into the Modest code of Table 19 after automatically executing the different steps of the iDSL

toolchain, as follows. First, hierarchies in iDSL are implemented using layered processes without parameters, e.g., as with process *supply_chain_seq()*.

Second, sequential composition in iDSL is converted to its equivalent in Modest, i.e., using semicolons.

Third, atomic tasks become references to single functions in Modest (to be defined in the system) that represent the mapping, with their respective loads as parameter, e.g., procurement(44) corresponds to load 44.

Fourth and finally, each process waits for a generator to be triggered through binary communications (to be explained in the scenario), indicated by a question mark, e.g., *generator_supply_chain_application?*.

Table 19: The generated Modest code from an iDSL *process*

```
process supply_chain_application_instance(){
  generator_supply_chain_application?;
  supply_chain_seq() }

process supply_chain_seq(){
  procurement(44);
  logistics();
  sales(25) }

process logistics(){
  inbound_logistics(44);
  storage(Uniform(80,140));
  outbound_logistics(134) }
```

**Resource.** The iDSL resource of Table 14 transforms into the Modest code of Table 20 after execution in iDSL, as follows. Two Modest processes are created for each iDSL resource. The first process (with prefix "machine_") comprises binary communications to handle concurrency and a delay that represents the resource being in use, i.e., processing a request. The self-recursion ensures that the resource stays alive forever. The length of a delay is a task time, i.e., the quotient of the load and rate. The second process (with prefix "machine_call_") abstracts communications (indicated with ? and !) from the process layer. Concurrency for resources is represented using nondeterministic choices, in a non-preemptive manner.

**System** The iDSL process of Table 15 transforms into the Modest code of Table 21 after execution in iDSL. Namely, iDSL creates Modest processes for each mapping. Each of these processes calls the mapped resource in Modest, i.e., sales is mapped to marketing and the others to logisticians.

Table 20: The generated Modest code from an iDSL *resource*

```
process machine_logisticians(){
  real taskload;
  machine_logisticians_start? {= taskload=sync_buffer =};
  delay (taskload/2 ) machinelogisticians_stop!;
    machine_logisticians() }

process machine_call_logisticians(real taskload){
  machine_logisticians_start! {= sync_buffer=taskload =};
  machine_logisticians_stop? }

process machine_marketing(){
  real taskload;
  machine_marketing_start? {= taskload=sync_buffer =};
  delay (taskload/5 ) machine_marketing_stop!;
    machine_marketing() }

process machine_call_marketing(real taskload){
  machine_marketing_start! {= sync_buffer=taskload =};
  machine_marketing_stop? }
```

Table 21: The generated Modest code from an iDSL *system*

```
process procurement(real taskload){
  machine_call_logisticians(taskload)}

process inbound_logistics(real taskload){
  machine_call_logisticians(taskload)}

process storage(real taskload){
  machine_call_logisticians(taskload)}

process outbound_logistics(real taskload){
  machine_call_logisticians(taskload)}

process sales(real taskload){
  machine_call_marketing(taskload)}
```

**Scenario.** The iDSL scenario of Table 16 transfers into several Modest instances, viz., one for each design instance. In Table 22, the situation for offset=200 (in bold) is displayed. The Modest code for other six designs is

Table 22: The generated Modest code from an iDSL *scenario*

```
process init_generator_supply_chain_service() {
  delay (0) generator_supply_chain_service() }

process generator_supply_chain_service() {
  clock c; tau = c=0 =;
  alt{
    :: generator_supply_chain_application!
    :: delay(1) tau // time-out };
  when urgent(c>=400)generator_supply_chain_service()}

process init_generator_supply_chain_service2() {
  delay (200) generator_supply_chain_service2() }

process generator_supply_chain_service2() {
  clock c; tau = c=0 =;
  alt{
    :: generator_supply_chain_application!
    :: delay(1) tau // time-out };
  when urgent(c >= ((400 + 200) - 200) )
      generator_supply_chain_service2() }
```

obtained by replacing the three boldfaced occurrences of 200 by the offset value of choice.

The Modest code comprises four Modest processes, viz., two processes for each stream of requests. The first process (with prefix "init_generator_") performs the initial delay once, e.g., 0 and 200, respectively, in the example.

The second process (with prefix "generator) then loops forever, with period of the inter-request time, e.g., 400 for both in the example, triggering the process once every loop. When the service system fails to respond to a request immediately, a time-out occurs that drops the request. Hence, no queueing takes place.

**Measure** The iDSL process of Table 17 contains two different measures which are individually transformed into distinct Modest code, as follows.

First, measure "ServiceResponseTimes" yields service response times for a given number of runs of a certain length (1 run of length 280 in the example). The measure also provides insight in resource utilizations and latency breakdowns, i.e., the latencies of subprocesses, in one go. The measure leads to the generation of a Modest model with STA as its underlying model and it is evaluated via DESs using MODES of the Modest toolset [64].

Table 23 shows that process "logistics" is augmented with a stopwatch

Table 23: The generated Modest code from an iDSL *measure*: latencies

```
process logistics() {
  tau {= stopwatch_logistics = 0, logistics_done = false =};
  ...
  tau {= logistics_done = true, counter_logistics++ =};
  tau {= logistics_done = false =}
}

property property_latency_logistics_1 =
  Xmax( stopwatch_logistics | stopwatch_logistics_done &&
        counter_logistics==1 );
property property_latency_logistics_2 =
  Xmax( stopwatch_logistics | stopwatch_logistics_done &&
        counter_logistics==2 );
...
property property_latency_logistics_280 =
  Xmax( stopwatch_logistics | stopwatch_logistics_done &&
        counter_logistics==280 );
```

Table 24: The generated Modest code from an iDSL *measure*: utilizations

```
property property_utilization_logistians =
  Xmax ( util_counter_logistians / 10000 | time == 10000 );
property property_utilization_marketing =
  Xmax ( util_counter_marketing / 10000 | time == 10000 );
process machine_logistians() { ...
  delay ( taskload/ 2 )
    tau {= util_counter_logisticians+=(taskload/2) =}; ...}
process machine_marketing() { ...
  delay ( taskload/ 5 )
    tau {= util_counter_marketing+=(taskload/5) =}; ...}
```

named "stopwatch_logistics" and a counter named "counter_logistics" by adding code before and after the original process. Using 280 properties, viz., one per request, the individual latencies are retrieved. Each (sub)process in the model is augmented a similar way.

Table 24 conveys that the processes for resource "logisticians" and "marketing" are extended with a counter (with prefix: "util_counter") that increments anytime activity takes place. In two corresponding properties this time is divided by the total elapsed, i.e., 10000 (boldfaced), to compute the utilization, i.e., the quotient of the time active and the total time. The value 10000 is a careful trade-off between simulation time and accuracy,

Table 25: The generated Modest code from an iDSL *study*

```
real sync_buffer;
closed par{
  :: do { supply_chain_application_instance() }
  :: do { supply_chain_application_instance() }
  :: init_generator_supply_chain_service()
  :: init_generator_supply_chain_service2()
  :: machine_logisticians()
  :: machine_marketing()
}
```

which likely needs to be adjusted in a different context.

Second, measure "ServiceResponse absolute times" leads to the absolute minimum and maximum response times, given a system and scenario. A Modest model is generated that adheres to TA as its underlying model by (among others): (i) turning real numbers into integers by rounding; and, (ii) turning probabilistic alternatives into nondeterminstic alternatives by omitting probabilities. MCTAU of the Modest toolset [64] is used iteratively to evaluate the model using a binary search algorithm [145, Section 3.5].

**Study** The iDSL study of Table 18 transforms into the Modest code of Table 25 after execution in iDSL. It comprises the parallel execution of: (i) two parallel processes that repeat forever; (ii) two scenarios that are service streams; (iii) a logisticians resource; and, (iv) a marketing resource.

## 5.4 The iDSL performance results

Section 5.2 conveyed an iDSL instance which transformed into Modest code in Section 5.3. Here, we convey the results for different performance evaluation techniques (see Section 4.2). They are obtained by executing Modest, followed by the automatic application of post-processing steps (using GraphViz [51] and GNUplot [120]) for visualizations (cf. [152, Section 4.5, 5.5, 6.5 and 7.6]).

**Back-of-the-envelope computations** are simple computations; they can literally be written down on the back of an envelop. The latency of the iDSL instance of Section 5.2 is determined by recognizing the following two cases.

In case of no concurrency, the service latency can be computed as the quotient of each individual load in the process (of Table 13) and the rate of

the resource (of Table 14) that it is mapped to (of Table 15). This yields

$$\frac{50 + 44 + min(80, 140) + 134}{2} + \frac{25}{5} = 159, \text{ and}$$

$$\frac{50 + 44 + max(80, 140) + 134}{2} + \frac{25}{5} = 189,$$

for the minimum and maximum time, respectively.

In case of concurrency, two services with potentially many service instances are being executed in the system. However, since the period between the instantiation of service instances of an individual service is 400 time units, and services maximally execute for 189 time units, only two service instances execute at any time. Hence, we multiply the results for the no concurrency case by two to obtain an overestimation of the upper bounds:

$$159 \cdot 2 = 318, \quad \text{and} \quad 189 \cdot 2 = 378,$$

for the minimum and maximum time, respectively

**Discrete-Event Simulations** in iDSL leads to service latencies, resource utilizations and latency breakdowns which are combined into a visualization. Figure 16 displays the result for the iDSL instance of Section 5.2. It reveals the process structure of the iDSL instance and the mapping to the resources. Regarding performance, process "Supply_chain_seq" averagely takes ca. 306 time units, and its subprocesses take ca. 37, 264 and 5 times units, respectively. Furthermore, it can be seen that the utilization of resource "CPU" is approximately 0.83, and the one of "GPU" merely 0.025.

**Traditional Model Checking** (see Section 5.3) leads to an absolute latency bounds, e.g., 159 and 189 for the iDSL instance of Section 5.2. For illustration, Table 26 shows the traces that has led to determining the lower bound of 159, which consists of eleven TA model checking iterations.

**PTA Model Checking** is the iterative application of model checking on a PTA model [145, 146] (cf. [152, Section 7.6]. It leads to a minimum and maximum latency distribution.

Table 26: The execution trace of Traditional Model Checking using a binary search

**LB: Compute lower bounds, execution trace**

```
LB(0,1024)  -> LB(0,512) -> LB(0,256) -> LB(128,256) ->
LB(128,192) -> LB(128,160) -> LB(144,160) ->
LB(152,160) -> LB(156,160) -> LB(158,160) ->
LB(159,160) -> 159
```

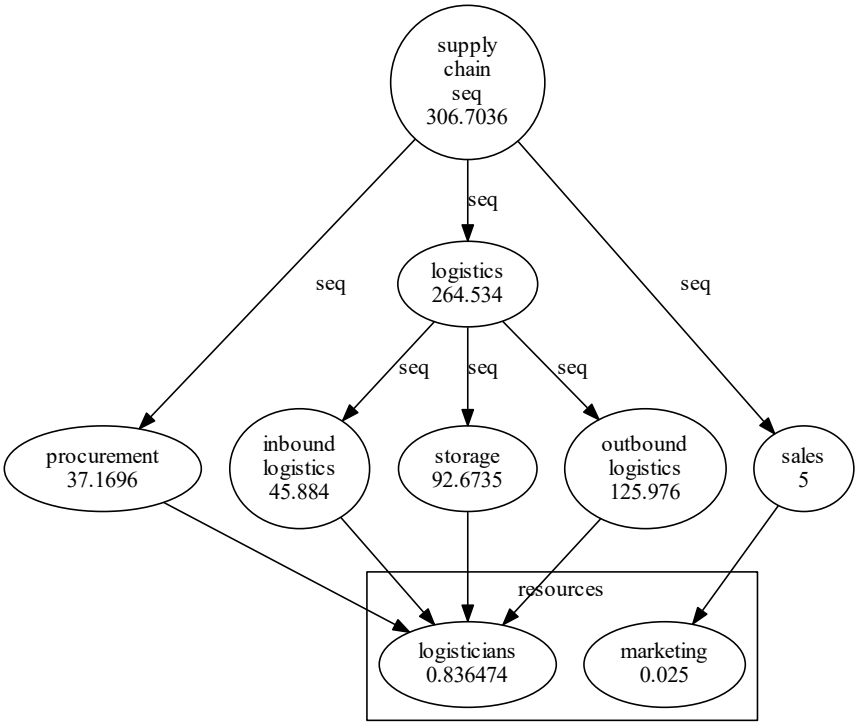



Figure 16: The latency breakdown chart and utilizations.

# 6 Extending BPMN for a transformation to iDSL

In Section 3, BPMN was introduced and formalized on the basis of six typical BPMN examples from literature [163]. After that, the performance evaluation approach iDSL was selected from seven approaches in Section 4 and more carefully discussed in Section 5.

The previous steps enables us to evaluate the performance of a given BPMN instance via a transformation from BPMN into iDSL. However, it will turn out here that a BPMN instance does not contain all the properties that are needed to specify a full-fledged iDSL model. Namely, a regular BPMN contains an iDSL process, but does not cover the remaining resource, system, scenario, measure and study (cf. Section 5.1).

Therefore, we extend the regular BPMN (cf. Table 37 of Appendix A.1) with the just mentioned iDSL concepts. The extension is brought about by investigating what each of the typical BPMN examples (of Section 3.1 till 3.6) need. At the same time, we extend the six examples with concepts that are not shown on the BPMN diagrams, e.g., taskloads and taskrates, to make them comply to the extended grammar. In total, this leads to so-called extended BPMN (cf. Table 38 of Appendix A.2).

## 6.1 A simple process

In Figure 3, the BPMN diagram of the "A simple process" example is displayed [163, Figure 1], which is formalized in Table 2 of Section 3.1. In Table 27, this formalization (lines 3-6) is augmented with extra information to enable a transformation to iDSL. In line with iDSL, the formalization of an extended BPMN instance contains six sections, as follows.

- Section "business process" (lines 3-6) contains the BPMN instance of Table 2. It is extended with task loads for each task (between brackets) which express the amount of work the respective task requires. That is, IdentifyPaymentMethod has task load 0, PaymentMethod 5, AcceptCashOrCheck 30, PreparePackageForCustomer 15, and ProcessCreditCard 10. Morever, probabilities have been added for the two alternatives, i.e., AcceptCashOrCheck gets executed with probability $\frac{1}{4}$ and ProcessCreditCard with $\frac{3}{4}$, represented by the values 1 and 3 before the different value chains, respectively.

- Section "resource" (lines 8-9) contains resources that are capable of performing the tasks that are defined in the previous section. The "A simple process" example comprises a compound resource "cashdesk", which consist of a single resource "Cashier". Atomic resource "Cashier" has task rate 1 (as indicated between brackets), which means it can process one load per time unit.

Table 27: The extended BPMN model instance "A simple process"

```
1.  Extended BPMN model
2.
3.  Section business process
4.      IdentifyPaymentMethod(0); PaymentMethod(5) + (
5.      1 AcceptCashOrCheck(30); PreparePackageForCustomer(15);.
6.      3 ProcessCreditCard(10); PreparePackageForCustomer(15);.)
7.
8.  Section resource
9.      cashdesk || (cashier(1);)
10.
11. Section mapping
12.     default→cashier
13.
14. Section scenario
15.     TriggerAt 0, dspace(interarrival), ... using 10 instances
16.
17. Section measure
18.     simulation 1 runs 1000 requests
19.
20. Section design space
21.     interarrival (37 38 39 40 41)
```

- Section "mapping" (lines 11-12) assigns tasks of the business process to respective resources to perform them. Since there is only one resource to perform tasks, all tasks are automatically mapped to the resource "cashier". This is indicated by the default operator.
- Section "scenario" (lines 14-15) indicates when a business process is executed for the first time and then periodically forever. This is defined using an initial and second request time. For the current example these values are 0 and dspace(inter-arrival). That is, the business process gets executed at time 0 and then periodically depending on a value for "inter-arrival" in the design space (to be explained later).
- Section "measure" (lines 17-18) defines which the performance metrics are of interest and how they are brought about. For this example, one simulation run of 1000 requests is used to obtain several measures of interest, such as utilization and latency.
- Section "design space" (line 20-21) provides a convenient way of defining many designs. For this purpose, one or more of dimensions are defined that each comprise one or more values. The $n^{ary}$ Cartasian product of these $n$ dimensions then yields the set of design instances.

Analogously, a design instance assigns a value to each of the design space dimensions. For the current example, a dimension "interarrival" with values 37, 38, 39, 40, and 41 has been defined. The usage of *dspace* operator in the scenario (line 15) leads to five different design instances that only differ in there inter-arrival times. Note that the dimension "interarrival" and its five values are unquoted. The quotes that iDSL calls for as Table 18 illustrates, are added by the from BPMN to iDSL generator for the study (cf. Table 54).

Table 28: The extended BPMN model instance "A segment of a process with more details"

```
1.  Extended BPMN model
2.
3.  Section business process
4.      AnySuppliers(1) + (
5.      SendNoSuppliers(0);.
6.      ((1,3)(1,2)(1,1)) times (SendRFQ(4); ReceiveQuote(dspace(receptiontime)); AddQuote(5);.) >< (FindOptimalQuote(2);.) )
7.
8.  Section resource
9.      server || (servernode(1);)
10.
11. Section mapping
12.     default→servernode
13.
14. Section scenario
15.     TriggerAt 0, 80, ... using 10 instances
16.
17. Section measure
18.     simulation 1 runs 1000 requests
19.
20. Section design space
21.     receptiontime (10 20 30 40 50 60)
```

## 6.2 A segment of a process with more details

Figure 5 conveys the BPMN diagram of the "A segment of a process with more details" example is displayed [163, Figure 2]. It is formalized in Table 4 of Section 3.2 while Table 28 augments this formalization (lines 3-7) to enable a transformation to iDSL, as follows.

- Section "business process" (lines 3-7) conveys that activity "AnySuppliers" has load 1, "SendNoSuppliers" no load (indicated by a 0), "SendRFQ" load 4, ReceiveQuote a load that depends on the design space dimension "receptiontime" (explained later), "AddQuote" load 5 and FindOptimalQuote load 2. It can be seen that the process contains a nondeterministic choice. This is in contrast with the previous BPMN model (cf. Table 27 of Section 6.1 in which probabilities were introduced.

- Section "resource" (lines 8-9) contains the only resource "servernode", which has a rate of 1.

- Section "mapping" (lines 11-12) maps all activities (lines 4-6) to the only resource "servernode", as indicated by the default keyword.

- The "scenario" section (lines 14-15) indicates that the scenario triggers at time 0 and then periodically every 80 time units, forever.

- The "measure" section (lines 17-18) conveys that one simulation run of 1000 requests is used to potentially obtain several measures of interest.

- Finally, the "design space" section (lines 20-21) comprises one dimension named reception time. This dimension is used to vary the load of activity "RececeiveQuote" from 10 till 60, which implicitly affects execution time of the business process.

