A SEMANTIC END-TO-END PROCESS CONSTRAINT MODELING FRAMEWORK

by

SHASHA LIU

(Under the Direction of Krys J. Kochut)

ABSTRACT

Since the introduction of Web services and the service-oriented architecture, they have been widely adopted in industry and academic settings. Web services have the advantages of standardization of service interfaces and facilitating system integration and cross-platform interoperability, which are extensively utilized by business process systems and scientific workflow systems. In the research area of model-driven software engineering, even the simplest software system needs to address the major issue of requirements and constraints during the whole lifecycle. Web service-based business processes and scientific workflows are no exception. Process constraint modeling and verification, focusing on how to enforce the conformity of process constraints throughout its lifecycle, including design, deployment and runtime execution, remains a big challenge, especially when such constraints are considered in the composite Web services and workflow systems. In this dissertation, we propose a semantic end-to-end Process Constraint Framework (PCF) to address this issue. The framework includes a process constraint model, consisting of a process constraint ontology (ProContO) and a process constraint language (PCL). In addition to constraint modeling at design time covered by the representation model, both static and dynamic verification in process constraint’s lifecycle are addressed in PCF. While the static verification of process constraints concentrates on syntactic, semantic and service specification
verifications during design and deployment phases, dynamic verification focuses on the runtime process execution with the help of the underlying monitoring module. Moreover, PCF supports both the definition and implementation of user-defined exception handling methods, as exceptions are ubiquitous in the distributed environment. Three motivating scenarios: (i) prevention of nepotism in job interviews, (ii) safe distance guarantee in tornado emergency reaction process, and (iii) selection of service network location in a glycomics scientific workflow process, are used to illustrate how ontology and semantic technologies are utilized to model different kinds of process constraints in their whole lifecycle.

INDEX WORDS: Workflow processes, Web services, Process constraint representation, Process constraint ontology, Process constraint language, Process constraint verification
A SEMANTIC END-TO-END PROCESS CONSTRAINT MODELING FRAMEWORK

by

SHASHA LIU
B.E., Huazhong University of Science and Technology, China, 2008

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2015
A SEMANTIC END-TO-END PROCESS CONSTRAINT MODELING FRAMEWORK

by

SHASHA LIU

Major Professor: Krys J. Kochut
Committee: John A. Miller
            William S. York

Electronic Version Approved:

Julie Coffield
Interim Dean of the Graduate School
The University of Georgia
May 2015
DEDICATION

To My Beloved Family
ACKNOWLEDGEMENTS

First, I would like to acknowledge the contribution of my advisor, Prof. Krys J. Kochut in making this work possible. I would like to express my grateful and sincere appreciation to him for his guidance, advice, encouragement, understanding and patience. I have learned numerous things from him, from paper writing, presentation skills and experimental methods to approaches of conducting research. From the bottom of my heart, I would like to thank him for making my graduate study an enjoyable experience.

I would like to express my gratitude to my committee members: Dr. John A. Miller and Dr. William S. York for their input, valuable discussions and accessibility. I would also like to mention my colleagues in the lab, Matthew Eavenson, Manual Correa and Anuj Shetye for their inspiration in the discussions and some much needed humor and entertainment.

Last but not least, I would like to thank my whole family who are always standing by me and providing their silent but strong support, their consistent encouragement and understanding.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGEMENTS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>PAGE</td>
</tr>
<tr>
<td></td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>PAGE</td>
</tr>
<tr>
<td></td>
<td>xi</td>
</tr>
</tbody>
</table>

## CHAPTER

1. **INTRODUCTION** ................................................................................................................... 1
   1.1. **OVERALL INTRODUCTION** ............................................................................................. 1
   1.2. **SUMMARY OF MAJOR CONTRIBUTIONS** ......................................................................... 6
   1.3. **STRUCTURAL OUTLINE OF THE DISSERTATION** .............................................................. 7