### 6.3 A process with pools

The BPMN diagram of the "A process with pools" example [163, Figure 3] is displayed in Figure 7 and formalized in Table 5) (of Section 3.3). In Table 29, this formalization (lines 3-15) is augmented with extra information to enable a transformation to iDSL, as follows.

- In the "business process" section (lines 3-15), the loads of all activities but two are constant, viz., activity "receiveDoctorRequest" has load 30, "sendAppointment" load 40, "receiveAppointment" load 70, "sendSymptoms" load 50, "receiveSymptoms" load 100, "sendPrescriptionPickup" load 80, "receivePrescriptionPickup" has load 40, "sendMedicineRequest" load 20, "receiveMedicineRequest" load 30 and "sendMedicine" load 70. In constrast, acitvity "sendDoctorRequest" has a load that is uniformly selected on range [50 : 70], and receiveMedicine one on [20 : 40].

- The "resource" section (lines 17-19) displays resource rates that depend on different design space dimensions. That is, the rate of resource "patient" relies on dimension "patientrate" whereas the rate of resource "doctor" depends on dimension "doctorrate".

Table 29: The extended BPMN model instance “A process with pools”

```
1.  Extended BPMN model
2.
3.  Section business process
4.      sendDoctorRequest(uniform 50 70);
5.      receiveDoctorRequest(30);
6.      sendAppointment(40);
7.      receiveAppointment(70);
8.      sendSymptoms(50);
9.      receiveSymptoms(100);
10.     sendPrescriptionPickup(80);
11.     receivePrescriptionPickup(40);
12.     sendMedicineRequest(20);
13.     receiveMedicineRequest(30);
14.     sendMedicine(70);
15.     receiveMedicine(uniform 20 40);.
16.
17. Section resource
18.     patientDoctorServ ||
19.  (patient(dspace(patientrate));doctor(dspace(doctorrate));)
20.
21. Section mapping
22.     receiveDoctorRequest→doctor
23.     sendAppointment→doctor
24.     receiveSymptoms→doctor
25.     sendPrescriptionPickup→doctor
26.     receiveMedicineRequest→doctor
27.     sendMedicine→doctor
28.     default→patient
29.
30. Section scenario
31.     TriggerAt 0, 100, ... using 10 instances
32.
33. Section measure
34.     simulation 1 runs 1000 requests
35.
36. Section design space
37.     patientrate (6 7 8 9) doctorrate (6 7 8 9)
```

- The "mapping" section (lines 21-28) conveys that resource "doctor" is assigned to perform six activities (lines 22-27), viz., "receiveDoctorRequest", "sendAppointment", "receiveSymptoms", "sendPrescriptionPickup", "receive-MedicineRequest", and "sendMedicine". Besides that, the remaining six activities are performed by resource "patient" (line 28).

- It shows in section "Scenario" (lines 30-31) that the scenario triggers at time 0 and then periodically every 100 time units, forever.

- The "measure" section (lines 33-34) contains one simulation run of 1000 requests, which is used to obtain several measures of interest.

- Finally, Section "design space" (line 36-37) contains two dimensions, viz., "patientrate" and "doctorrate". The dimensions have four values each and therefore yield a design space of 16 instances, viz., $4 \times 4$ or the Cartesian product of the dimensions. The dimensions are used to individually vary the rate of resource "patient" and "doctor", respectively. Therefore, this design space enables the exploration of different scenarios in which the rates of the "patient" and "doctor" vary independently. The design space is expressed in a concise way.

### 6.4 A segment of a process with lanes

Figure 9 shows the BPMN diagram of the "A segment of a process with lanes" example [163, Figure 4]. Table 7) (of Section 3.4) formalizes it and in Table 30 this formalization is augmented with extra information to enable a transformation to iDSL.

- The "business process" section (lines 3-10) comprises an activity that is followed by two parallel activities. Namely, activity "DispatchToApprover" has a uniform load on range $[10 : 30]$. It is followed by two main activities, as follows. First, activity "PreparePO" has load 40 and decomposes into three sub-activities; "PreparePOHeader" with load 10, "PreparePoBody with load 20, and "PreparePOFooter" with load 10. Note that this is modelled by giving "PreparePO" load 0, after which the sub-activities execute in sequence with their corresponding loads. Second, activity "ApproveRequest" comprises a nondeterministic binary choice between a load of 10 plus the value of design space dimension "adminoverhead", and a load of 30 plus the value of design space dimension "adminoverhead".

- The "resource" section (lines 12-13) contains three resources named "webServer", "management", and "administration", respectively. They all have a rate of 1.

Table 30: The extended BPMN model instance "A segment of a process with lanes"

```
 1.  Extended BPMN model
 2.
 3.  Section business process
 4.      DispatchToApprover (uniform 10 30) || (
 5.          PreparePO(0) || ( PreparePOHeader(10);
 6.              PreparePOBody(20); PreparePOFooter(10);. )
 7.          ApproveRequest(0) + (
 8.              ApproveRequest(10+dspace(adminoverhead));.
 9.              ApproveRequest(30+dspace(adminoverhead));. )
10.      )
11.
12.  Section resource
13.      PO_resource || ( webServer(1); management(1); administra-
     tion(1); )
14.
15.  Section mapping
16.      DispatchToApprover→webServer
17.      default→management
18.      ApproveRequest→administration
19.
20.  Section scenario
21.      TriggerAt 0, 1000, ... using 1 instances
22.
23.  Section measure
24.      model checking
25.
26.  Section design space
27.      adminoverhead (0 20 40)
```

- The "mapping" section (lines 15-18) states that activity "DispatchToApprover" is performed by resource "webServer", and activity "ApproveRequest" by "administration". The remaining three activities "PreparePOHeader", "PreparePoBody, and "PreparePOFooter" are mapped to resource "management" as indicated by the default keyword.

- The section "Scenario" (lines 20-21) contains a scenario that triggers at time 0 and then periodically every 1000 time units, forever.

- The "measure" section (lines 23-24) provides model checking as a means to retrieve latency distributions.

- Section “design space” (lines 26-27) comprises one dimension “adminoverhead” which is used to vary the load of activity “ApproveRequest” in the process between 0, 20 and 40.

### 6.5 A high-level business process

Figure 11 conveys the BPMN diagram of the “A high-level business process” example [163, Figure 5]. It is formalized in Table 9) (of Section 3.5). Additionally, this formalization (lines 3-10) is augmented with extra information to enable a transformation to iDSL in Table 31, as follows.

- The “business process” section (lines 3-10) encompasses a sequential process of three activities. These activities have uniformly distributed loads, viz., activity “pages2and3” has a uniform load on segment $[1:7]$, activity “pages4and5and6and7” on segment $[1:10]$, and “pages8and9and10” on segment $[1:15]$. However, the process can aborted after every activity. For this purpose, the process contains two of the newly introduced “abort” activities with a load of 0 (line 5 and 7). To introduce variation, the abort after the first activity occurs in a probabilistic way, viz., based on a coin-flip (a probability of $\frac{1}{2}$ each), and the activity after the second abort in a nondeterministic way.

- The “resource” section (lines 12-13) consist one resource named “processor”, which has a rate of 1.

- In the “mapping” section (lines 15-16), all activities are mapped to the only resource ”processor” (line 16).

- Section “scenario” (lines 18-19) contains a scenario that triggers at time 0 and then periodically every 35 time units, forever.

- The “measure” section (lines 21-22) provides model checking as means to retrieve latency distributions, which is parameterless.

- Section “design space” (lines 24) presents a design space that has no dimensions. This means there is exactly one design instance. Consequently, the *dspace* operator has not been and cannot be used in the five aforementioned sections.

Table 31: The extended BPMN model instance “A high-level business process”

```
1.  Extended BPMN model
2.
3.  Section business process
4.      pages2and3(uniform 1 7) + (
5.          1 abort(0);.
6.          1 pages4and5and6and7(uniform 1 10) + (
7.              abort(0);.
8.              pages8and9and10(uniform 1 15);.
9.          )
10. )
11.
12. Section resource
13.     res || ( processor(1); )
14.
15. Section mapping
16.     default→processor
17.
18. Section scenario
19.     TriggerAt 0, 35, ... using 1 instances
20.
21. Section measure
22.     model checking
23.
24. Section design space
```

## 6.6 A hierarchical business process

Figure 13 shows the BPMN diagram of the "A hierarchical business process" example [163, Figure 6]. The folded version is formalized in Table 10 (of Section 3.6) and the unfolded version can be found in Table 11 (of Section 3.6). Additionally, the folded version is augmented with extra information to enable a transformation to iDSL in Table 32 and the unfolded version in Table 33. They only differ in the process and mapping, and are therefore discussed simultaneously, as follows.

- The "business process" section (lines 3-4) contains activity "Procurement" with load 50, followed by one repetition of activity "Logistics" with a uniform load on segment [258 : 318], and finally an activity "Sales" with load 25. In case activity "Logistics" is unfolded, it is being replaced by three sub-activities, viz., "Inbound_logistics" with load 44, activity "Storage" with a uniform load on segment [80 : 140] and activity "bound_logistics" with load 134.

- The "resource" section (lines 6-7) provides two resources, viz., "Logistians" with rate 2 and "Marketing" with rate 5.

- The "mapping" section (lines 9-11) contains a mapping of activity "Sales" to resource "Marketing". All other activities are mapped to "Logistians" regardless of the case. That is, activity "Logistics" is mapped to resource "Logistians" in the folded case. On the other hand, Activities "Inbound_logistics", "Storage" and "bound_logistics" are mapped to resource "Logistians" in the unfolded case.

- Section "scenario" (lines 13-15) provides two concurrent scenarios that are used to test the effect of concurrency. One scenario triggers at time 0 and then periodically every 400 times units, forever. The other scenario triggers periodically every 400 times unit too and forever, but the initial trigger depends on the "offset" dimension of the design space; it varies between 0 (maximum concurrency), 20, 40, 80, 120, 160 and 200 (no concurrency).

- The "measure" section (lines 17-19) shows that two measures are used. Namely, both model checking for latency distributions and one discrete-event simulation run of 280 requests to obtain various measures.

- Section "design space" (lines 21-22) conveys a the design space with one dimension "offset". As mentioned before, it is used to vary the degree of concurrency in the scenario.

Table 32: The extended BPMN model instance “A hierarchical business process” (folded)

```
1.  Extended BPMN model
2.
3.  Section business process
4.      Procurement(50); ((1,1)) times (Logistics(uniform 258 318);.)
    >< (Sales(25);.)
5.
6.  Section resource
7.      resources || (Logistians(2); Marketing(5);)
8.
9.  Section mapping
10.     default→Logistians
11.     Sales→Marketing
12.
13. Section scenario
14.     TriggerAt 0, 400, ... using 10 instances
15.     TriggerAt dspace(offset), (400+dspace(offset)), ... using 10 in-
    stances
16.
17. Section measure
18.     simulation 1 runs 280 requests
19.     model checking
20.
21. Section design space
22.     offset (0 20 40 80 120 160 200)
```

Table 33: The extended BPMN model instance “A hierarchical business process” (unfolded)

| | |
|---|---|
| 1. | Extended BPMN model |
| 2. | |
| 3. | Section business process |
| 4. | Procurement(50); ((1,1)) times (Inbound_logistics(44);Storage(uniform 80 140);Outbound_logistics(134);.) >< (Sales(25);.) |
| 5. | |
| 6. | Section resource |
| 7. | resources \|\| (Logistians(2);Marketing(5);) |
| 8. | |
| 9. | Section mapping |
| 10. | default→Logistians |
| 11. | Sales→Marketing |
| 12. | |
| 13. | Section scenario |
| 14. | TriggerAt 0, 400, ... using 10 instances |
| 15. | TriggerAt dspace(offset), (400+dspace(offset)), ... using 10 instances |
| 16. | |
| 17. | Section measure |
| 18. | simulation 1 runs 280 requests |
| 19. | model checking |
| 20. | |
| 21. | Section design space |
| 22. | offset (0 20 40 80 120 160 200) |

# 7 Performance evaluation results

In Section 7.1 till 7.5, we present and discuss the visual performance results that are generated by iDSL for five of the six typical business process (cf. Section 3.1 till 3.5) for validation, respectively. We have omitted the sixth use case (of Section 3.6) here, because it does not introduce new language constructs, viz., its structure is similar to the third business process (of Section 3.3) which encompasses a sequence of activities.

The visual performance results are evaluated by performing the following five steps for each BPMN process (as also graphically depicted in Figure 1):

1. The typical, informal BPMN process (cf. Figure 3 till 11 in Section 3, respectively) is formalized manually, which leads to a formal BPMN process (cf. Table 2 till 9 in Section 3, respectively).
2. The formal BPMN process of the previous step is extended to contain all the all the information iDSL requires, leading to a extended BPMN process as Table 27 till 31 in Section 6 demonstrate.
3. The extended BPMN process of the previous step is converted into an iDSL instance (cf. Table 56 till 60 in Appendix C.3, respectively) using the iDSL generator (cf. Appendix C.1 and explained briefly below).
4. The iDSL toolset is executed with an iDSL instance of the previous step as input, leading to a visualization as displayed in Figure 17 till 21 of Appendix 7, respectively.
5. Applying the visualization of the previous step to the informal BPMN process of step 1 and draw conclusions regarding performance from it.

Before looking at the results, we briefly explain how the iDSL generator of step 3 functions (cf. Appendix C.1 for an extensive explanation). The iDSL generator transforms an extended BPMN process into an iDSL instance by individually transforming each of the six following sections of an extended BPMN process into a corresponding section of an iDSL instance.

1. **Process:** Shorthand notations that are used in a business process, like ||, +, ;, are replaced by iDSL process algebra counterparts, i.e., *par*, *alt* and *seq*, respectively. Moreover, a probabilistic value chain in a business process requires a slightly more complex transformation to become a *palt* construct in iDSL.
2. **Resource:** The hierarchical structure of resources in a business process, using the || operator, is replaced by a structure using "atom" and "decomp" keywords in iDSL.
3. **System:** A list of resources that perform activities in BPMN is transformed into a list of processes that are mapped to resources in iDSL. One fixed high-level Process, Resource and Service each are added in iDSL.

4. **Scenario:** A triggerAt statement in business process transforms into a Scenario with a ServiceRequests that relate to the triggerAt in iDSL.
5. **Measure:** There are two measures in a BPMN model, viz., "model checking" and "simulation x runs y requests", which have an equivalent in iDSL albeit with longer names.
6. **Study:** A business process study comprises a design space, which has an similar equivalent in iDSL.

| **Prob.** | **0.0** | **0.1** | **0.2** | **0.3** | **0.4** | **0.5** | **0.6** | **0.7** | **0.8** | **0.9** | **1.0** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **ia=37** | 30 | 30 | 30 | 33 | 40 | 44 | 50 | 57 | 67 | 91 | 260 |
| **ia=38** | 30 | 30 | 30 | 30 | 34 | 42 | 46 | 50 | 59 | 74 | 230 |
| **ia=39** | 30 | 30 | 30 | 30 | 32 | 38 | 41 | 50 | 51 | 63 | 199 |
| **ia=40** | 30 | 30 | 30 | 30 | 30 | 30 | 40 | 40 | 50 | 60 | 160 |
| **ia=41** | 30 | 30 | 30 | 30 | 30 | 30 | 39 | 46 | 50 | 56 | 114 |

Table 34: A textual-representation of the cumulative latency distributions of "A simple process" for different designs.

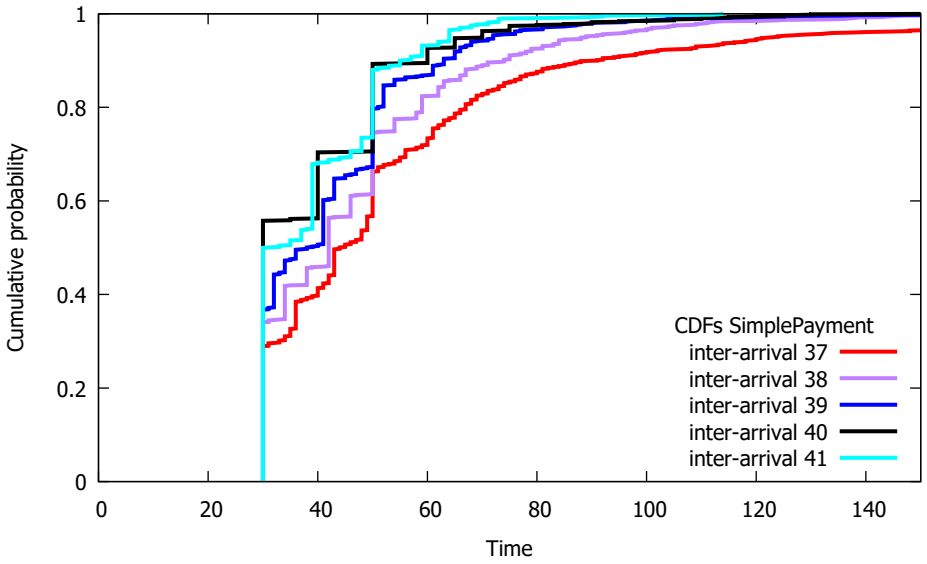



Figure 17: Cumulative latency distributions of "A simple process" for different designs.

## 7.1 A simple process

In this section, we apply the performance visualization of "A simple process" (cf. Figure 17 which is the result of executing iDSL specification (cf.

Table 56), to its belonging informal BPMN process (cf. Figure 3). For increased readability, Table 34 provides a textual representation of Figure 17. The results are created using discrete-event simulations (cf. Section 4.2.2).

Figure 17 is a so-called cumulative probability graph (CDF), which conveys a relation between the cumulative probability (on the y-axis) and the latency (on the x-axis). In this thesis, the latency is the time it takes for a whole BPMN process instance to finish. Moreover, Figure 17 shows this probability for multiple designs using one plot each, i.e., an aggregated CDF. The designs range on the design dimension "inter-arrival" from 37 till 41, which correspond to the time between different requests. For illustration, we emphasise that Table 34 conveys that for design with inter-arrival time 37, the probability it has finished after 44 times units is 0.5, after 50 time units is 0.6, after 57 time unit is 0.7, etc.

It can be seen in Figure 3 that the whole BPMN process comprises a sequence of three activities, viz., activity "PaymentMethod", either "AcceptCashOrCheck" or "ProcessCreditCard", and "PreparePackageForCustomer". The choice between "AcceptCashOrCheck" or "ProcessCreditCard" and the possibility for concurrency between different instances, make performance analysis complex. All activities are performed by one resource "Cashier" with load 1. Based on this information, we derive the following performance properties from Figure 17 and Table 34:

1. Figure 17 and Table 34 reveal that no latency value is below 30 regardless of the design. This is called best case latency and occurs when the activity "ProcessCreditCard" is invoked (with probability $\frac{3}{4}$) opposed to "AcceptCashOrCheck" (with probabiltiy $\frac{1}{4}$). In that case, the sum of the activities "PaymentMethod", "ProcessCreditCard" and "PreparePackageForCustomer" is $5 + 10 + 15 = 30$ loads of work. Given that default resource "Cashier" has a load of 1, the activities take 30 time units to be executed. Moreover, the latency of 30 is smaller than all the inter-arrival times, which are at least 37. Hence, the probability that a process ends in 30 time units is greater than 0 for each design.
2. Figure 17 and Table 34 convey that whenever the interarrival dimension of the design space is increased, the latency goes down for each probability. Concretely, moving down in Table 34 yields a latency that is decreasing. This is because when the inter-arrival times increase, the degree of currency decreases, the resource utilization decreases, and ultimately the latency decreases.
3. Table 34 shows that each design has a finite latency value for probability 1.0. Intuitively, this indicates there is an upper bound. However, the latency values are acquired using a discrete-event simulation run, which has the following two consequences. First, the simulation run comprises 1000 instances. In iDSL, this means that the latencies are derived from the first 1000 instances that complete. Hence, in-

stances that take long or even forever to finish, will be *overtaken* by instances that started later. Consequently, the latencies of very slow instances are neglected. Second, the performance properties are the result of random choices, viz., choosing between "ProcessCreditCard" (with probability $\frac{3}{4}$) and "AcceptCashOrCheck" (with probability $\frac{1}{4}$). In practice, "AcceptCashOrCheck" does not get selected relatively frequently, hence, a simulation run is unlikely to display worst case behavior. Moreover, when we manually compute the execution time, it shows that activities "PaymentMethod", "AcceptCashOrCheck", and "PreparePackageForCustomer" take $5+30+15=50$ time units. This value is higher than any of the inter-arrival times, viz., they are maximally 41, which means that in the worst case an increasing queue occurs.

The just mentioned performance properties give rise to the following two suggestions for improvement:

1. In order to decrease the best latency of 30, it is an option to make the "Cashier" perform better. Replacing the current cashier by one with a higher rate of $1+\frac{1}{p}$ leads to a best latency of $\frac{30}{1+\frac{1}{p}}$. Further improvement can be obtained by adding more cashiers to the resources, which can then perform activities in parallel.
2. If possible, try to increase the inter-arrival time as much as possible, i.e., 41, so that the amount of concurrency is the least. Ideally, the inter-arrival time is at least 50 to ensure that instances have finite latencies.

## 7.2 A segment of a process with more details

In this section, we apply the performance visualization of "A segment of a process with more details" (cf. Figure 18), which is the result of executing iDSL specification (cf. Table 57), to its informal BPMN process (cf. Figure 5). Table 35 provides a textual representation of Figure 18. The results

| **Prob.** | **0** | **0.1** | **0.2** | **0.3** | **0.4** | **0.5** | **0.6** | **0.8** | **0.9** | **1.0** |
|---|---|---|---|---|---|---|---|---|---|---|
| **rt=10** | 3 | 3 | 3 | 3 | 3 | 3 | 22 | 41 | 60 | 60 |
| **rt=20** | 3 | 3 | 3 | 3 | 3 | 32 | 32 | 71 | 91 | 151 |
| **rt=30** | 3 | 3 | 3 | 4 | 38 | 42 | 81 | 124 | 169 | 572 |
| **rt=40** | 3 | 38 | 102 | 169 | 267 | 385 | 522 | 866 | 1129 | 2370 |
| **rt=50** | 3 | 188 | 373 | 549 | 731 | 865 | 1016 | 1381 | 1630 | 2731 |
| **rt=60** | 3 | 306 | 522 | 675 | 858 | 1026 | 1199 | 1644 | 1991 | 3523 |

Table 35: A textual-representation of the cumulative latency distributions of "A segment of a process with more details" for different designs.

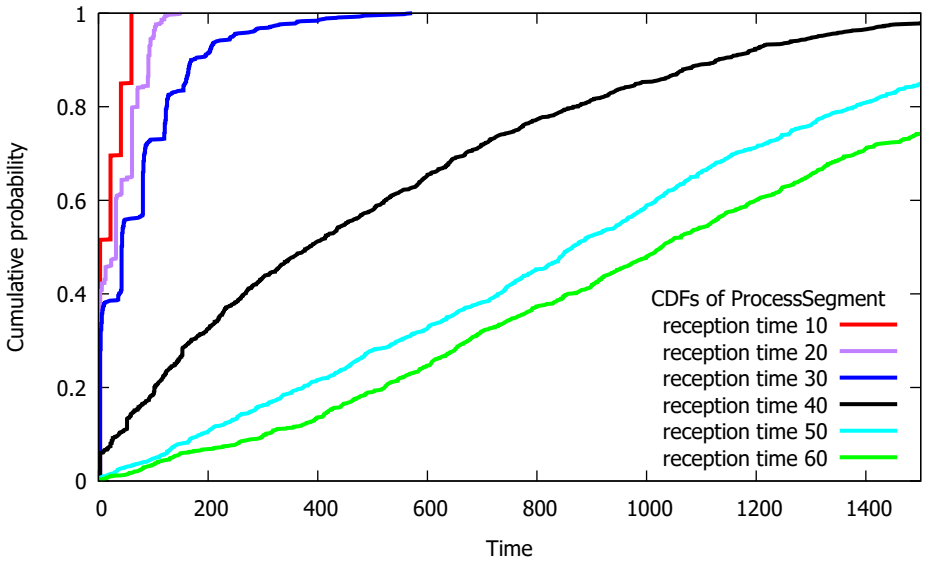



Figure 18: Cumulative latency distributions of "A segment of a process with more details" for different designs.

are created using discrete-event simulations (cf. Section 4.2.2).

Just like Figure 17 (of Section 7.1), Figure 18 is a cumulative probability graph. It contains plots for multiple designs that range on the dimension "reception-time" from 10 till 60 with steps of 10. They correspond to the time it takes to receive a quote, i.e., the load of activity "ReceiveQuote".

Figure 5 conveys that the whole BPMN process primarily consists of a compound activity that repeats either 0 (in case of no suppliers) or 1, 2 or 3 times. The compound activity includes a significant activity "ReceiveQuote" whose load depends on the design dimension "reception-time". All the activities are performed by one resource "ServerNode" with rate 1. Using this information, we draw the following performance properties from Figure 18 and Table 35:

1. Figure 18 displays that when the reception time is at least 40, the latency is significantly higher than when it is below 30. This is directly related to how often activity "ReceiveQuote" executes each instance, which is 0 (with probability $\frac{1}{2}$ due to a nondeterministic choice), and 1, 2 and 3 (with probability $\frac{1}{6}$ each). The inter-arrival time of the instances is 80. Therefore, for a reception time of at least 40, only 1 execution of "ReceiveQuote" can be performed within 80 times units, for 30, 2 executions, and for at most 20, 3 executions.
2. Figure 22 (of Appendix D.1) conveys, for different reception times, the latencies of the first 1000 instances as displayed from left to right. In line with Figure 18, the latencies are higher whenever the reception in time is higher. Furthermore, it can be seen here is that adjacent

instances vary more in latency, i.e., jitter. This is because the number of times "ReceiveQuote" is executed has a greater effect on the latency for designs with a larger reception time.

We derive from the just mentioned performance properties that it is key to keep the reception-time as low as possible. Especially when the reception-time increases from 30 to 40, a large increase in latency is noticeable.

### 7.3 A process with pools

In this section, we apply the performance visualization of "A process with pools" (cf. Figure 19), which is the result of executing iDSL specification (cf. Table 58), to its informal BPMN process (cf. Figure 7). The results are created using discrete-event simulations (cf. Section 4.2.2).

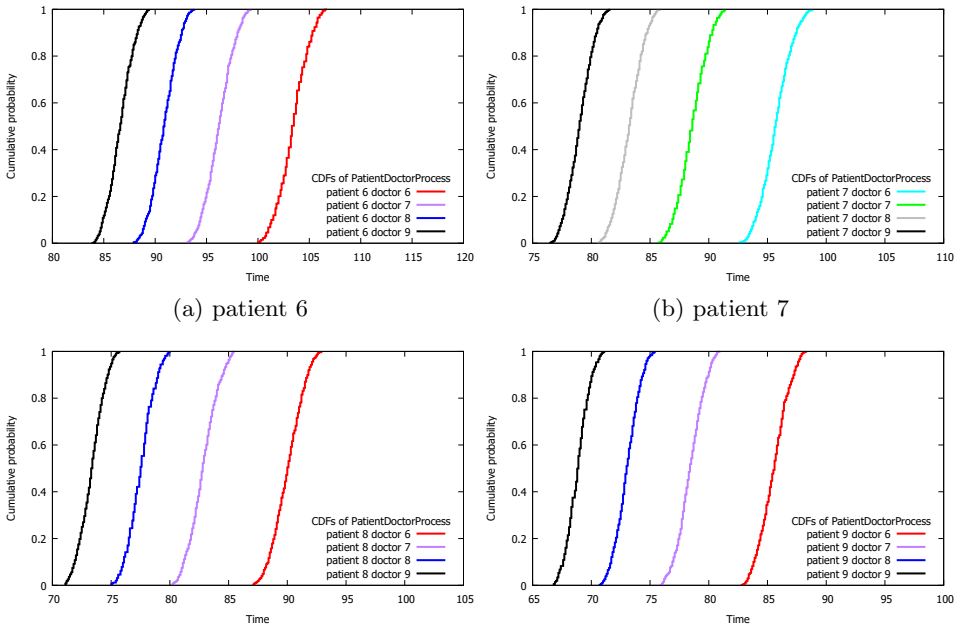
The design space of this business process contains two dimensions, viz., "patientrate" and "doctorrate". Therefore, Figure 19 has one subgraph for each "patientrate" (cf. Figure 19a till 19d) which in turn have one plot for each "doctorrate".

Comparing the different subgraphs conveys that the higher the "patientrate" is, the lower the service execution becomes, viz., the rate of resource "patient" directly depends on "patientrate" and as a consequence all processes that map to this resource are performed faster. Similarly, a higher "doctorrate" within a subgraph yields a higher rate for resource "doctor", which in turns leads to a faster execution of processes that map to this resource.

The iDSL code of Table 58 shows a uniform load for process "sendDoctorRequest" which depends on "patientrate". Consequently, all CDF graphs are not constant for different probabilities. Moreover, graphs with a lower "patientrate" convey more variation because the range of the uniform load is larger for those.

Figure 7 displays that the whole BPMN process is a sequence of activities. Hence, the flow of the process is very simple and deterministic. Contrarily, the load of the different activities depend on "doctorrate" and "patientrate" from the design space, and two uniform loads that make the load probabilistic. We derive the following performance properties from Figure 19:

1. The design space contains two dimensions which are both rates. The higher these rates are, the lower the overall latency is. For instance, the plots for "patientrate" equals 7 (cf. Figure 19b) are lower than for "patientrate" equals 6 (cf. Figure 19a). For another instance, Figure 19a conveys that, for a fixed "patientrate" of 6, the "doctorrate" and its latency correlate negatively.
2. The iDSL specification of Table 58 contains two uniform loads, which are accumulated to determine the overall latency. When many uniform




(a) patient 6 (b) patient 7

(c) patient 8 (d) patient 9

Figure 19: Cumulative latency distributions of "A process with pools" for different designs.

loads are added together, their sum starts to look like the normal distribution [160]. Consequently, all the plots in Figure 19 have different values for different probabilities and are moreover shaped like a normal distribution.

3. The just mentioned uniform loads both depend on the dimension "patientrate" from the deisng space. Hence, the larger the "patientrate" is, the less the variation in the latency. For instance, the latency ranges ranges from 81 till 91 when "patientrate" equals 9 and "doctorrate" equals 6, opposed to 100 till 118 when "patientrate" equals 6 and "doctorrate" equal 6. The range is not only smaller in absolute sense, viz., $|81 - 91| < |100 - 118|$, but also relatively, viz., $\frac{91}{81} < \frac{118}{100}$.
4. The worst case scenario occurs when all activities take their maximum execution time occurs when: (i) "patientrate" equals 6; "doctorrate" equals 6; and, both uniform distributions return their maximum value. The worst case scenario is

$$\frac{70 + 30 + 40 + 70 + 50 + 100 + 80 + 40 + 20 + 30 + 70 + 40}{6} \approx 107$$

This value is higher than the inter-arrival time of 100 in Table 58. When the most optimistic scenario for the distributions is being con-

sidered, we find a high value that is still rather high, namely

$$\frac{50 + 30 + 40 + 70 + 50 + 100 + 80 + 40 + 20 + 30 + 70 + 20}{6} = 100$$

We observe that CDF for "patientrate" and "doctorrate" equal 6 has value 118 for probability 1. This means the following two things. First, buffering took place because the latency of 118 is larger than the maximum 107 for when there is no concurrency. Second, there have been unfinished instances, because the maximum latency of 118 is too low to account for the instance latencies that continuously exceed the inter-arrival time.

The just mentioned performance properties give rise to suggestions for improvement, as follows:

1. As seen, improving either the "patientrate" and/or "doctorrate" yields direct improvements in latency, hence, having these rates as high as possible is recommendable.
2. The iDSL specification (cf. Table 58, line 47) conveys that there can be 10 simultaneously-active instances. Also, the business process has many activities. Therefore, it is worth having multiple patient and doctor resources whose mapping can be distributed over the different activities to enable parallelism.

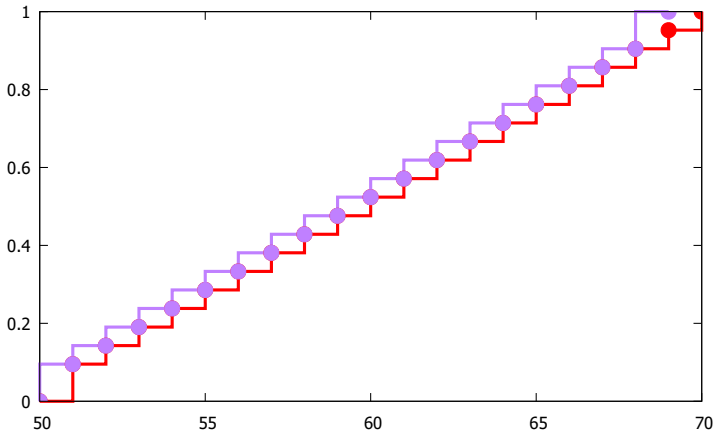
### 7.4 A segment of a process with lanes

In this section, we apply the performance visualization of "A segment of a process with lanes" (cf. Figure 20), which is the result of executing iDSL specification (cf. Table 59), to its informal BPMN process (cf. Figure 9). Unlike the previous cases, PTA model checking (cf. Section 4.2.4) has been used as a means here to retrieve results instead of discrete-event simulations.

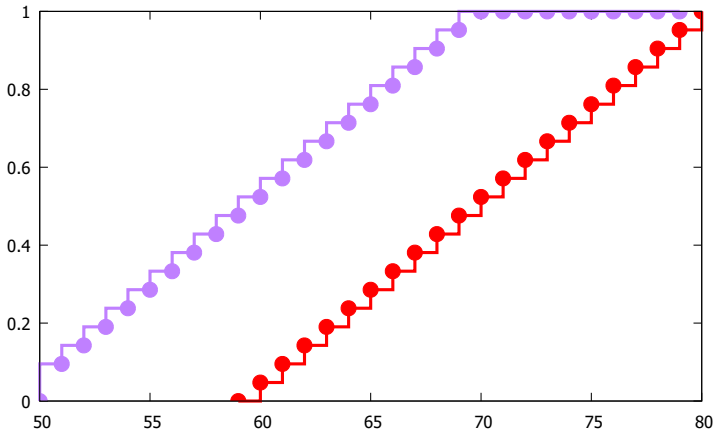
Figure 20 is again a cumulative distribution function. However, it contains two plots, a minimum and maximum plot, to express differences that are the result of nondeterminism, i.e., the area between the plots is the result of nondeterminism.

The business process starts with a process "DispatchToApprover" that has a uniform load on [10 : 30]. After this, a process "PreparePO" with a fixed load of 40 and a process "ApproveRequest" are executed in parallel. The load of "ApproveRequest" is a binary nondeterministic choice and moreover depends on design space dimension "adminoverhead". Figure 20 conveys performance properties for each of the three designs. We discuss each design individually, as follows.

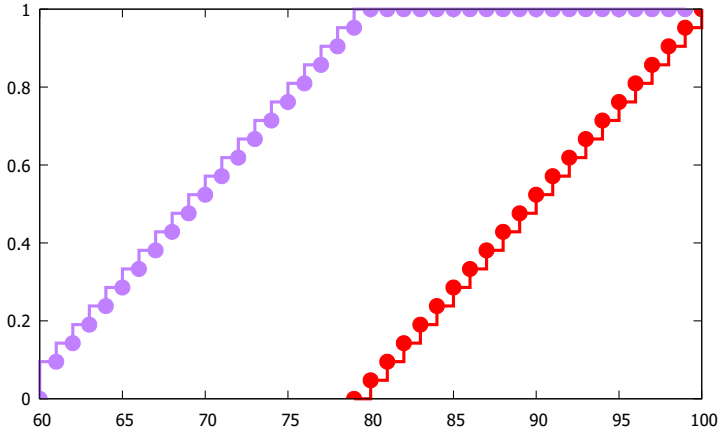
1. Figure 20a conveys the case in which "adminoverhead" equals 0. The execution time of "DispatchToApprover" is uniformly drawn from [10 :




(a) admin overhead 0




(b) admin overhead 20




(c) admin overhead 40

Figure 20: Cumulative latency distribution ranges of "A segment of a process with lanes" for different designs.

30], "PreparePO" is 10 + 20 + 10 = 40, and "ApproveRequest" is either 10 or 30, nondeterministically chosen. Since the execution time of "Prepare" is larger than the one of "ApproveRequest", we neglect the execution time of "ApproveRequest" and thereby any presence of nondeterminism. Therefore, the total execution time is a uniform draw from [10 : 30] plus 40, which can be rewritten as [50 : 70].

2. Figure 20b show the case in which "adminoverhead" equals 20. The execution time of "DispatchToApprover" is again uniformly drawn from [10 : 30], "PreparePO" is $10 + 20 + 10 = 40$, and "ApproveRequest" is either 30 or 50. Depending on the execution time of "ApproveRequest" either one of the following two options occurs. First, when "ApproveRequest" finishes after 30 time units, activity "Prepare" will take longer to finish, hence, the result is like Figure 20a. Second, when "ApproveRequest" finishes after 50 times units, it will take 10 time units longer than "PreparePO". Consequently, the result is like Figure 20a but 10 time units longer for each probability. The former option is the lower bound of the cumulative distribution function and the latter option the upper bound.
3. Figure 20c describes the case in which "adminoverhead" equals 40. The execution time of "DispatchToApprover" is again uniformly drawn from [10 : 30], "PreparePO" is 10+20+10 = 40, and "ApproveRequest" is either 50 or 70 time units, chosen nondeterministically. Hence, "PreparePO" always finishes before "ApprovedRequest". Figure 20c shows that the minimum latency is when "ApproveRequest" equals 50 and the maximum when "ApproveRequest" equals 70.

The aforementioned performance properties lead to the following suggestions for improvement:

1. Figure 20 conveys that the performance is better when "adminoverhead" is lower. Hence, ideally an "adminoverhead" of 0 is preferred.
2. The three activities are nicely mapped to three resources. Therefore, the performance can only be improved by increasing the rate of one or more resources.