2. **BACKGROUND** ..................................................................................................................... 9
   2.1. **REQUIREMENTS AND CONSTRAINTS IN SOFTWARE ENGINEERING** ......................... 9
   2.2. **WORKFLOWS** ............................................................................................................... 11
   2.3. **SERVICE-ORIENTED ARCHITECTURE (SOA)** ................................................................. 14
   2.4. **PROCESS DEFINITION STANDARDS** ............................................................................. 15
   2.5. **ONTOLOGY AND OWL** ................................................................................................. 21
   2.6. **OBJECT CONSTRAINT LANGUAGE** .............................................................................. 23
   2.7. **MODEL DRIVEN ENGINEERING (MDE)** ..................................................................... 25

3. **MOTIVATING EXAMPLES** .................................................................................................... 27
   3.1. **PREVENTION OF NEPOTISM IN JOB INTERVIEWS** ................................................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2. TORNADO EMERGENCY REACTION WORKFLOW PROCESS</td>
<td>28</td>
</tr>
<tr>
<td>3.3. GLYCOMICS SCIENTIFIC WORKFLOW</td>
<td>29</td>
</tr>
<tr>
<td>4. LITERATURE REVIEW</td>
<td>31</td>
</tr>
<tr>
<td>4.1. CONSTRAINTS MODELING AND REPRESENTATION</td>
<td>31</td>
</tr>
<tr>
<td>4.2. PROCESS CONSTRAINT VERIFICATION</td>
<td>33</td>
</tr>
<tr>
<td>4.3. EXCEPTION HANDLING</td>
<td>35</td>
</tr>
<tr>
<td>5. PROCESS CONSTRAINT REPRESENTATION MODEL</td>
<td>38</td>
</tr>
<tr>
<td>5.1. PROCESS CONSTRAINT ONTOLOGY (ProContO)</td>
<td>40</td>
</tr>
<tr>
<td>5.2. PROCESS CONSTRAINT LANGUAGE (PCL)</td>
<td>46</td>
</tr>
<tr>
<td>6. PROCESS CONSTRAINT VERIFICATION</td>
<td>51</td>
</tr>
<tr>
<td>6.1. STATIC VERIFICATION</td>
<td>52</td>
</tr>
<tr>
<td>6.2. DYNAMIC VERIFICATION</td>
<td>56</td>
</tr>
<tr>
<td>7. PROTOTYPE IMPLEMENTATION</td>
<td>61</td>
</tr>
<tr>
<td>7.1. SERVICE COMPONENT ARCHITECTURE (SCA) AND ENTERPRISE INTEGRATION PATTERNS</td>
<td>61</td>
</tr>
<tr>
<td>7.2. ARCHITECTURE OF THE PCF FRAMEWORK</td>
<td>69</td>
</tr>
<tr>
<td>7.3. PROCESSES AND CONSTRAINT IMPLEMENTATION</td>
<td>76</td>
</tr>
<tr>
<td>8. EVALUATION</td>
<td>83</td>
</tr>
<tr>
<td>8.1. QUALITY EVALUATION</td>
<td>83</td>
</tr>
<tr>
<td>8.2. PERFORMANCE EVALUATION</td>
<td>87</td>
</tr>
<tr>
<td>9. CONCLUSIONS AND FUTURE WORK</td>
<td>91</td>
</tr>
<tr>
<td>9.1. SUMMARY AND DISCUSSION</td>
<td>91</td>
</tr>
<tr>
<td>9.2. ORIGINAL CONTRIBUTIONS</td>
<td>92</td>
</tr>
</tbody>
</table>
9.3. FUTURE WORK ........................................................................................................... 93

BIBLIOGRAPHY ................................................................................................................ 95