### 7.5 A high-level business process

In this section, we apply the performance visualization of "A high-level business process" (cf. Figure 21), which is the result of executing iDSL specification (cf. Table 60), to its informal BPMN process (cf. Figure 11). Table 36 provides a textual representation of Figure 21 including the difference ($\delta$) between *pmin* and *pmax* for each time unit. Just like the previous section, PTA model checking (cf. Section 4.2.4) has been used as a means to retrieve results. On top of this, we provide a so-called anytime algorithm that produces intermediate results (cf. Appendix D.2).

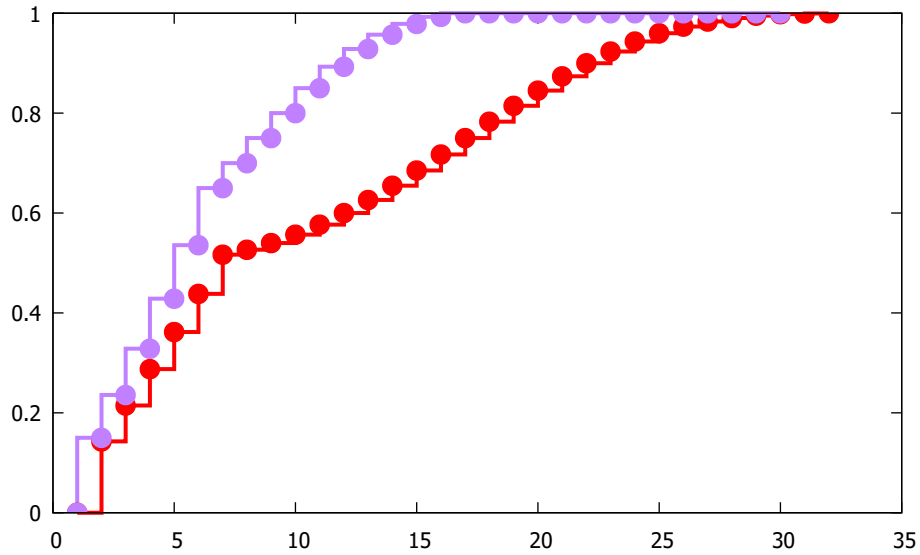



Figure 21: Cumulative latency distribution range of “A high-level business process”.

| **value** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **pmin** | 0.00 | 0.14 | 0.21 | 0.29 | 0.36 | 0.44 | 0.52 | 0.53 | 0.54 | 0.56 |
| **pmax** | 0.00 | 0.15 | 0.24 | 0.33 | 0.43 | 0.54 | 0.65 | 0.70 | 0.75 | 0.80 |
| $\delta$ | 0.00 | 0.01 | 0.03 | 0.04 | 0.07 | 0.10 | 0.13 | 0.17 | 0.21 | 0.24 |

| **value** | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|
| **pmin** | 0.58 | 0.60 | 0.63 | 0.65 | 0.69 | 0.72 | 0.75 | 0.78 | 0.81 | 0.85 |
| **pmax** | 0.85 | 0.89 | 0.93 | 0.96 | 0.98 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 |
| $\delta$ | 0.27 | 0.29 | 0.30 | 0.31 | 0.29 | 0.27 | 0.25 | 0.22 | 0.19 | 0.15 |

| **value** | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|
| **pmin** | 0.87 | 0.90 | 0.92 | 0.94 | 0.96 | 0.97 | 0.98 | 0.99 | 1.00 | 1.00 |
| **pmax** | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| $\delta$ | 0.13 | 0.10 | 0.08 | 0.06 | 0.04 | 0.03 | 0.02 | 0.01 | 0.00 | 0.00 |

Table 36: A textual representation of the cumulative latency distribution range of “A high-level business process”.

Figure 11 conveys that the BPMN process comprises a sequence of three activities. Moreover, the process can abort, i.e., terminate without executing any more activities, after the first and second activity. The iDSL specification (of Table 60) conveys that the first abort happens probabilistically with a probability of $\frac{1}{2}$ and the second abort with a nondeterministic choice.

Furthermore, it shows that there is one resource "processor" with rate 1, which means that a task load 1 of corresponds to 1 time unit of processing. The iDSL specification also reveals that the worst case scenario if the three activities, viz., $7 + 10 + 15 = 32$, is smaller than the inter-arrival time of 35. Hence, there is no concurrency between instances.

The three subsequent activities of the BPMN process all have a load that is drawn from a uniform distribution, viz., on $[1 : 7]$, $[1 : 10]$ and $[1 : 15]$, respectively. We recognize three different cases, as follows.

1. The BPMN process aborts after the first activity with probability $\frac{1}{2}$ and a uniformly distributed execution time on $[1 : 7]$.
2. The BPMN process aborts after the second activity, hence, it continues after the first activity with probability $\frac{1}{2}$ and aborts in a nondeterministic way after the second activity. The total execution time is the sum of the execution times of the first and second activities, which is a uniform distribution on $[1 : 7]$ followed by one on $[1 : 10]$.
3. The BPMN process completes all three activities. It continues after the first activity with probability $\frac{1}{2}$ and avoids an abort. in a nondeterministic way after the second activity. The total execution time is determined by the execution time of all three activities, i.e., a uniform distribution on $[1 : 7]$, one on $[1 : 10]$, and finally one on $[1 : 15]$

Using these three different cases, we informally explain the results in Table 36 and graphically depicted in Figure 21, as follows.

1. For *pmax* (see Table 36), only the first two activities are of concern. Up to value 7, the probability increases with relatively large steps, i.e., about 0.1 per value. The large increase is caused by two factors that contribute to the probability, viz., the case only the first activity has been executed and the case in which the first and second activity have been executed. Contrarily, it can be seen that the probability increases less between value 7 and 17, because the increase can be explained by only the second activity finishing.
2. To explain the case of *pmin*, we use *pmax* as a basis and take into account that the third activity gets executed too as a result of nondeterminism. It can be seen in Figure 21 that for probabilities up to 0.5, *pmin* and *pmax* are similar. Namely, the probability that the BPMN process aborts after the first activity is 0.5 and when it does the third activity does not get executed for sure.
3. In case of a probability of at least 0.5, *pmin* and *pmax* diverge as Figure 36 displays. Namely, the higher the total latency becomes, the more this can be the result of the latency of the third activity, which is the difference between *pmin* and *pmax*.

Next, we introduce a so-called anytime algorithm [79, 170], which is based on PTA model checking (cf. Section 4.2.4). This algorithm provides inter-

mediate results so that the system designer can already make performance estimates before the algorithm has finished. For this purpose, the PTA model checking algorithm performs multiple iterations of TA model checking that are executed in sequence and each yield a result; each result is used to improve the overall result. Figure 23 (in Appendix D.2) conveys 14 out of 30 intermediate results for *pmin* of "A high-level business process". We make the following observations:

1. Figure 23 shows that when the number of steps increases, the total area between *pmax* and *pmin* decreases. This is inherent to this algorithm as each TA model checking iteration reduces this area, i.e., increases the amount of information. For instance, the left rectangle in Figure 23a has been reduced to three smaller rectangles in Figure 23b as a result of two TA model checking iterations.
2. For fast results, the anytime algorithm performs TA model checking first on the area that has the largest height, i.e., probability difference. For instance, the left rectangle in Figure 23a gets selected first twice for TA model checking because of its height of 0.7 as Figure 23b shows. In the following steps, the top rectangle finally gets selected as Figure 23c conveys.
3. It can be seen that a shape similar to the shape after step 30 (see Figure 23o) is attained relatively early. For instance, the area between *pmax* and *pmin* is relatively small after step 12 (see Figure 23h) and almost minimized after step 18 (see Figure 23k). Therefore, the system designer can save roughly about half the time it takes the full algorithm to run and yet have results that approximate the final result well.

# 8 Conclusion

In this thesis, we have investigated how the evaluation of the performance of business processes can be done; we used the following six steps.

First, we determined what a business process is. In literature, it is depicted as a collection of business activities that transform a certain input into a output valuable to a business. Business activities can be obtained via a combination of vertical and horizontal differentiation of a business, i.e., splitting a business in subsequent and parallel activities, respectively. A business activity is performed by a resource, which need to have the right capabilities to do so. A resource is scarce because can only perform activity at once.

Second, we have researched which formalisms for business processes are there. Business process model and notation (BPMN) appears to be the de facto standard. Because of its visual language, BPMN enables the communication of a wide variety of information and to different audiences. Moreover, BPMN is designed to cover many types of modelling and at different levels of abstraction. A prominent BPMN paper [164] introduces six business processes that cover the typical BPMN constructs, including a sequence, choice, repeat, abort and composition construct.

Third, we have addressed the need for knowing the performance of a business process. Using the six aforementioned business processes, we show that gaining insight in their performance can lead to better understanding how a business process functions.

Fourth, literature provides many techniques for performance evaluation, including various automata, queuing networks and Petri-nets. These techniques can conveniently be applied using a number of well-known tools, such as Modest, STORM, PIPE2, POOSL, UPPAAL, PRISM and iDSL. We have decided to use iDSL as the approach in this thesis, because: (i) we have access to its source code; (ii) its high-level language; and, (iii) its support for various evaluation techniques such as discrete-event simulations, TA model checking and (iterative) PTA model checking.

Fifth, we have used two performance evaluation techniques, viz., DSEs and iterative PTA model checking. Generally, DSE scales well because it only considers one outcome per run at random, i.e., DSE yields average behaviour. However, DSE does not convey whether instances have an upper bound since it will always return a finite value. DSE cannot address nondeterminism as it uses a random distribution to deal with alternatives. Therefore, iterative PTA model checking has been used for the remaining two business processes. Iterative PTA model checking comprises a number of exhaustive PTA model checking iterations. This makes its results very accurate, but also rather expensive to obtain. Hence, the application of iterative PTA model checking is limited.

Sixth and finally, we apply the aforementioned performance techniques

to business processes, viz., DSE was used in four business processes and PTA model checking in two. For this purpose, we manually transformed the six aforementioned business processes into a textual representation with an underlying grammar. Moreover, we extended this representation for it to contain all the concepts iDSL requires. An iDSL model is then derived which is executed using the iDSL toolset. This leads to, among others, a or more visualizations which can be used to evaluated the performance.

**Results.** We have investigated the results for the different business processes and came to the following conclusions. First, when a resource is a so-called bottleneck, the best case latency of a business process is influenced much by that resource.

Second, design spaces have proven to be a useful and efficient mechanism to vary and optimize parts of a business process, e.g., loads and rates.

Third, using (iterative) PTA model checking exact bounds can be determined, which cannot be exceeded in practice.

Fourth, a so-called anytime algorithm makes it possible to retrieve approximate results before the whole algorithm has finished executing. Generally, the intermediate results are similar to the final result in an early stage, e.g., about halfway, because the anytime algorithms attempts to reduce the differences in probabilities.

**Future work.** We make four recommendations for future work. First, it would be interesting to make transformations to one or more performance models other than iDSL to test the generalizability of the approach. This includes checking whether the results are as easy to retrieve and similar.

Second, six BPMN processes have been used as an example in this thesis that introduced a variety of constructs for generating BPMN processes. It might be interesting to look for more constructs, either from other BPMN processes or other languages, that can be used in the BPMN context.

Third, the approach can be tested on more extensive business processes to test the expressibility of the underlying language and to see if the iDSL toolset can handle the resulting iDSL models that are larger.

Fourth and finally, we consider the correctness of the iDSL model that a transformation from BPMN brings about, which is currently represented in Xtend code. However, this Xtend code is not thoroughly checked for correctness yet. For instance, a specification can be used to relate input properties of BPMN to output properties of iDSL. For another instance, it can be useful to design several test cases that pinpoint the different aspects of a BPMN model and see if they are transformed well into iDSL. A test case typically consist of one BPMN model and a iDSL model it has to be transformed into.

# A The regular and extended BPMN grammar

This appendix contains both the regular and extended grammar of BPMN; the extended grammar extends the regular one. The regular grammar of Section A.1 is used for the six simple BPMN examples (cf. Section 3.1 till 3.1). On top of that, the extended grammar of Section A.2 is used for the six extended BPMN examples (cf Section 6.1 till 6.6).

## A.1 The regular BPMN grammar

Table 37 displays the regular BPMN grammar. It comprises a tree structure of value chains (lines 4-23), which have business activities (line 25) as leafs.

Table 37: The grammar of regular BPMN

```
1.  Model:
2.   'Regular' 'BPMN' 'model' business+=ValueChain;
3.
4.  ValueChain:
5.   RepeatValueChain | EmptyValueChain | BusActValueChain;
6.
7.  BusActValueChain:
8.   ParallelValueChain | AlternateValueChain | SequentialVal-
     ueChain;
9.
10. RepeatValueChain:
11.  '(' rvc=BusActValueChain ')' '><' '('
     vcs+=SequentialValueChain ')';
12.
13. EmptyValueChain:
14.  {EmptyValueChain} '.';
15.
16. ParallelValueChain:
17.  ba=BusinessActivity '||' '(' vcs+=ValueChain+ ')';
18.
19. AlternateValueChain:
20.  ba=BusinessActivity '+' '(' vcs+=ValueChain+ ')';
21.
22. SequentialValueChain:
23.  ba=BusinessActivity ';' vcs+=ValueChain;
24.
25. BusinessActivity: name=ID;
```

## A.2 The extended BPMN grammar

Table 38 displays the extended BPMN grammar. Besides the aforementioned structure of the regular BPMN grammar (lines 9-31), the extended BPMN grammar contains five additional sections, viz., resource, mapping, scenario, measure and design space. They provide the additionally needed information to transform into a iDSL model (cf. Table 55 of Appendix C.2).

Table 38: The grammar of extended BPMN

```
1.  Model: 'Extended' 'BPMN' 'model'
2.   business+=ValueChain
3.   resource+=Resource+
4.   mapping+=Mapping+
5.   scenario+=Scenario
6.   measure+=Measure+
7.   dspace+=DesignSpace;
8.
9.  ValueChain:
10.  RepeatValueChain | EmptyValueChain | ProbValueChain | Bus-
     ActValueChain;
11.
12. BusActValueChain:
13.  ParallelValueChain | AlternateValueChain | SequentialVal-
     ueChain;
14.
15. RepeatValueChain:
16.  'VC' rvc=BusActValueChain '*' ft=FrequencyTable 'VC'
     vcs+=SequentialValueChain;
17.
18. EmptyValueChain:
19.  {EmptyValueChain} '.';
20.
21. ParallelValueChain:
22.  ba=BusinessActivity '||' '(' 'VCs' vcs+=ValueChain+ ')';
23.
24. AlternateValueChain:
25.  ba=BusinessActivity '+' '(' 'VCs' vcs+=ValueChain+ ')';
26.
27. SequentialValueChain:
28.  ba=BusinessActivity ';' 'VC' vcs+=ValueChain;
29.
```

```
30.  BusinessActivity:
31.   'BA' name=ID load=Load;
32.
33.  ProbValueChain:
34.   prob+=INT+ vc+=ValueChain+;
35.
36.  Load: {Load}
37.   'load' load=AExp
38.   | 'load' 'uniform' '(' min=AExp ',' max=AExp ')'
39.   | 'load' 'fixed' '(' min=AExp ',' max=AExp ')';
40.
41.  FrequencyTable: {FrequencyTable}
42.   '(' pf+=ProbabilityFrequency+ ')';
43.
44.  ProbabilityFrequency:
45.   '(' probability=INT ',' frequency=INT ')';
46.
47.  Resource:
48.   'Resource' name=ID 'rate' rate=AExp;
49.
50.  Mapping:
51.   'Mapping' '(' 'Resource' resource=ID 'performs' 'BusinessActivity' busines-
     sactivity=ID ')';
52.
53.  Scenario:
54.   'Trigger' 'at' fst=AExp ',' snd=AExp ',' '...'  'using' numinstances=AExp
     'instances';
55.
56.  Measure:
57.   MeasureSimulation | MeasureModelChecking;
58.
59.  MeasureSimulation:
60.   numruns=INT 'simulation' 'runs' 'of' numrequests=INT 'requests';
61.
62.  MeasureModelChecking: {MeasureModelChecking}
63.   'PTA' 'model' 'checking';
64.
```

```
65.  DesignSpace:
66.   'DesignSpace' (dimension=DesignSpaceDimension+ | 'noDimensions');
67.
68.  DesignSpaceDimension:
69.   name=ID '(' values=STRING+ ')';
70.
71.  AExp: val=INT | 'dspace' '('dim=ID')' | '(' a1=AExp op=Op a2=AExp ')';
72.  Op: '+' | '-' | '*' | '/';
```

# B Graphviz code generator and code examples

This appendix contains the constructed Xtend code (in Table 39 of Appendix B.1) that has been used to transform the BPMN formalizations into corresponding Graphviz code (as conveyed in Table 40 till 46 of Appendix B.2).

## B.1 The Graphviz code generator

- Function **doGenerate** (lines 18-23) extracts the model from the resource, applies the generateGviz function to it, and writes the result to a file.
- Function **subgraphCount** (lines 27-30) is used to keep track of a global counter (line 25) to uniquely identify each created cluster.
- Overloaded function **generateGviz** (lines 32-43) takes a model and delegates the generation to a specific generateGviz function (lines 57-111), depending on the model type.
- Function **returnFirstBusinessActivity** (lnes 45-51) returns the first business activity of a Value Chain. In the simple case, this is the .ba field, but in case of a repeat value chain, it is the first business activity of the Value Chain.
- Function **returnLastBusinessActivity** (lines 53-58) returns the last business activity of a sequential chain. In the simple case, this is the .ba field. However, in case the Sequential Value Chain contains another Sequential Value Chain, are recursive call to the function is made using the latter Sequential Value Chain as a parameter.
- Function **returnLastBusinessActivity** (lines 60-64) calls the previous function when the parameter is a Sequential Value Chain and return null otherwise.
- Five overloaded functions **generateViz** (lines 57-111) provide an implementation for the high-level generateViz function (lines 32-43) for each Value Chain construct. They make use of recursive calls to generateViz functions when the grammar calls for it. Additionally, function returnFirstBusinessActivity (lines 46-52) and returnLastBusinessActivity (line 53-58) are used to extract Business Activities from the value chain. Finally, subgraphCount (lines 27-30) is used to assign unique numbers to clusters.

In this thesis, the generator has been used to transform the BPMN formalizations as presented in Table 2, 4, 5, 7, 9, 10 and 11 into corresponding Graphviz code, which can be found in of Table 40 till 46), respectively. After this, the Graphviz code for each use case has been executed using Graphviz [51, 63], which yields a number of visualizations as displayed in Figure 4, 6, 8, 10, 12 and 14, respectively).

Table 39: Xtend code that transforms BPMN model instances into Graphviz code

```
 1.  package org.generator
 2.
 3.  import org.eclipse.emf.ecore.resource.Resource
 4.  ...
15.  import org.bPMN.BusinessActivity
16.
17.  class BPMNGenerator extends AbstractGenerator {
18.   override void doGenerate(Resource resource, IFileSystemAccess2
          fsa, IGeneratorContext context) {
19.    var model = resource.allContents.toIterable.filter(typeof(Model))
          .toList.head
20.    var fsapath = resource.getURI().lastSegment+".gviz"
21.    System.out.println("Generating file "+fsapath)
22.    fsa.generateFile(fsapath,generateGviz(model))
23.  }
24.
25.   var static subgraphCounter=0 // a unique ID for each subgraph
26.
27.   def static subgraphCount (){
28.    subgraphCounter=subgraphCounter+1
29.    return subgraphCounter
30.   }
31.
32.   def static generateGviz(Model m)
33.   """ digraph{ generateGviz(m.business.head) } """
34.
35.   def static generateGviz(ValueChain vc){
36.    switch(vc){
37.     ParallelValueChain: generateGviz(vc)
38.     AlternateValueChain: generateGviz(vc)
39.     EmptyValueChain: generateGviz(vc)
40.     SequentialValueChain: generateGviz(vc)
41.     RepeatValueChain: generateGviz(vc)
42.    }
43.   }
44.
```

```
45.   def static returnFirstBusinessActivity(ValueChain vc){
46.    switch(vc){
47.     BusActValueChain: return vc.ba
48.     RepeatValueChain: return vc.rvc.ba
49.     default: return null
50.    }
51.   }
52.
53.   def static BusinessActivity returnLastBusinessActiv-
      ity(SequentialValueChain svc){
54.    switch(svc.vcs.head){
55.     SequentialValueChain: return returnLastBusinessActivity (svc.vcs.head)
56.     default: return svc.ba
57.    }
58.   }
59.
60.   def static BusinessActivity returnLastBusinessActivity(ValueChain vc){
61.    switch(vc){
62.     SequentialValueChain: returnLastBusinessActivity(vc)
63.     default: return null
64.    }
65.   }
66.
67.   def static generateGviz(ParallelValueChain pvc)
68.    """
69.    <<FOR vc:pvc.vcs>>
70.     <<var ba=returnFirstBusinessActivity(vc)>>
71.     <<IF ba!=null>>
72.     <<pvc.ba.name>>→<<ba.name>>[label="par"]
73.     <<ENDIF>>
74.     <<generateGviz(vc)>>
75.    <<ENDFOR>>
76.  """
77.
78.   def static generateGviz(AlternateValueChain avc)
79.    """
80.    <<FOR vc:avc.vcs>>
81.     <<var ba=returnFirstBusinessActivity(vc)>>
82.     <<IF ba!=null>>
83.      <<avc.ba.name>>→<<ba.name>>[label="alt"]
84.     <<ENDIF>>
85.     <<generateGviz(vc)>>
86.    <<ENDFOR>>
87.  """
```

```
88.
89.   def static generateGviz(EmptyValueChain evc) """""""
90.
91.   def static generateGviz(SequentialValueChain svc)
92.   """
93.   <<var ba=returnFirstBusinessActivity(svc.vcs.head)>>
94.   <<IF ba!=null>>
95.   <<svc.ba.name>>→<<ba.name>>
96.   <<ENDIF>>
97.   <<generateGviz(svc.vcs.head)>>
98.   """
99.
100.   def static generateGviz(RepeatValueChain rvc)
101.    """
102.    <<var counter=subgraphCount>>
103.    subgraph cluster_c<<counter>>{
104.     label="<<counter>>";
105.     <<generateGviz(rvc.rvc)>>
106.    }
107.    <<var lastBA = returnLastBusinessActivity(rvc.rvc)>>
108.    <<var firstBA = returnFirstBusinessActivity(rvc.vcs.head)>>
109.    <<lastBA.name>>→<<firstBA.name>>
110.    <<generateGviz(rvc.vcs.head)>>
111.    """
112.
113.  }
```

## B.2 The Graphviz code examples per use case

This appendix displays the generated Graphviz code for each business process (cf. Section 3.1 till 3.6) in Appendix B.2.1 till B.2.6, respectively. The Graphviz code for each process has been generated by applying the generator of Section B.1 on the extended BPMN intances of Section 6.1 till 6.6, respectively.

### B.2.1 A simple process

Table 40 display the generated code for "A simple process", which leads to Figure 4 in Section 3.1 after running Graphviz.

Table 40: The Graphviz code of "A simple process"

```
1.  digraph {
2.   IdentifyPaymentMethod→PaymentMethod
3.    PaymentMethod→AcceptCashOrCheck[label="alt"]
4.     AcceptCashOrCheck→PreparePackageForCustomer
5.    PaymentMethod→ProcesCreditCard[label="alt"]
6.    ProcesCreditCard→PreparePackageForCustomer
7.  }
```

### B.2.2 A segment of a process with more details

Table 41 conveys the generated code for "A segment of a process with more details". It yields Figure 6 in Section 3.2 after executing Graphviz.

Table 41: The Graphviz code of "A segment of a process with more details"

```
1.   digraph {
2.    AnySuppliers→SendNoSuppliers[label="alt"]
3.    AnySuppliers→SendRFQ[label="alt"]
4.    subgraph cluster_c2 {
5.     label="repeat_1";
6.     SendRFQ→ReceiveQuote
7.     ReceiveQuote→AddQuote
8.    }
9.    AddQuote→FindOptimalQuote
10.  }
```

### B.2.3 A process with pools

Table 42 shows the generated code for "A process with pools". It yields Figure 8 in Section 3.3 after executing Graphviz.

Table 42: The Graphviz code of "A process with pools"

```
1.  digraph {
2.      sendDoctorRequest→receiveDoctorRequest
3.      receiveDoctorRequest→sendAppointment
4.      sendAppointment→receiveAppointment
5.      receiveAppointment→sendSymptoms
6.      sendSymptoms→receiveSymptoms
7.      receiveSymptoms→sendPrescriptionPickup
8.      sendPrescriptionPickup→receivePrescriptionPickup
9.      receivePrescriptionPickup→sendMedicineRequest
10.     sendMedicineRequest→receiveMedicineRequest
11.     receiveMedicineRequest→sendMedicine
12.     sendMedicine→receiveMedicine
13. }
```

### B.2.4 A segment of a process with lanes

Table 43 displays the generated code for "A segment of a process with lanes". It yields Figure 10 in Section 3.4 after running Graphviz.

Table 43: The Graphviz code of "A segment of a process with lanes"

```
1.  digraph {
2.      DispatchToApprover→PreparePO[label="par"]
3.      DispatchToApprover→ApproveRequest[label="par"]
4.  }
```

### B.2.5 A high-level business process

Table 44 displays the generated code for "A high-level business process". Figure 12 in Section 3.5 shows the result after running Graphviz.

Table 44: The Graphviz code of "A high-level business process"

```
1.  digraph {
2.   pages2and3→abort[label="alt"]
3.   pages2and3→pages4and5and6and7[label="alt"]
4.   pages4and5and6and7→abort[label="alt"]
5.   pages4and5and6and7→pages8and9and10[label="alt"]
6.  }
```

### B.2.6 A hierarchical business process

Table 45 and 46 display the generated code for "A hierarchical business process", both folded and unfolded, respectively. In turn, Figure 13a and 13b in Section 3.6 shows the results after running Graphviz for both folder and unfolded, respectively.

Table 45: The Graphviz code of "A hierarchical business process" (folded)

```
1.  digraph{
2.   Procurement→Logistics
3.   subgraph cluster_c9{
4.    label="9";
5.   }
6.   Logistics→Sales
7.  }
```

Table 46: The Graphviz code of "A hierarchical business process" (unfolded)

```
1.  digraph{
2.   Procurement→Inboundlogistics
3.   subgraph cluster_c8{
4.    label="8";
5.    Inboundlogistics→Storage
6.    Storage→Outboundlogistics
7.   }
8.   Outboundlogistics→Sales
9.  }
```

# C The iDSL code generator, grammar and code

In this appendix, we present the iDSL code generator that turns extended BPMN instances into iDSL equivalents in Section C.1. After this, the corresponding grammar of iDSL in Section C.2 is presented, followed by the resulting iDSL code per use case in Section C.3.

## C.1 The iDSL generator

Below, we provide the semantics of BPMN via a transformation into iDSL in twofold. iDSL is the language of the performance evaluation approach that was introduced in Section 5.

First, Section C.1.1 contains high-level Xtend code that delegates BPMN constructs to iDSL ones and writes them to disk. Second, Section C.1.2 till C.1.7 present the Xtend code that is used to turn the extended BPMN intances of Section 6.1 till 6.6 into their iDSL counterparts, respectively. The code relies on the iDSL grammar that will be presented in Appendix C.2.

Note that the code is mainly meant a proof-of-concept and does not take the layout of the iDSL code it outputs into account. We have manually formatted the iDSL code, i.e., by adding spaces, tabs and newlines, however, without changing the semantics.

### C.1.1 The high-level code of the iDSL generator

At its highest level, the the Xtend code that transforms extended BPMN into its iDSL equivalents contains the following two functions:

- **dogenerate** (see Table 47) comprises the generator function, which: (i) retrieves the BPMN model (line 13); (ii) sets the location to store the iDSL model (line 14); (iii) converts the BPMN into iDSL using the generateiDSL method that is described next (line 15); and, (iv) writes it to a file (line 15).
- **generateiDSL** (see Table 48) is the high-level template to convert a BPMN model into an iDSL counterpart; each BPMN section (as mentioned in Section 6.1 till Section 6.6) is transformed into a respective iDSL section (as specified in Section 5.2.1 till 5.2.6). For this purpose, generateiDSL makes use of its constituent functions generateProcess, generateResource, generateSystem, generateScenario, generateMeasure and generateStudy, respectively.

In the remainder of this section, we will discuss how the BPMN sections transform into their iDSL equivalents in Section C.1.2 till C.1.7, respectively.

Table 47: The iDSL code generator (doGenerate)

```
1.  package org.generator
2.
3.  import java.util.List
4.  import java.util.ArrayList
5.  import org.eclipse.xtext.generator.AbstractGenerator
6.  import org.eclipse.xtext.generator.IFileSystemAccess2
7.  import org.eclipse.xtext.generator.IGeneratorContext
8.  import org.bPMN.Model
9.  import org.bPMN.*
10.
11. class BPMNGenerator extends AbstractGenerator {
12.  override void doGenerate(org.eclipse.emf.ecore.resource.Resource resource,
    IFileSystemAccess2 fsa, IGeneratorContext context) {
13.   var model = resource.allContents.toIterable.filter(
    typeof(Model)).toList.head
14.   var fsapath = resource.getURI().lastSegment+".idsl"
15.   fsa.generateFile(fsapath,generateIDSL(model))
16.  }
```

Table 48: The iDSL code generator (generateIDSL)

```
1.  def static generateIDSL (Model model)"""
2.   Section Process
3.    ProcessModel process_0
4.    seq { <<generateProcess(model.business.head)>> }
5.
6.   Section Resource
7.    ResourceModel resource_0
8.    <<generateResource(model.resource.head)>>
9.
10.  Section System
11.   Service service_0
12.    Process process_0
13.    Resource resource_0
14.    <<generateSystem(model.mapping)>>
15.
16.  Section Scenario <<generateScenario(model.scenario)>>
17.
18.  Section Measure <<generateMeasure(model.measure)>>
19.
20.  Section Study <<generateStudy(model.dspace.head)>>
21.  """
```

### C.1.2 From a BPMN "Business Process" to an iDSL "Process"

The generateProcess function (lines 1-92) is defined in Table 49 for different types using overloading and switch statements, as follows:

- Type: **ValueChain** (lines 1-8) yields a switch statement between RepeatValueChain, EmptyValueChain, BusActValueChain and ProbValueChain.
- Type: **List<ValueChain>** (lines 10-13) calls the function of type ValueChain for each item of the list.
- Type: (**List<Integer>** probs, **List<ValueChain>**) (lines 15-20) call the function of type ValueChain for each item of the list and prepends it with a corresponding probability.
- Type: **RepeatValueChain** (lines 22-24) leads to the generation of three components, viz., a value chain, a repeat keyword and a frequency table, respectively. For instance, it can be observed that the **times** construct in Table 28 (line 6) becomes a **repeat** construct in iDSL (see Table 57, lines 5-6).
- Type: **EmptyValueChain** (lines 26-28) yield an empty process which is represented by having no code iDSL. For instance, each **.** (dot) represents an empty process in Table 27 (line 5 and 6), which transform into no activity after PreparePackageForCustomer (see Table 56, line 6 and 9).
- Type: **BusActValueChain** (lines 30-39) results into a business process followed by switch statement between either ParallelValueChain, AlternativeValueChain, SequentialValueChain or ProbValueChain.
- Type: **ParallelValueChain** (lines 41-43) returns a par-construct in iDSL. For instance, each || in Table 30 (line 4 and 5) yields a par construct in iDSL (see Table 59, line 5 and 7).
- Type: **AlternativeValueChain** (lines 45-47) returns an iDSL alt-construct. For instance, each + in Table 31 (line 4 and 6) results into an alt construct in iDSL (see Table 60, line 5 and 7).
- Type: **SequentialValueChain** (lines 49-55) returns an iDSL seq-construct. For instance, each **;** occurence in Table 27 (lines 3-5) yields a seq construct in iDSL (as in Table 56, lines 3, 4, 6 and 9).
- Type: **ProbValueChain** (lines 57-59) returns an iDSL palt-construct. For instance, the + in Table 27 (line 4) leads to a palt construct in iDSL (see Table 56, line 5) when augmented with probabilities.

Additionally, the different generateProcess functions make use of the following five supporting functions:

- **generateFrequencyTable** (lines 61-63) yields a frequency table, which is used as part of the RepeatValueChain.
- **generateBusinessActivity** (lines 65-67) is used by BusActValueChain to create a business activity that proceeds a value chain, i.e., an iDSL

Table 49: The iDSL code generator (generateProcess)

```
1.  def static CharSequence generateProcess (ValueChain vc){
2.   switch(vc){
3.    RepeatValueChain: generateProcess(vc)
4.    EmptyValueChain: generateProcess(vc)
5.    BusActValueChain: generateProcess(vc)
6.    ProbValueChain: generateProcess(vc.prob,vc.vc)
7.   }
8.  }
9.
10. def static CharSequence generateProcess (List<ValueChain> vcs)"""
11.  <<FOR vc:vcs>><<generateProcess(vc)>>
12.  <<ENDFOR>>
13. """
14.
15. def static CharSequence generateProcess (List<Integer> probs,
    List<ValueChain> vcs)"""
16.  <<FOR cnt:0..probs.size-1>>
17.   <<probs.get(cnt)>>
18.   <<generateProcess(vcs.get(cnt))>>
19.  <<ENDFOR>>
20. """
21.
22. def static CharSequence generateProcess (RepeatValueChain rvc)"""
23.  repeat <<generateFrequencyTable(rvc.freqtab)>>
    <<generateProcess(rvc.rvc)>> <<generateProcess(rvc.vcs)>>
24. """
25.
```

business activity that comprises an *atom* and *load* keyword.

- **generateLoad** (lines 69-75) is used to create a load to be used by the empty value chain and business activity.
- **generateAExp** (lines 77-83) is used to create an arithmetic expression, which is used to, among others, quantify a load or rate.
- **generateOp** (lines 85-91) converts a mathematical operator into a string representation that are used in an arithmetic expression.

```
26.  def static CharSequence generateProcess (EmptyValueChain evc)
27.   """
28.  """
29.
30.  def static String generateProcess (BusActValueChain bvc){
31.   var ba=generateBusinessActivity(bvc.ba)
32.   var vc=bvc.vc
33.   switch(vc){
34.    ParallelValueChain: ba+generateProcess(vc).toString
35.    AlternateValueChain: ba+generateProcess(vc).toString
36.    SequentialValueChain: ba+generateProcess(vc).toString
37.    ProbValueChain: ba+generateProcess(vc).toString
38.   }
39.  }
40.
41.  def static CharSequence generateProcess (ParallelValueChain pvc)
42.   """
43.   par { <<generateProcess(pvc.vcs)>> }"""
44.
45.  def static CharSequence generateProcess (AlternateValueChain avc)
46.   """
47.   alt { <<generateProcess(avc.vcs)>> }"""
48.
49.  def static generateProcess (SequentialValueChain svc){
50.   var vcs=svc.vcs.head
51.   switch(vcs){
52.    EmptyValueChain: ""
53.    default: "seq { "+generateProcess(vcs)+" } "
54.   }
55.  }
56.
57.  def static CharSequence generateProcess (ProbValueChain pvc)
58.   """
59.   palt { <<generateProcess(pvc.vcs.map[prob],pvc.vcs.map[vc])>> }"""
60.
```

```
61.  def static CharSequence generateFrequencyTable (FrequencyTable ft)"""
62.   <<FOR pf:ft.pf>>(<<pf.probability>>:<<pf.frequency>>)
     <<ENDFOR>>
63.  """
64.
65.  def static CharSequence generateBusinessActivity (BusinessActivity
     ba)"""
66.   atom <<ba.name>> load <<generateLoad(ba.load)>>
67.  """
68.
69.  def static String generateLoad (Load load){
70.   switch(load){
71.    ConstantLoad: generateAExp(load.load)
72.    UniformLoad: "uniformLoad ("+generateAExp(load.min)+" "+gener-
     ateAExp(load.max)+")"
73.    NondeterministicLoad: generateAExp(load.min)+" "+generateA-
     Exp(load.max)
74.   }
75.  }
76.
77.  def static String generateAExp (AExp aexp){
78.   switch(aexp){
79.    AExpValue: aexp.value.toString
80.    AExpDspace: "dspace(¨"+aexp.dim+"¨)"
81.    AExpOp: generateAExp(aexp.a1)+generateOp(aexp.op)+ generateA-
     Exp(aexp.a2)
82.   }
83.  }
84.
85.  def static generateOp (Op operator){
86.   switch(operator){
87.    OpPlus: "+"
88.    OpMinus: "-"
89.    OpMultiply: "*"
90.    OpDivision: "/"
91.   }
92.  }
```

Table 50: The iDSL code generator (generateResource)

```
1.  def static CharSequence generateResource (Resource resource){
2.   switch(resource){
3.    AtomicResource: generateResource(resource)
4.    CompounedResource: generateResource(resource)
5.   }
6.  }
7.
8.  def static CharSequence generateResource (AtomicResource ar)"""
9.   atom <<ar.name>> rate <<generateAExp(ar.rate.rate)>>
10. """
11.
12. def static CharSequence generateResource (CompounedResource cr)"""
13.  decomp <<cr.name>> { <<FOR res:cr.res>>
    <<generateResource(res)>> <<ENDFOR>> }
14. """
```

Table 51: The iDSL code generator (generateSystem)

```
1.  def static CharSequence generateSystem (List<Mapping> mappings)"""
2.   <<FOR mapping:mappings>>(<<IF mapping.businessactivity!==null>>
    <<mapping.businessactivity>><< ELSE>>default
    <<ENDIF>>,<<mapping.resource>>)<<ENDFOR>>
3. """
```

### C.1.3 From a BPMN "Resource" to an iDSL "Resource"

The generateResource function is defined in Table 50 for three kinds of resources, as follows.