APPENDICES ..................................................................................................................... 107
   A. PROCESS CONSTRAINT ONTOLOGY (ProContO) IN OWL ........................................ 107
   B. NEPOTISM ONTOLOGY ............................................................................................ 110
   C. PROCESS CONSTRAINT LANGUAGE (PCL) PROCESSER ........................................ 125
   D. PCF FRAMEWORK PROJECT .................................................................................... 132
   E. PCF FRAMEWORK DEVELOPMENT ENVIRONMENT SETUP .................................. 133
   F. SIMPLE SCA TUTORIAL ............................................................................................ 135
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Workflow management lifecycle and business process management lifecycle [34]</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Scientific workflow lifecycle [40]</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Process constraint lifecycle corresponding to the process management lifecycle</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>BPMN core components: flow objects</td>
<td>19</td>
</tr>
<tr>
<td>2.5</td>
<td>BPMN core components: connecting objects</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>BPMN core components: swimlanes</td>
<td>20</td>
</tr>
<tr>
<td>2.7</td>
<td>BPMN core components: artifacts</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>Petri nets: an example</td>
<td>21</td>
</tr>
<tr>
<td>2.9</td>
<td>Ontological representation of Parent and Child</td>
<td>23</td>
</tr>
<tr>
<td>2.10</td>
<td>UML model example: Person and Company</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>A Simple Job Interview Workflow Process</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Tornado Emergency Reaction Workflow Process</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Glycomics Scientific Workflow</td>
<td>30</td>
</tr>
<tr>
<td>5.1</td>
<td>UML model for PCL: promoting constraints as first-class citizens</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>Backbone of the Process Constraint Ontology (ProContO)</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>ProcessElement Class and Its Basic Components</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>Examples of ConstraintAttribute Class and The Relationship between ProcessElement and ConstraintAttribute</td>
<td>43</td>
</tr>
</tbody>
</table>
Figure 5.5. Examples of ConstraintOperation class and the relationship between ConstraintOperation and ConstraintAttribute .............................................................. 44

Figure 5.6. Ontology Definitions for The Geospatial Constraint in Tornado Emergency Reaction Workflow Process with Exception Modeling .......................................................... 45

Figure 6.1. End-to-end process constraint verification ........................................................................................................... 52

Figure 6.2. Context-aware monitoring module .......................................................................................................................... 57

Figure 7.1. Switchyard HelloWorld example .......................................................................................................................... 63

Figure 7.2. Enterprise integration - wiretap pattern [110] ........................................................................................................... 66

Figure 7.3. Enterprise integration - Message broker pattern [110] ............................................................................................ 66

Figure 7.4. Enterprise integration - Content-based pattern [110] ............................................................................................. 67

Figure 7.5. Service integration template: before and after ........................................................................................................ 69

Figure 7.6. Process constraint framework prototype architecture – Layer cake model ......................................................... 70

Figure 7.7. Process constraint framework prototype architecture – Example illustration ..................................................... 71

Figure 7.8: Nepotism ontology .................................................................................................................................................. 73

Figure 7.9. Job interview process in BPMN representation ...................................................................................................... 77

Figure 7.10. Job interview process implementation in SwitchYard .......................................................................................... 78

Figure 7.11. BPEL process for emergency reaction workflow .................................................................................................. 79

Figure 7.12. Emergency Reaction Workflow implementation in SwitchYard ........................................................................ 80

Figure 7.13. Glycomics Workflow in SwitchYard ...................................................................................................................... 81

Figure 8.1. An end-to-end process constraint framework covering the whole lifecycle of process constraints .......................................................... 86