- Type: **Resource** (lines 1-6) is a switch statement between AtomicResource and CompooundResource, which are defined next. For instance, Table 27 (line 9) represents a composite resource "cashdesk" and an atomic resource "'cashier". In iDSL, they are a decomp and atom resource (Table 56, line 13), respectively.
- Type: **AtomicResource** (lines 8-10) is used to create an atomic resource with a rate.
- Type: **CompoundResource** (lines 12-14) is used to create a compound resource that consists of one or more resources. Each of these resource are either of type AtomicResource or CompoundResource in turn.

### C.1.4 From a BPMN "Mapping" to an iDSL "System"

Function **GenerateSystem** (as defined in Table 51) states which resources performs a which business activities, a so-called mapping. In iDSL, the order is reversed, i.e., the process precedes a resource.

Furthermore, iDSL supports the use of multiple high-level Services, Processes and Resources. In this thesis, we have not taken advantage of this feature. Instead, we assume there is one high-level Process, Resource and Service each. As a result, function **generateiDSL** (see Table 48) contains some fixed code (lines 3,7,11-13) in Section Process, Resource and System.

### C.1.5 From a BPMN "Scenario" to an iDSL "Scenario"

Function **GenerateScenario** as presented in Table 52 yields a scenario. For instance, a **TriggerAt** function with execution times (as in Table 27, line 15) results into a scenario and service which has the same execution times.

### C.1.6 From a BPMN "Measure" to na iDSL "Measure"

A measure (see Table 53 for the generation code) is either one of the following three kinds:

- Type: **Measure** (lines 5-10) leads to a switch statement between either MeasureModelChecking or MeasureSimulation.
- Type: **MeasureModelChecking** (lines 11-13) represents a model checking measure. In constrast to simulation, It has no parameters because its results are universal. For instance, model checking is defined in Table 30 (line 24) without parameters. It results into something similar in iDSL as Table 59 (line 45) demonstrates.
- Type: **MeasureSimulation** (line 15-16) represents a simulation measure, which requires a number of runs and a fixed length for each run. For instance, the measure defined in Table 27 (line 18) using a number of runs and requests. iDSL implements this in a similar fashion as Table 56 (line 27) illustrates.

Additionally, function **generateMeasure** (line 1-3) is applied to turn a number of measures into their iDSL counterparts.

### C.1.7 From a BPMN "Design Space" to an iDSL "Study"

A **GenerateStudy** as conveyed in Table 54 defines a study. Using an underlying design space, many designs can be defined in a convenient way. For instance, Table 29 (line 37) defines a design space that comprises two dimensions, viz., "patientrate" and "doctorrate", and $4 \times 4 = 16$ instances. In iDSL, this leads to similar code as Table 58 (lines 54-56) conveys.

Table 52: The iDSL code generator (generateScenario)

```
1.  def static CharSequence generateScenario (List<Scenario> scenarios)"""
2.   <<FOR scenario:scenarios>>Scenario scenario_0
3.    ServiceRequest service_0
4.    at time <<generateAExp(scenario.fst)>>,
    <<generateAExp(scenario.snd)>>, ... using
    <<generateAExp(scenario.numinstances)>> instances
5.  <<ENDFOR>> """
```

Table 53: The iDSL code generator (generateMeasure)

```
1.   def static CharSequence generateMeasure (List<Measure> measures)"""
2.    <<FOR measure:measures>><<generateMeasure(measure)>>
3.    <<ENDFOR>> """
4.
5.   def static CharSequence generateMeasure (Measure measure){
6.    switch(measure){
7.     MeasureModelChecking: generateMeasure(measure)
8.     MeasureSimulation: generateMeasure(measure)
9.    }
10.  }
11.  def static CharSequence generateMeasure (MeasureModelChecking
    mmc)"""
12.   Measure CDF of ServiceResponseTimes via advanced PTA model checking
13.  """
14.
15.  def static CharSequence generateMeasure (MeasureSimulation ms)"""
16.   Measure ServiceResponse times using <<ms.numruns>> runs of
    <<ms.numrequests>> ServiceRequests """
```

Table 54: The iDSL code generator (generateStudy)

```
1.  def static CharSequence generateStudy (DesignSpace ds)"""
2.    Scenario scenario_0
3.    <<IF !ds.dimension.empty>>DesignSpace <<ENDIF>>
4.    <<FOR dimension:ds.dimension>>( "<<dimension.name>>" { <<FOR
      value:dimension.dsvalues.head.v̂al>> "<<value>>"<<ENDFOR>> } )
5.    <<ENDFOR>>
6.  """
7.  }
```

## C.2 The iDSL grammar

Appendix C.1 presented the code that turns extended BPMN instances into iDSL equivalents. Below, we present the underlying iDSL grammar in Xtext format that has been used to accomplish this in Table 55. The grammar comprises a **Model** (lines 5-11) that decomposes into six parts, viz., a **Process** (lines 13-41), **Resource** (lines 43-47), **System** (lines 49-56), **Scenario** (lines 57-63), **Measure** (lines 65-71) and **Study** (lines 73-81).

Table 55: The grammar of iDSL

```
1.  grammar org.Idsl with org.eclipse.xtext.common.Terminals
2.
3.  generate idsl "http://www.Idsl.org"
4.
5.  Model:
6.   'Section Process' p +=ProcessModel+
7.   'Section Resource' r +=ResourceModel+
8.   'Section System' s +=System
9.   'Section Scenario' sc +=ScenarioModel+
10.  'Section Measure' m +=Measure+
11.  'Section Study' st +=Study;
12.
13. ProcessModel:
14.  'ProcessModel' name=ID ps +=ProcessStructure;
15.
16. ProcessStructure:
17.  ProcessStructureAtom | ProcessStructureSequence | ProcessStruc-
     tureProbAlternative | ProcessStructureAlternative | ProcessStruc-
     tureParallel | ProcessStructureRepeat;
18.
19. ProcessStructureAtom:
20.  'atom' name=ID +=Load;
21. ProcessStructureSequence:
22.  'seq' name+=ID? '{' ps+=ProcessStructure+ '}';
23. ProcessStructureProbAlternative:
24.  'palt' name+=ID? '{' pps+=ProbProcessStructure+ '}';
25. ProcessStructureAlternative:
26.  'alt' name+=ID? '{' ps+=ProcessStructure+ '}';
27. ProcessStructureParallel:
28.  'par' name+=ID? '{' ps+=ProcessStructure+ '}';
29. ProcessStructureRepeat:
30.  'repeat' pf+=ProbFrequencyPair+;
31.
```

```
32. Load:
33.  RegularLoad | UniformLoad;
34. RegularLoad:
35.  'load' load=AExp;
36. UniformLoad:
37.  'uniformLoad' '(' load1=AExp ',' load2=AExp ')';
38. ProbProcessStructure:
39.  prob+=INT p+=ProcessStructure;
40. ProbFrequencyPair:
41.  '(' probability=INT ':' frequency=INT ')';
42.
43. ResourceModel:
44.  'ResourceModel' name=ID 'decomp' '{' rs+=ResourceStructure+ '}'; //
    connections
45.
46. ResourceStructure:
47.  'atom' name=ID 'rate' rate=AExp;
48.
49. System:
50.  s+=Service+;
51. Service:
52.  'Service' name=ID 'Process' pname=ID 'Resource' rname=ID
    m+=Mapping;
53. Mapping:
54.  'Mapping' 'assign' '{' m+=MappingAssignment+ '}';
55. MappingAssignment:
56.  '(' process=ID ',' resource=ID ')';
57. ScenarioModel:
58.  'Scenario' name=ID s+=ServiceRequest+;
59.
60. ServiceRequest:
61.  'ServiceRequest' name=ID 'at' 'time' start=AExp ',' interval=AExp ',' '...'
    i+=ServiceRequestInstances?;
62. ServiceRequestInstances:
63.  'using' instances=AExp 'instances';
64.
```

```
65.  Measure:
66.   MeasureSimulation | MeasureModelChecking;
67.
68.  MeasureSimulation:
69.   'Measure' 'ServiceResponse' 'times' 'using' runs=INT 'runs' 'of' re-
      quests=INT 'ServiceRequests';
70.  MeasureModelChecking:
71.   {MeasureModelChecking} 'Measure' 'CDF' 'of' 'ServiceResponseTimes'
      'via' 'advanced' 'PTA' 'model' 'checking';
72.
73.  Study:
74.   'Scenario' name=ID d+=DesignSpace?;
75.
76.  DesignSpace:
77.   'DesignSpace' d+=DesignSpaceDimension+;
78.  DesignSpaceDimension:
79.   '(' name=STRING d+=DesignSpaceValues ')';
80.  DesignSpaceValues:
81.   '{' values+=STRING+ '}';
82.
83.  AExp:
84.   AExpInteger — AExpAExp — AExpDspace;
85.
86.  AExpInteger:
87.   value=INT;
88.  AExpAExp:
89.   '(' value1=AExp o=Op value2=AExp ')';
90.  AExpDspace:
91.   'dspace' '(' dimension=STRING ')';
92.
93.  Op:
94.   {Op} '/' | '*' | '+' | '-';
```

## C.3 The iDSL code per use case

Appendix C.1 presented the code that turns extended BPMN instances into iDSL equivalents on the basis of the iDSL grammar of Appendix C.2. Next, Appendix C.3.1 till C.3.5 displays the iDSL results of applying the generator to the BPMN instances of Section 6.1 till 6.5, respectively.

### C.3.1 A simple process

Table 56 shows iDSL code that is based on the BPMN code of Table 27.

Table 56: The iDSL example of "A simple process"

```
1.  Section Process
2.   ProcessModel process_0
3.    seq { atom IdentifyPaymentMethod load 0
4.     seq { atom PaymentMethod load 5
5.      palt { 1 atom AcceptCashOrCheck load 30
6.       seq { atom PreparePackageForCustomer load 15
7.       }
8.      3 atom ProcessCreditCard load 10
9.     seq { atom PreparePackageForCustomer load 15 } } } }
10.
11. Section Resource
12.  ResourceModel resource_0
13.   decomp cashdesk { atom cashier rate 1 }
14.
15. Section System
16.  Service service_0
17.   Process process_0
18.   Resource resource_0
19.   (default,cashier)
20.
21. Section Scenario
22.  Scenario scenario_0
23.   ServiceRequest service_0
24.    at time 0, dspace("interarrival"), ... using 10 instances
25.
26. Section Measure
27.  Measure ServiceResponse times using 1 runs of 1000 Ser-
     viceRequests
28.
29. Section Study
30.  Scenario scenario_0
31.   DesignSpace ( "interarrival" { "37" "38" "39" "40" "41" } )
```

### C.3.2 A segment of a process with more details

Table 57 shows iDSL code that is based on the BPMN code of Table 28.

Table 57: The iDSL example of "A segment of a process with more details"

```
1.  Section Process
2.   ProcessModel process_0
3.    seq { atom AnySuppliers load 1
4.     alt { atom SendNoSuppliers load 0
5.      repeat (1:3) (1:2) (1:1)
6.       atom SendRFQ load 4
7.       seq { atom ReceiveQuote load dspace("receptiontime")
8.      seq { atom AddQuote load 5 } }
9.    atom FindOptimalQuote load 2 } }
10.
11. Section Resource
12.  ResourceModel resource_0
13.   decomp server { atom servernode rate 1 }
14.
15. Section System
16.  Service service_0
17.   Process process_0
18.   Resource resource_0
19.   (default,servernode)
20.
21. Section Scenario
22.  Scenario scenario_0
23.   ServiceRequest service_0
24.    at time 0, 80, ... using 10 instances
25.
26. Section Measure
27.  Measure ServiceResponse times using 1 runs of 1000 Ser-
     viceRequests
28.
29. Section Study
30.  Scenario scenario_0
31.   DesignSpace ("receptiontime" {"10" "20" "30" "40" "50" "60"})
```

### C.3.3 A process with pools

Table 58 shows iDSL code that is based on the BPMN code of Table 29.

Table 58: The iDSL example of "A process with pools"

```
1.  Section Process
2.  ProcessModel process_0
3.   seq {
4.    atom sendDoctorRequest load uniformLoad
    (50/dspace("patientrate") 70/dspace("patientrate"))
5.    seq {
6.     atom receiveDoctorRequest load 30/dspace("doctorrate")
7.     seq {
8.      atom sendAppointment load 40/dspace("doctorrate")
9.      seq {
10.      atom receiveAppointment load 70/dspace("patientrate")
11.      seq {
12.       atom sendSymptoms load 50/dspace("patientrate")
13.       seq {
14.        atom receiveSymptoms load 100/dspace("doctorrate")
15.        seq {
16.         atom sendPrescriptionPickup load 80/dspace("doctorrate")
17.         seq {
18.          atom receivePrescriptionPickup load
    40/dspace("patientrate")
19.          seq {
20.           atom sendMedicineRequest load 20/dspace("patientrate")
21.           seq {
22.            atom receiveMedicineRequest load
    30/dspace("doctorrate")
23.            seq {
24.             atom sendMedicine load 70/dspace("doctorrate")
25.             seq {
26.              atom receiveMedicine load uniformLoad
    (20/dspace("patientrate") 40/dspace("patientrate"))
27.   } } } } } } } } } } } }
28.
```

```
29.  Section Resource
30.   ResourceModel resource_0
31.    decomp patientDoctorServ {
32.     atom patient rate dspace("patientrate")
33.     atom doctor rate dspace("doctorrate")
34.    }
35.
36.  Section System
37.   Service service_0
38.    Process process_0
39.    Resource resource_0
40.    (receiveDoctorRequest,doctor)(sendAppointment,doctor)
41.    (receiveSymptoms,doctor)(sendPrescriptionPickup,doctor)
42.    (receiveMedicineRequest,doctor)(sendMedicine,doctor)(default,patient)
43.
44.  Section Scenario
45.   Scenario scenario_0
46.    ServiceRequest service_0
47.     at time 0, 100, ... using 10 instances
48.
49.  Section Measure
50.   Measure ServiceResponse times using 1 runs of 1000 ServiceRequests
51.
52.  Section Study
53.   Scenario scenario_0
54.    DesignSpace
55.     ( "patientrate" { "6" "7" "8" "9" } )
56.     ( "doctorrate" { "6" "7" "8" "9" } )
```

### C.3.4 A segment of a process with lanes

Table 59 shows iDSL code that is based on the BPMN code of Table 30.

Table 59: The iDSL example of "A segment of a process with lanes"

```
1.  Section Process
2.   ProcessModel process_0
3.    seq {
4.     atom DispatchToApprover load uniformLoad (10 30)
5.     par {
6.      atom PreparePO load 0
7.      par {
8.       atom PreparePOHeader load 10
9.       seq {
10.       atom PreparePOBody load 20
11.       seq {
12.        atom PreparePOFooter load 10
13.       }
14.      }
15.     }
16.    atom ApproveRequest load 0
17.    alt {
18.     atom ApproveRequest load 10+dspace("adminoverhead")
19.     atom ApproveRequest load 30+dspace("adminoverhead")
20.    }
21.  } }
22.
23.  Section Resource
24.   ResourceModel resource_0
25.    decomp PO_resource {
26.     atom webServer rate 1
27.     atom management rate 1
28.     atom administration rate 1
29.   }
30.
```

```
31.  Section System
32.   Service service_0
33.    Process process_0
34.    Resource resource_0
35.    (DispatchToApprover,webServer)
36.    (default,management)
37.    (ApproveRequest,administration)
38.
39.  Section Scenario
40.   Scenario scenario_0
41.    ServiceRequest service_0
42.     at time 0, 1000, ... using 1 instances
43.
44.  Section Measure
45.   Measure CDF of ServiceResponseTimes via advanced PTA model checking
46.
47.  Section Study
48.   Scenario scenario_0
49.    DesignSpace
50.     ( "adminoverhead" { "0" "20" "40" } )
```

### C.3.5 A high-level business process

Table 60 shows iDSL code that is based on the BPMN code of Table 31.

Table 60: The iDSL example of "A high-level business process"

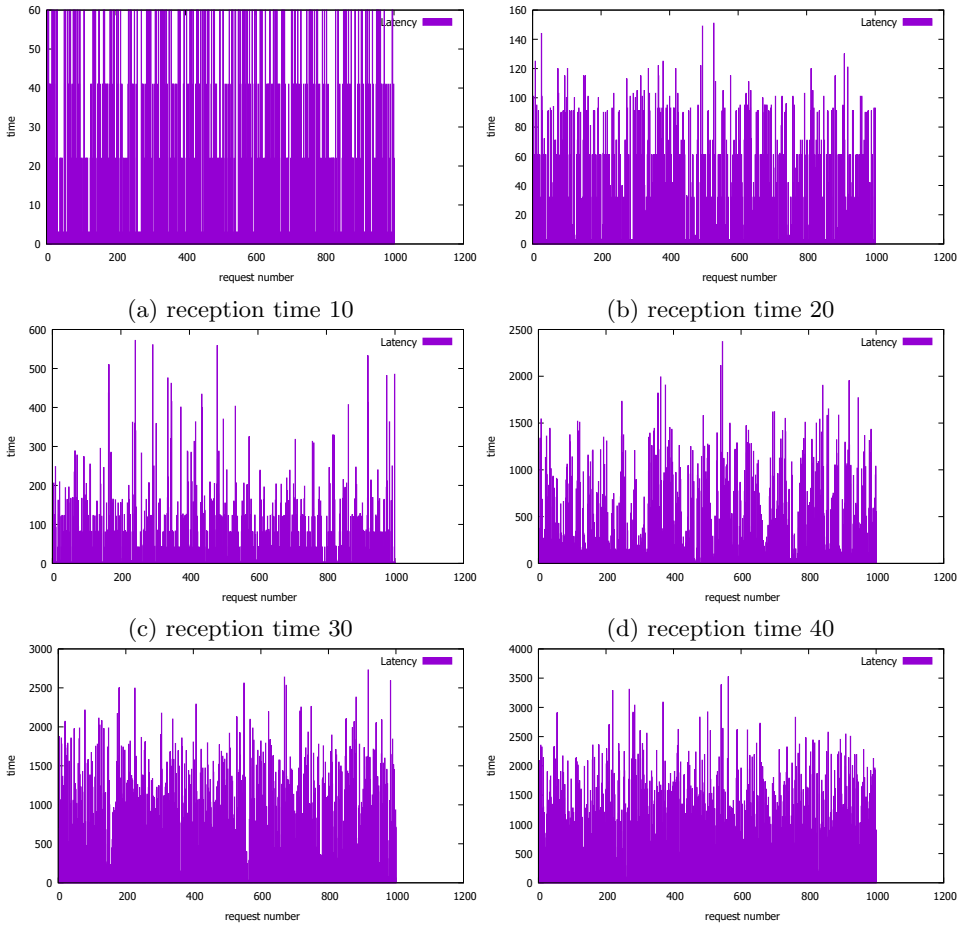
```
1.  Section Process
2.   ProcessModel process_0
3.    seq {
4.     atom pages2and3 load uniformLoad (1 7)
5.     palt { 1 atom abort load 0
6.       1 atom pages4and5and6and7 load uniformLoad (1 10)
7.      alt { atom abort load 0
8.       atom pages8and9and10 load uniformLoad (1 15)
9.    } } }
10.
11. Section Resource
12.  ResourceModel resource_0
13.   decomp res { atom processor rate 1 }
14.
15. Section System
16.  Service service_0
17.   Process process_0
18.   Resource resource_0
19.   (default,processor)
20.
21. Section Scenario
22.  Scenario scenario_0
23.   ServiceRequest service_0
24.    at time 0, 35, ... using 1 instances
25.
26. Section Measure
27.  Measure CDF of ServiceResponseTimes via advanced PTA model
    checking
28.
29. Section Study
30.  Scenario scenario_0
```

# D Additional performance results

This appendix provides performance results in addition to the ones that were given in Section 7 in twofold. Namely, Section D.1 provides results for "A segment of a process with more details" (of Section 7.2), whereas Section D.2 provides results for "A high-level business process" (of Section 7.5).