Figure 8.2. An end-to-end process constraint framework covering the whole lifecycle of process constraints .......................................................... 86
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>SWRL rule example: inferring a Person is a Parent</td>
<td>23</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>OCL example: invariants</td>
<td>24</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>OCL example: pre- and post-conditions</td>
<td>25</td>
</tr>
<tr>
<td>Table 2.4</td>
<td>OCL example: navigating associations</td>
<td>25</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>PCL Expressions</td>
<td>47</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>PCL Operators</td>
<td>47</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>PCL Constraint Declaration</td>
<td>48</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>GeoLocationProximity constraint in tornado emergency reaction workflow process</td>
<td>49</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>PCL syntax verification example -- missing definition of variable &quot;a&quot;</td>
<td>53</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>PCL semantic verification example -- no concept &quot;location&quot; declaration in ProContO</td>
<td>54</td>
</tr>
<tr>
<td>Table 6.3</td>
<td>PCL semantic verification example -- non-navigable from &quot;geoLocation&quot; to &quot;location&quot; in ProContO</td>
<td>54</td>
</tr>
<tr>
<td>Table 6.4</td>
<td>Service specification verification example – incomplete parameter list</td>
<td>55</td>
</tr>
<tr>
<td>Table 6.5</td>
<td>Constraint of nepotism restriction in interview process</td>
<td>59</td>
</tr>
<tr>
<td>Table 6.6</td>
<td>SWRL rules for family relationship in ProContO</td>
<td>59</td>
</tr>
<tr>
<td>Table 7.1</td>
<td>Service Integration Template for PCF framework</td>
<td>68</td>
</tr>
<tr>
<td>Table 7.2</td>
<td>Nepotism SWRL rules</td>
<td>73</td>
</tr>
<tr>
<td>Table 7.3</td>
<td>PCL parser implementation snippet in Scala</td>
<td>74</td>
</tr>
</tbody>
</table>
Table 7.4: Camel Routing in JobInterview service task ................................................................. 78
Table 7.5. Glycomics workflow in Camel .................................................................................. 82
Table 7.6. Camel route for data processor service ....................................................................... 82
Table 8.1: Process, process elements and their constraints. ....................................................... 84
Table 8.2. Response time for services and constraint operations (unit: second) ...................... 89
CHAPTER

1. INTRODUCTION

1.1. OVERALL INTRODUCTION

1.1.1 REQUIREMENTS AND CONSTRAINTS

Ever since the advent of software engineering as a discipline in computer science, requirements and constraints have been vital to various types of software system development, ranging from standalone programs to large complex systems based on workflow processes, Web services, and the new computing infrastructures, e.g., cloud computing. While the requirements include all features that a software product must possess, constraints describe real-world limits or boundaries around them [1]. Requirements and constraints usually go side-by-side and address the system under study (SUS) from different perspectives and granularities. Typically, when a new requirement is identified, one or more corresponding constraints are also provided to explicitly specify what limitations and boundaries should be observed and met. For example, a general requirement could be stated as “the system should quickly load a page with a list of the manager’s approval items”, and a corresponding constraint could be “response time must be less than 1 second”.

Requirements can be further subdivided into three groups: functional, non-functional and domain-specific requirements. While functional requirements focus on what the software should be able to do, the other two describe under what conditions the software should achieve it [2]. Our research work focuses on non-functional and domain-specific requirements because they are the
major driving force in engineering high quality software, e.g., offering higher availability and scalability, better user-friendliness, more secure communication, and many others.

Instead of working on the non-functional and domain-specific constraints of the general software engineering systems at a high level, we narrow down the research scope and concentrate on the constraints of workflow process systems and composed Web services. Over the span of the last few decades, workflow systems have been successfully adopted in many industries, ranging from banking to manufacturing. They provide rich capabilities of process-defined task execution, fault tolerance and compensation to facilitate the collaboration among various participants within and across enterprises. Workflow process systems have also achieved wide acceptance in helping scientists in data processing and in conducting scientific experiments. Recently, a number of workflow management systems have been developed for scientific applications [3, 4]. We believe that workflow process systems will gain even wider application and acceptance by collaborating with the nowadays prevailing Service Oriented Architecture (SOA) as the underlying implementation, deployment and execution infrastructure, because the SOA paradigm enables process applications to be more agile and flexible and facilitate system level integration and interoperability.

Much research has been published on constraints within workflow processes and composed Web services, frequently expressed and modeled using the Unified Modeling Language (UML) [5] and Object Constraint Language (OCL) [6]. Even though UML originated from object-oriented methodologies and specializes in modeling software system from the object-oriented perspective, it has been widely applied in modeling Web service compositions [7] and workflow systems [8]. As a supplement to UML, OCL is both declarative and navigational. It has become a key component of many model-driven engineering (MDE) techniques for expressing model queries
and constraint specification [9]. UML offers a powerful and expressive notation in creating domain models, however it lacks the capacity of explicitly specifying constraints, thus the OCL has been introduced to fulfill this purpose. The combination of UML and OCL are a great example for creating domain models and defining constraints imposed on them.

The wide adoption of UML and OCL has led to a more general model-driven engineering (MDE) methodology for software development, whose ultimate goal is to maintain the consistency among artifacts generated in various phases of software development, ranging from design, development, and integration, to the real-world deployment and execution. Most of the MDE-related research has been focused on the domain knowledge modeling and domain application development, but the requirements and constraints have remained to be one of the biggest challenges in the MDE area [10]. Several recent research works have attempted to address this limitation. For example, model-driven constraint engineering is proposed in [11] and MDE is utilized for a requirement engineering problem in industrial automation system [12].

As indicated previously, our main research focus is on the requirements and process engineering from an MDE’s perspective and the scope is not for a general software systems, but for business processes and scientific workflows in an SOA environment. Researchers have conducted numerous related research on domain modeling methodologies in software engineering over the last few decades, especially on maintaining the constraints consistency in the whole lifecycle of software development, including every aspect of dealing with constraints: representation modeling, the validation and verification of constraints and the enforcement during runtime execution. Related topics of interest to our research scope include: (i) meta-level constraint modeling, (ii) constraint ontology development, (iii) dynamic Web service composition with regard to global constraints, such as security, cost, and response time. In regard to (i) and (ii), most
of the existing research work focused on the aspects of theoretical model representation and methodologies and failed to address the real-world implementation details. Concerning (iii), various constraint models have been proposed and built to serve a very specific purpose (e.g., QoS metrics), but they have not been a good fit for the applications in other domains. Furthermore, despite the fact that much research work has been done on the individual phases of MDE, an integrated end-to-end approach remains yet to be proposed.

1.1.2 CONSTRAINTS IN WORKFLOW PROCESS SYSTEMS

In order to model process constraints, the very first task is to model the underlying business process or scientific workflow. As defined by the Workflow Management Coalition (WfMC), a workflow is “the computerized facilitation or automation of a business process, in whole or part” [13]. Over the past few decades, workflow systems have been successfully applied in numerous areas of industries, including banking, manufacturing and scientific research. A workflow is often described by various process definition languages. Business Process Model and Notation (BPMN) [14] and Business Process Execution Language (BPEL) [15] are the prevailing business process models used in these two fields. Several process definition languages and workflow languages have been developed for academic purpose, e.g., YAWL [16], OrbWork [17] and WebWork [18]. However, these process definition models lack the ability of handling process constraints, especially non-functional requirements (NFRs) and domain specific ones, alongside the functional behavior definitions of processes. For example, BPMN cannot support NFRs [19]. Using three motivating scenarios as examples, it is difficult, if not impossible, for BPMN/BPEL to (i) specify the geographic proximity constraint in an emergency handling workflow process, or to (ii) prevent potential nepotism between two performers in process activities. However, some functional and non-functional constraints are vital to process definitions of virtually all workflow applications.
during the design and development phases [20]. Therefore, we believe the research work on the constraint in process and workflow systems is of great importance to both industry and academia.

Constraints and NFRs have been a focus in workflow research since the introduction of workflow management systems due to their high impact on the overall success of workflow applications. Quality of service (QoS) is an important subset of NFRs [21]. Other process constraints may involve such factors as geographic or network locations and properties of system resources. For example, an emergency handling workflow process may require some of its tasks to be executed in close geographic proximity. Similarly, a scientific workflow may prohibit transfers and analysis of the generated experimental data by external workflows due to privacy or security concerns. In order to express such application-specific constraints and other NFRs with a process definition, workflow designers need an intuitive and clear method, and these workflow constraints and NFRs should be enforced at runtime by the workflow engine to meet users’ needs. Mapping these high-level requirement specifications to the low-level workflow execution remains a big challenge for the researchers and developers of workflow systems.

Rather than relying on a specific process definition, our work concentrates on the common aspects for a process definition, i.e., a meta process model. It generalizes from both workflow process systems and composed Web service processes, so that we will not be limited to a specific process definition language. Thus, in order not to restrict to a specific process definition language, we include the common concepts (e.g., performer, activity) from BPMN and BPEL in ProContO and later map them to a concrete process definition.