## D.1 A segment of a process with more details

Figure 22 conveys results in addition to the one that have been presented in Section 7.2. The six subfigures reveal the latencies of the first 1000 instances, from left to right, for designs with reception times ranging from 10 till 60, step 10, respectively.




(a) reception time 10

(b) reception time 20

(c) reception time 30

(d) reception time 40

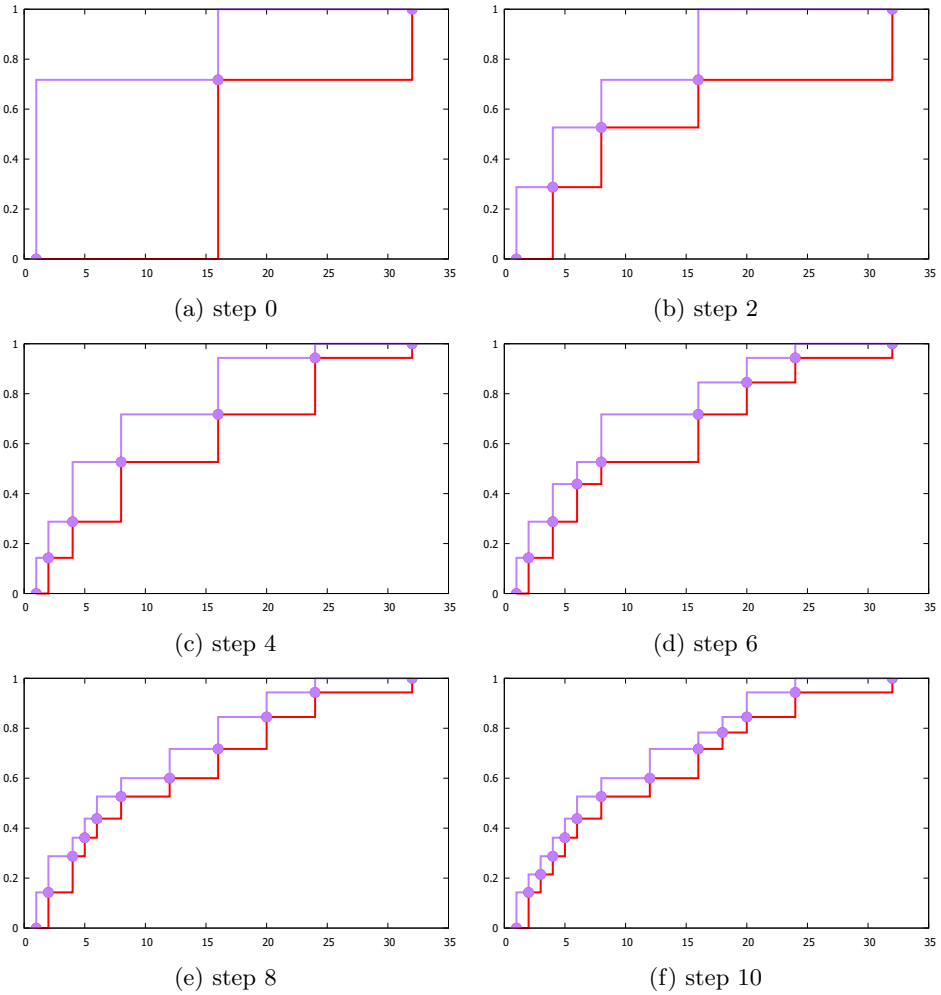
(e) reception time 50

(f) reception time 60

Figure 22: Latency bar charts of "A segment of a process with more details" for different designs.

## D.2 A high-level business process

Figure 23 conveys an additional result in addition to the result of Section 7.5, viz., it shows 14 out of 30 different intermediate steps that are performed to compute the minimum probability for each latency in Figure 21. The subfigures (cf. Figure 23a till 23o) show the different steps of a so-called anytime algorithm that delivers increasingly accurate results as time progresses. Note that the area between the minimum and maximum plot can be used as measure of accuracy, where a smaller area corresponds to more accuracy. It shows that Figure 23o is the same as the bottom plot in Figure 21, i.e., the final result.

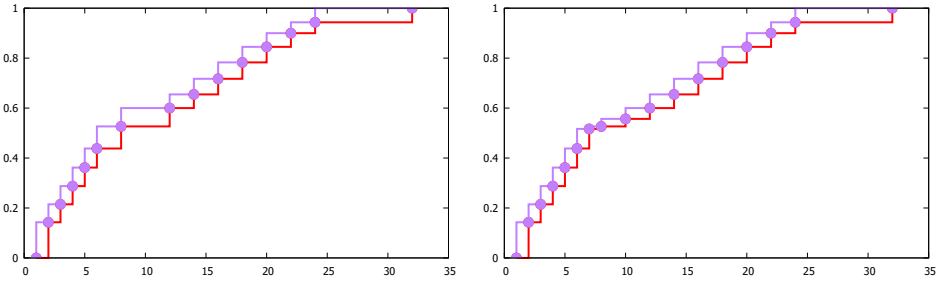


(a) step 0 (b) step 2

(c) step 4 (d) step 6

(e) step 8 (f) step 10

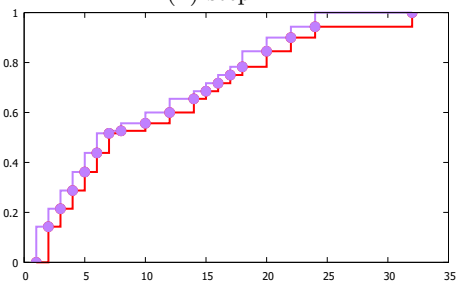
Figure 23: Cumulative latency distributions of the anytime algorithm's intermediate upper bound results of "A high-level business process".



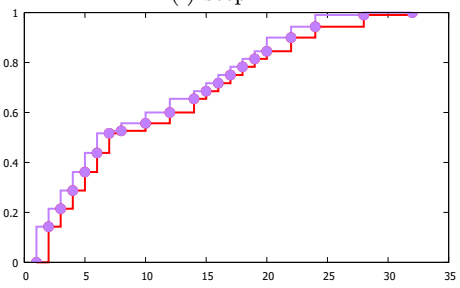
(h) step 12



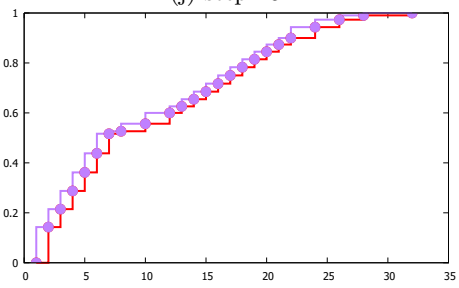
(i) step 14



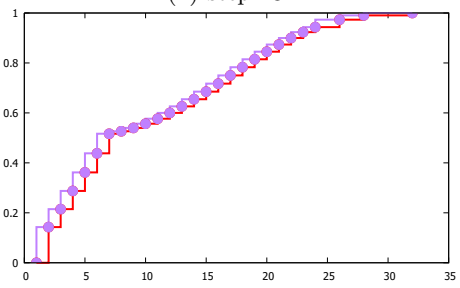
(j) step 16



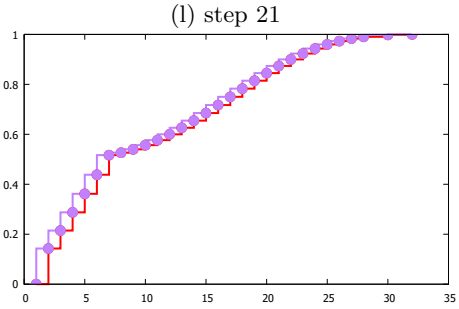
(k) step 18



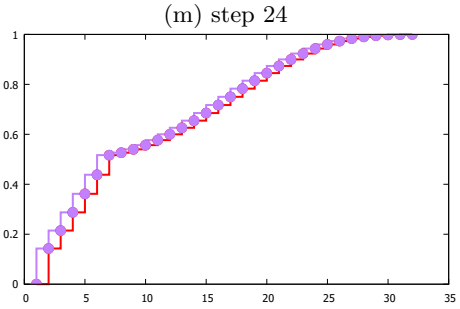
(l) step 21



(m) step 24



(n) step 27



(o) step 30

Figure 23: Cumulative latency distributions of the anytime algorithm's intermediate upper bound results of "A high-level business process" (continued).

# References


[1] EFQM—the official website. http://www.efqm.org/. Last accessed 2016-12-09.

[2] Types of Process Performance Metrics. https://www.heflo.com/blog/business-management/process-performance-metrics/. Last accessed 2016-12-09.

[3] Xtext user guide. http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.html.

[4] (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

[5] Ajmone Marsan, M., Balbo, G., Conte, G., Donatelli, S., and Franceschinis, G. (1995). *Modelling with Generalized Stochastic Petri Nets*. John Wiley and Sons. ISBN: 978-0-471-93059-4.

[6] Akharware, N. and Miee, M. (2005). PIPE2: Platform independent Petri net editor. http://pipe2.sourceforge.net/.

[7] Albrow, M. and King, E. (1990). *Globalization, knowledge and society: readings from international sociology*. Sage.

[8] Alur, R., Courcoubetis, C., Henzinger, T. A., and Ho, P.-H. (1993). Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*, pages 209–229. Springer.

[9] Amit, R. and Schoemaker, P. (1993). Strategic assets and organizational rent. *Strategic Management Journal*, 14(1):33–46. doi: 10.1002/smj.4250140105.

[10] Arlbjørn, J. S. and Haug, A. (2010). *Business process optimization*. Academica.

[11] Baier, C., Haverkort, B., Hermanns, H., and Katoen, J.-P. (2000). Model checking continuous-time markov chains by transient analysis. In *International Conference on Computer Aided Verification*, pages 358–372. Springer.

[12] Baier, C., Haverkort, B.R., Hermanns, H., and Katoen, J.-P. (2003). Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541. doi: 10.1109/tse.2003.1205180.

[13] Baier, C. and Katoen, J.-P. (2008). *Principles of model checking*. MIT press.

[14] Balsamo, S. and Iazeolla, G. (1985). Product-Form Synthesis of Queueing Networks. *IEEE Transactions on Software Engineering*, 11(2):194–199. doi: 10.1109/tse.1985.232194.

[15] Banks, J., Carson, J., Nelson, B., and Nicol, D. (2000). *Discrete-Event System Simulation (3rd Edition)*. Prentice Hall. ISBN: 978-0-130-88702-3.

[16] Basten, A., Hendrix, M., Trcka, N., Somers, L., Geilen, M., Yang, Y., Corporaal, H., Igna, G., Vaandrager, F., Smet, S., Voorhoeve, M., and van der Aalst, W. (2013). Model-Driven Design-Space Exploration for Software-Intensive Embedded Systems. In *Model-Based Design of Adaptive Embedded Systems*, pages 189–244. Springer. doi: 10.1007/978-1-4614-4821-1_7.

[17] Basten, A., Van Benthum, E., Geilen, M., Hendriks, M., Houben, F., Igna, G., Reckers, F., De Smet, S., Somers, L., and Teeselink, E. (2010). Model-driven design-space exploration for embedded systems: the Octopus toolset. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6415 of *Lecture Notes in Computer Science*, pages 90–105. Springer. doi: 10.1007/978-3-642-16558-0_10.

[18] Beauquier, D. (2003). On probabilistic timed automata. *Theoretical Computer Science*, 292(1):65–84. doi: 10.1016/s0304-3975(01)00215-8.

[19] Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer.

[20] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W. (1995). UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer. doi: 10.1007/bfb0020949.

[21] Bengtsson, J. and Yi, W. (2004). Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, pages 87–124. Springer. doi: 10.1007/978-3-540-27755-2_3.

[22] Bollen, P. (2015). The evolution towards a uniform referencing mode in fact-based modeling. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 226–234. Springer.

[23] Bradley, J., Dingle, N., Gilmore, S., and Knottenbelt, W. (2003). Extracting passage times from PEPA models with the HYDRA tool: A case study. In *Proceedings of the* $19^{th}$ *Annual UK Performance Engineering Workshop*, pages 79–90. doi: 10.1109/mascot.2003.1240679.

[24] Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Kretínský, J., Kwiatkowska, M. Z., Parker, D., and Ujma, M. (2014). Verification of Markov Decision Processes Using Learning Algorithms. In *Automated Technology for Verification and Analysis*, volume 8837 of *Lecture Notes in Computer Science*, pages 98–114. Springer. doi: 10.1007/978-3-319-11936-6_8.

[25] Brennan, R., Canning, L., and McDowell, R. (2020). *Business-to-business marketing*. Sage.

[26] Bulychev, P., David, A., Larsen, K. G., Mikučionis, M., Poulsen, D. B., Legay, A., and Wang, Z. (2012). UPPAAL-SMC: Statistical model checking for priced timed automata. *arXiv preprint arXiv:1207.1272*.

[27] Cassandras, C. and Lafortune, S. (1999). Stochastic Timed Automata. In *Introduction to Discrete Event Systems*, volume 11 of *The Kluwer International Series on Discrete Event Dynamic Systems*, pages 317–365. Springer. doi: 10.1007/978-1-4757-4070-7_6.

[28] Chang, J. F. (2016). *Business process management systems: strategy and implementation*. Auerbach Publications.

[29] Chinosi, M. and Trombetta, A. (2012). BPMN: An introduction to the standard. *Computer Standards & Interfaces*, 34(1):124–134.

[30] Clarke, E. (2008). Model Checking – My 27-Year Quest to Overcome the State Explosion Problem. In *Logic for Programming, Artificial Intelligence, and Reasoning*, page 182. Springer. doi: 10.1007/978-3-540-89439-1_13.

[31] Clarke, E., Klieber, W., Novácek, M., and Zuliani, P. (2011a). Model Checking and the State Explosion Problem. In *LASER Summer School*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer. doi: 10.1007/978-3-642-35746-6_1.

[32] Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model checking*. MIT press.

[33] Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P. (2011b). Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer.

[34] Cordys. Cordy product paper: Business Process Management. `https://www.galeos.eu/uploads/Soubory/Cordys/Cordys_bpm_productpaper.pdf`. Last accessed 2018-07-23.

[35] Cross, K. F. and Lynch, R. L. (1988). The "SMART" way to define and sustain success. *National Productivity Review*, 8(1):23–33.

[36] David, A., Larsen, K. G., Legay, A., Mikučionis, M., and Poulsen, D. B. (2015). Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415.

[37] De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

[38] Decker, G., Dijkman, R., Dumas, M., and nuelos, L. G.-B. (2008). Transforming BPMN diagrams into YAWL nets. In *BPM*, pages 386–389. Springer.

[39] Dehnert, C., Junges, S., Katoen, J.-P., and Volk, M. (2016). The probabilistic model checker storm. *arXiv preprint arXiv:1610.08713*.

[40] Dehnert, C., Junges, S., Katoen, J.-P., and Volk, M. (2017). A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*, pages 592–600. Springer.

[41] Department of Computing, Imperial College London. Platform Independent Petri net Editor. `http://pipe2.sourceforge.net/`. Last accessed 2016-12-09.

[42] DiffLQN. Differential Equation Analysis of Layered Queuing Networks - homepage. `http://sysma.imtlucca.it/tools/difflqn/`. Last accessed 2016-12-09.

[43] Dijkman, R. M., Dumas, M., and Ouyang, C. (2008). Semantics and analysis of business process models in BPMN. *Information and Software technology*, 50(12):1281–1294.

[44] Dingle, N., Harrison, P., and Knottenbelt, W. (2004). Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large Markov models. *Journal of Parallel and Distributed Computing*, 64(8):908–920. doi: 10.1016/j.jpdc.2004.03.017.

[45] Dingle, N., Knottenbelt, W., and Suto, T. (2009a). PIPE2: a tool for the performance evaluation of generalised stochastic Petri Nets. *SIGMETRICS Performance Evaluation Review*, 36(4):34–39. doi: 10.1145/1530873.1530881.

[46] Dingle, N. J., Knottenbelt, W. J., and Suto, T. (2009b). PIPE2: a tool for the performance evaluation of generalised stochastic Petri Nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):34–39.

[47] Dittmann, F., Rettberg, A., and Schulte, F. (2005). A Y-Chart Based Tool for Reconfigurable System Design. In *Workshop on Dynamically Reconfigurable Systems*, pages 67–73. VDE Verlag.

[48] Eindhoven University of Technology. Software/Hardware Engineering - High-Speed Simulation of POOSL Models with Rotalumis. `hhttp://poosl.esi.nl/download/`. Last accessed 2016-12-09.

[49] Eisentraut, C., Hermanns, H., and Zhang, L. (2010a). Concurrency and composition in a stochastic world. In *International Conference on Concurrency Theory*, pages 21–39. Springer.

[50] Eisentraut, C., Hermanns, H., and Zhang, L. (2010b). On probabilistic automata in continuous time. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 342–351. IEEE.

[51] Ellson, J., Gansner, E., Koutsofios, L., North, S., and Woodhull, G. (2002). Graphviz—open source graph drawing tools. In *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 483–484. Springer. doi: 10.1007/3-540-45848-4_57.

[52] Embedded Systems Innovation by TNO. POOSL-IDE: Parallel object-oriented specification language. `http://poosl.esi.nl/`. Last accessed 2016-12-09.

[53] Eriksson, H.-E. and Penker, M. (2000). Business modeling with UML. *New York*, pages 1–12.

[54] Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309.

[55] Fishman, G. (1973). *Concepts and methods in discrete event digital simulation*. John Wiley.

[56] Fowler, M. (2004). *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional. ISBN: 978-032-119-368-1.

[57] Franks, G., Al-Omari, T., Woodside, M., Das, O., and Derisavi, S. (2009). Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Transactions on Software Engineering*, 35(2):148–161. doi: 10.1109/tse.2008.74.

[58] Fränzle, M., Hahn, E., Hermanns, H., Wolovick, N., and Zhang, L. (2011). Measurability and safety verification for stochastic hybrid systems. In *Proceedings of the* $14^{th}$ *International Conference on Hybrid Systems: Computation and Control*, pages 43–52. ACM. doi: 10.1145/1967701.1967710.

[59] Gelenbe, E. and Pujolle, G. (1998). *Introduction to Queueing Networks*. John Wiley and Sons, Chichester. ISBN: 978-0-471-96294-6.

[60] German, R., Kelling, C., Zimmermann, A., and Hommel, G. (1995). TimeNET: a toolkit for evaluating non-Markovian stochastic Petri nets. *Performance Evaluation*, 24(1-2):69–87.

[61] Girault, C. and Valk, R. (2003). *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag, Berlin. ISBN: 978-3-540-41217-5.

[62] GNUplot. gnuplot homepage. `http://www.gnuplot.info/`. Last accessed 2016-12-09.

[63] GraphViz. Graphviz - Graph Visualization Software. `http://www.graphviz.org/`. Last accessed 2016-12-09.

[64] Hahn, E., Hartmanns, A., Hermanns, H., and Katoen, J.-P. (2012). A Compositional Modelling and Analysis Framework for Stochastic Hybrid Systems. *Formal Methods in System Design*, 43(2):191–232. doi: 10.1007/s10703-012-0167-z.

[65] Hammer, M. and Champy, J. (2009). *Reengineering the Corporation: Manifesto for Business Revolution, A*. Zondervan.

[66] Hansson, H. and Jonsson, B. (1994). A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535. doi: 10.1007/bf01211866.

[67] Harrison, P. (1993). Response Time Distributions in Queueing Network Models. In *Performance/SIGMETRICS Tutorials*, volume 729 of *Lecture Notes in Computer Science*, pages 147–164. Springer. doi: 10.1007/bfb0013852.

[68] Hartmanns, A. and Hermanns, H. (2009). A Modest approach to checking probabilistic timed automata. In *Quantitative Evaluation of Systems, 2009. QEST'09. Sixth International Conference on the*, pages 187–196. IEEE.

[69] Hartmanns, A. and Hermanns, H. (2014a). The modest toolset: an integrated environment for quantitative modelling and verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 593–598. Springer.

[70] Hartmanns, A. and Hermanns, H. (2014b). The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 593–598. Springer. doi: 10.1007/978-3-642-54862-8_51.

[71] Hartmanns, A. and Hermanns, H. (2015). In the quantitative automata zoo. *Science of Computer Programming*, 112:3–23.

[72] Haverkort, B.R. (1998). *Performance of computer communication systems - a model-based approach*. Wiley. ISBN: 978-0-471-97228-0.

[73] Henzinger, T. A. (2000). The theory of hybrid automata. In *Verification of Digital and Hybrid Systems*, pages 265–292. Springer.

[74] Hermanns, H. (2002). Interactive markov chains. In *Interactive Markov Chains*, pages 57–88. Springer.

[75] Hu, J., Lygeros, J., and Sastry, S. (2000). Towards a Theory of Stochastic Hybrid Systems. In *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 160–173. Springer. doi: 10.1007/3-540-46430-1_16.

[76] Igna, G. and Vaandrager, F. (2010). Verification of printer datapaths using timed automata. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *Lecture Notes in Computer Science*, pages 412–423. Springer. doi: 10.1007/978-3-642-16561-0_38.

[77] Jensen, K., Kristensen, L., and Wells, L. (2007). Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254. doi: 10.1007/s10009-007-0038-x.

[78] Jeston, J. (2014). *Business process management*. Routledge.

[79] Jesus, A. D., Liefooghe, A., Derbel, B., and Paquete, L. (2020). Algorithm selection of anytime algorithms. In *Proceedings of the 2020 genetic and evolutionary computation conference*, pages 850–858.

[80] Johansson, H. J. (1993). *Business process reengineering: Breakpoint strategies for market dominance*. John Wiley & Sons.

[81] Juiz, C. and Puigjaner, R. (2000). Queueing Analysis of Pools in Soft Real-Time Systems. In *Computer Performance Evaluation / TOOLS*, volume 1786 of *Lecture Notes in Computer Science*, pages 56–70. Springer. doi: 10.1007/3-540-46429-8_5.

[82] Kaplan, R. S., Kaplan, R. S., Norton, D. P., and Norton, D. P. (1996). *The balanced scorecard: translating strategy into action*. Harvard Business Press.

[83] Kaplan, R. S. and Norton, D. P. (1995). Putting the Balanced Scorecard. *Performance measurement, management, and appraisal sourcebook*, page 66.

[84] Kaplan, R. S. and Norton, D. P. (1996). Using the balanced scorecard as a strategic management system.

[85] Kaplan, R. S., Robert, N. P. D. K. S., Davenport, T. H., Kaplan, R. S., and Norton, D. P. (2001). *The strategy-focused organization: How balanced scorecard companies thrive in the new business environment*. Harvard Business Press.

[86] Katoen, J.-P., Zapreev, I. S., Hahn, E. M., Hermanns, H., and Jansen, D. N. (2011). The ins and outs of the probabilistic model checker MRMC. *Performance evaluation*, 68(2):90–104.

[87] Kienhuis, B., Deprettere, E., van der Wolf, P., and Vissers, K. (2002). A methodology to design programmable embedded systems. In *Embedded Processor Design Challenges*, volume 2268 of *Lecture Notes in Computer Science*, pages 18–37. Springer. doi: 10.1007/3-540-45874-3_2.

[88] Kirchmer, M. (2017). *High performance through business process management*. Springer.

[89] Koziolek, H. and Reussner, R. (2008). A Model Transformation from the Palladio Component Model to Layered Queueing Networks. In *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *Lecture Notes in Computer Science*, pages 58–78. Springer. doi: 10.1007/978-3-540-69814-2_6.

[90] Kozlenkova, I., Hult, T., Lund, D., A. Mena, J., and Kekec, P. (2015). The Role of Marketing Channels in Supply Chain Management. 91.

[91] Krajewski, L. J., Ritzman, L. P., and Malhotra, M. K. (2010). *Operations management: processes and supply chains*. Pearson Upper Saddle River, New Jersey.

[92] Krishna, C. M. (2001). Real-Time Systems. *Wiley Encyclopedia of Electrical and Electronics Engineering*.

[93] Kueng, P. (2000). Process performance measurement system: a tool to support process-based organizations. *Total Quality Management*, 11(1):67–85.

[94] Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: verification of probabilistic real-time systems. In *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer. doi: 10.1007/978-3-642-22110-1_47.

[95] Kwiatkowska, M., Norman, G., Segala, R., and Sproston, J. (2002). Automatic verification of real-time systems with discrete probability distributions. *Electronic Notes in Theoretical Computer Science*, 282(1):101–150. doi: 10.1016/s0304-3975(01)00046-9.

[96] Larsen, K. G., Pettersson, P., and Yi, W. (1997). UPPAAL in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152.

[97] Legay, A. and Delahaye, B. (2010). Statistical Model Checking : An Overview. *Computing Research Repository*, abs/1005.1327. doi: 10.1007/978-3-642-16612-9_11.

[98] Lijnse, B. and Plasmeijer, R. (2009). iTasks 2: iTasks for End-users. In *IFL*, pages 36–54. Springer.

[99] Liu, F., Narayanan, A., and Bai, Q. (2000). Real-time systems.

[100] Lucking-Reiley, D. and Spulber, D. F. (2001). Business-to-business electronic commerce. *Journal of Economic Perspectives*, 15(1):55–68.

[101] Marsan, M. A., Balbo, G., Conte, G., Donatelli, S., and Franceschinis, G. (1994). *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, Inc.

[102] Metropolis, N. and Ulam, S. (1949). The Monte carlo method. *Journal of the American statistical association*, 44(247):335–341. doi: 10.1080/01621459.1949.10483310.

[103] Modest. The Modest Toolset. `http://www.modestchecker.net/`. Last accessed 2016-12-09.

[104] Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

[105] Nagurney, A. (2006). *Supply chain network economics: dynamics of prices, flows and profits*. Edward Elgar Publishing.

[106] Neely, A., Mills, J., Platts, K., Richards, H., Gregory, M., Bourne, M., and Kennerley, M. Performance measurement system design: developing and testing a process-based approach. *International journal of operations & production management*, 20.

[107] Newbert, S. L. (2008). Value, rareness, competitive advantage, and performance: a conceptual-level empirical investigation of the resource-based view of the firm. *Strategic management journal*, 29(7):745–768. doi: 10.1002/smj.686.

[108] OMG. Business Process Model and Notation (BPMN). `http://www.omg.org/spec/BPMN/2.0`. Last accessed 2016-12-09.

[109] O'sullivan, A. and Sheffrin, S. M. (2003). Economics: Principles in action.

[110] O'Sullivan, A. and Sheffrin, S. M. (2005). *Economics: Principles in Action*. Prentice Hall. ISBN: 978-0131334830.

[111] Ouyang, C., Dumas, M., Ter Hofstede, A. H., and Van der Aalst, W. M. (2006). From BPMN process models to BPEL web services. In *Web Services, 2006. ICWS'06. International Conference on*, pages 285–292. IEEE.

[112] Pech, V., Shatalin, A., and Voelter, M. (2013). Jetbrains mps as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 165–168.

[113] Pelánek, R. (2008). Fighting state space explosion: Review and evaluation. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 37–52. Springer.

[114] Plasmeijer, R., Achten, P., and Koopman, P. (2007). iTasks: executable specifications of interactive work flow systems for the web. *SIGPLAN Notices*, 42(9):141–152. doi: 10.1145/1291220.1291174.

[115] PMNL. Petri Net Markup Language. `http://www.pnml.org/`. Last accessed 2016-12-09.

[116] Porter, M. E. (1985). Competitive advantage: creating and sustaining superior performance. 1985. *New York: FreePress*, 43:214.

[117] PRISM. Probabilistic Symbolic Model Checker. `http://www.prismmodelchecker.org/`. Last accessed 2016-12-09.

[118] Privault, N. (2018). Discrete-time Markov chains. In *Understanding Markov Chains*, pages 89–113. Springer.

[119] Puterman, M. L. (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[120] Racine, J. (2006). GNUplot 4.0: a portable interactive plotting utility. *Journal of Applied Econometrics*, 21(1):133–141. doi: 10.1002/jae.885.

[121] Rantzer, A. (2003). CPN Tools for Editing, Simulating and Analysing Coloured Petri Nets. In *Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003*, volume 2679, pages 450–462. doi: 10.1007/3-540-44919-1_28.

[122] Razavian, M., Vanderfeesten, I., and Turetken, O. (2017). Towards a solution space for bpm issues based on debiasing techniques. In *International Conference on Business Process Management*, pages 384–390. Springer.

[123] Reisig, W. (2013). *Understanding Petri Nets*. Springer. ISBN: 978-3-642-33277-7.

[124] Roman, J. H. (2001). The Web as an API. `https://www.osti.gov/servlets/purl/975922`.

[125] Rowe, A. J. (1989). *Strategic management: A methodological approach*. Addison Wesley publishing company.

[126] Rummler, G. and Brache, A. (1995). Improving performance: Managing the white space in organizations.

[127] Score, K. (1996). Using the Right Metrics to Drive World-Class Performance. *Purchasing and supply chain management. Productivity Inc.*

[128] Sen, K., Viswanathan, M., and Agha, G. (2005). Vesta: A statistical model-checker and analyzer for probabilistic systems. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 251–252. IEEE.

[129] Sericola, B. (2013). Discrete-Time Markov Chains. *Markov Chains: Theory, Algorithms and Applications*, pages 1–87.

[130] Silver, B. and Richard, B. (2009). *BPMN method and style*, volume 2. Cody-Cassidy Press Aptos. ISBN: 978-098-236-811-4.

[131] Sirmon, D. G., Hitt, M. A., and Ireland, R. D. (2007). Managing firm resources in dynamic environments to create value: Looking inside the black box. *Academy of management review*, 32(1):273–292. doi: 10.5465/amr.2007.23466005.

[132] Smith, A. (1776). Of the division of labour. *Classics of organization theory*, pages 40–45.

[133] Stuijk, S., Geilen, M., and Basten, T. (2006). SDF3: SDF For Free. In $6^{th}$ *International Conference on Application of Concurrency to System Design*, pages 276–278. IEEE Computer Society. doi: 10.1109/acsd.2006.23.

[134] Theelen, B., Geilen, M., Basten, A., Voeten, J., Gheorghita, S., and Stuijk, S. (2006). A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In $4^{th}$ *ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 185–194. IEEE Computer Society. doi: 10.1109/memcod.2006.1695924.

[135] Theelen, B., Voeten, J., and Kramer, R. (2003). Performance modelling of a network processor using POOSL. *Computer Networks*, 41(5):667–684. doi: 10.1016/s1389-1286(02)00455-3.

[136] Tretmans, J. (1996). Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer networks and ISDN systems*, 29(1):49–79. doi: 10.1016/s0169-7552(96)00017-7.

[137] Tribastone, M. (2013). A Fluid Model for Layered Queueing Networks. *IEEE Trans. Software Eng.*, 39(6):744–756. doi: 10.1109/tse.2012.66.

[138] Upadhaya, B., Munir, R., and Blount, Y. (2014). Association between performance measurement systems and organisational effectiveness. *International Journal of Operations & Production Management*, 34(7):853–875.

[139] Uppaal. Uppsala Aalborg model checker. `http://www.uppaal.org/`. Last accessed 2016-12-09.

[140] Valmari, A. (1996). The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer.

[141] van den Berg, F. The official iDSL homepage. `http://www-i-dsl.org`. Last accessed 2016-12-09.

[142] van den Berg, F. (2010). Strongly typed BPMN. Master's thesis, Radboud University Nijmegen.

[143] van den Berg, F., Camra, V., Hendriks, M., Geilen, M. C., Hnetynka, P., Manteca, F., Sanchez, P., Bures, T., and Basten, A. T. (2020). QRML: A Component Language and Toolset for Quality and Resource Management. In *Proceedings of FDL 2020, Kiel, Germany*. IEEE.

[144] van den Berg, F., Garousi, V., Tekinerdogan, B., and Haverkort, B. R. (2018a). Designing cyber-physical systems with adsl: A domain-specific language and tool support. In *2018 13th Annual Conference on System of Systems Engineering (SoSE)*, pages 225–232. IEEE.

[145] van den Berg, F., Haverkort, B.R., and Hooman, J. (2015a). Efficiently Computing Latency Distributions by Combined Performance Evaluation Techniques. In *Proceedings of the* $9^{th}$ *EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, pages 158–163. ICST. doi: 10.4108/eai.14-12-2015.2262725.

[146] van den Berg, F., Hooman, J., Hartmanns, A., Haverkort, B.R., and Remke, A. (2015b). Computing Response Time Distributions Using Iterative Probabilistic Model Checking. In *Computer Performance Engineering*, volume 9272 of *Lecture Notes in Computer Science*, pages 208–224. Springer. doi: 10.1007/978-3-319-23267-6_14.

[147] van den Berg, F., Hooman, J., and Haverkort, B. R. (2018b). A Domain-Specific Language and Toolchain for Performance Evaluation

Based on Measurements. In *International Conference on Measurement, Modelling and Evaluation of Computing Systems*, pages 295–301. Springer.

[148] van den Berg, F., Postema, B., and Haverkort, B.R. (2016). Evaluating load balancing policies for performance and energy-efficiency. In *Quantitative Aspects of Programming Languages and Systems*, volume 227 of *Electronic Proceedings in Theoretical Computer Science*, pages 98–117. doi: 10.4204/eptcs.227.7.

[149] Van Den Berg, F., Remke, A., and Haverkort, B. R. (2014). A domain specific language for performance evaluation of medical imaging systems. In *OASIcs-OpenAccess Series in Informatics*, volume 36. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[150] van den Berg, F., Remke, A., and Haverkort, B.R. (2014). A Domain Specific Language for Performance Evaluation of Medical Imaging Systems. In *5th Workshop on Medical Cyber-Physical Systems*, volume 36 of *OpenAccess Series in Informatics*, pages 80–93. Schloss Dagstuhl. doi: 10.4230/OASIcs.MCPS.2014.80.

[151] van den Berg, F., Remke, A., and Haverkort, B.R. (2015c). iDSL: Automated Performance Prediction and Analysis of Medical Imaging Systems. In *Computer Performance Engineering*, volume 9272, pages 227–242. Springer. doi: 10.1007/978-3-319-23267-6_15.

[152] van den Berg, F. G. B. (2017). *Automated performance evaluation of service-oriented systems*. University of Twente.

[153] Van der Aalst, W. M. (1998). The application of Petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66.

[154] Van der Aalst, W. M. and ter Hofstede, A. H. (2005). YAWL: yet another workflow language. *Information systems*, 30(4):245–275. doi: 10.1016/j.is.2004.02.002 .

[155] Vergidis, K., Tiwari, A., and Majeed, B. (2008). Business process analysis and optimization: Beyond reengineering. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(1):69–82.

[156] Voelter, M. and Pech, V. (2012). Language modularity with the mps language workbench. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1449–1450. IEEE.

[157] Voeten, J. P. (2002). Performance evaluation with temporal rewards. *Performance Evaluation*, 50(2):189–218. doi: 10.1016/s0166-5316(02)00105-0.

[158] Von Halle, B. (2001). *Business rules applied: building better systems using the business rules approach*. Wiley Publishing.

[159] Waizmann, T. and Tribastone, M. (2016). DiffLQN: Differential Equation Analysis of Layered Queuing Networks. In *International Conference on Performance Engineering*, pages 63–68. ACM. doi: 10.1145/2859889.2859896.

[160] Weisstein, E. W. (2002). Normal distribution. *https://mathworld. wolfram. com/*.

[161] Weske, M. (2007). Evolution of enterprise systems architectures. *Business Process Management: Concepts, Languages, Architectures*, pages 25–69.

[162] Weske, M. (2012). Business process management architectures. In *Business Process Management*, pages 333–371. Springer.

[163] White, S. A. (2004). Introduction to BPMN. *IBM Cooperation*, 2(0):0.

[164] White, S. A. (2008). *BPMN modeling and reference guide: understanding and using BPMN*. Future Strategies Inc.

[165] Wilcox, P. A. and Gurău, C. (2003). Business modelling with UML: the implementation of CRM systems for online retailing. *Journal of Retailing and Consumer Services*, 10(3):181–191.

[166] Wise, R. and Morrison, D. (2000). Beyond the exchange–the future of B2B. *Harvard business review*, 78(6):86–96.

[167] Wolf, M. (2014). Shaping globalization. *Finance and development*, 51(3):22–25.

[168] Wolfinbarger, M. and Gilly, M. C. (2001). Shopping online for freedom, control, and fun. *California management review*, 43(2):34–55.

[169] Younes, H. L. (2005). Verification and planning for stochastic processes with asynchronous events. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE.

[170] Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73–73.

[171] Zimmermann, A., Freiheit, J., German, R., and Hommel, G. (2000). Petri net modelling and performability evaluation with TimeNET 3.0. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 188–202. Springer.

[172] Zott, C. and Amit, R. (2010). Business model design: an activity system perspective. *Long range planning*, 43(2-3):216–226.