Due to the importance of constraints, researchers have proposed to elevate them to a primary level in software engineering [11, 22] from the MDE’s perspective. Model-driven constraint engineering has been proposed in [11], while constraints have been promoted to the role
of “first class citizens” in [22]. In general, a first class citizen is defined as an entity available to other entities in all operations [23]. Because one common disadvantage of current process definition languages is the lack of the capability of describing additional process constraints and non-functional requirements. Our work treats constraints as the first-class citizens and enforces the conformity of the constraint model’s various artifacts during its whole lifecycle in the process development.

1.2. SUMMARY OF MAJOR CONTRIBUTIONS

We summarize the high-level objectives of a constraint-enabled process system as follows: (i) constraint specifications should be expressed using a commonly agreed-upon vocabulary; (ii) constraint specifications should be reusable, extensible and intuitive to create; (iii) constraint specifications should be attached to any process elements, (iv) constraint enforcement should be supported in the runtime environment, and (v) when a constraint verification fails at runtime, a suitable exception should be handled properly by the runtime.

Our research work aims to treat constraints as first-class citizens in the proposed semantic end-to-end constraint framework in an SOA environment. To the best of our knowledge, little work has been done on the development of a general, extensible, end-to-end framework that supports all of the following: (i) constraint modeling at the design time, (ii) static verification at the deployment time, (iii) dynamic verification, namely, the enactment and enforcement of constraints at runtime, and (iv) utilization of semantic techniques (e.g. ontology, semantic Web services, semantic reasoning) throughout the whole lifecycle of the constraints.

The main contribution of our work is three-fold [24, 25]: (i) we have created a process constraint ontology, named ProContO, which enables process designers to express and share their knowledge of process NFRs and domain-specific constraints; (ii) we have developed a process...
constraint language, named PCL, which can be used to specify process constraints and NFRs in terms of process elements (such as BPMN tasks and data objects) and the constraint vocabulary defined in the ontology; and (iii) based on the solid foundations of constraint representation modeling laid down by ProContO and PCL, constraints are verified in an SOA environment dynamically with the help of monitoring and, if needed, constraint exceptions are handled by the process constraint framework.

1.3. Structural Outline of the Dissertation

In the rest of this dissertation, the problems mentioned above will be discussed in detail. We will provide the background knowledge that is needed first, and then present our proposed process constraint framework in a bottom-up fashion. To be more specific:

Chapter 2 gives the background knowledge of the concepts of requirements and constraints in software engineering, and workflow processes, as the whole dissertation is about constraints in workflow processes. We then provide a concise introduction to the knowledge and technologies that are used for building the process constraint framework, such as ontologies, service oriented architecture (SOA), object constraint language (OCL), and model driven engineering (MDE). All of the information presented in Chapter 3 will help us to better understand the problem domain of this dissertation as well as our proposed solutions.

We describe two motivating examples in Chapter 3, and both of them represent some typical but common scenarios in workflow processes. The first example is about some geospatial domain specific constraints in an emergency reaction workflow, and the second one involves the prevention of nepotism in career interviews.
Chapter 4 surveys existing approaches for constraint representation in both general software systems and workflow process systems. It also reviews technologies for constraint verification as well as exception handling mechanisms.

The foundation of our proposed process constraint framework lies in the model used to represent constraints in workflow processes. Such a representation model is discussed in Chapter 5. Motivations of devising such a representation model are explained first, followed by the illustrations of the two modules in the representation model: a process constraint ontology (ProContO) and a process constraint language (PCL).

Based on the representation model, the next step of the process constraints in their lifecycle is the validation and verification. Chapter 6 covers both the static verification in the design and deployment phases of workflow processes, and the dynamic verification in the execution runtime. It also includes exception handling when the verification of constraint fails.

In Chapter 7, we present the architecture of the prototype implementation for the process constraint framework. It includes the system modules and implementation details. Following this chapter, some preliminary evaluation is presented in Chapter 8.

Finally, we conclude our work in Chapter 9 with a summary of original contributions and discussions of future work and potential research directions.
CHAPTER

2. BACKGROUND

This chapter contains the background information needed for understanding problems introduced in the previous chapter, as well as the solutions proposed in the following chapters. First, we introduce background information on the requirements and constraints then follow it with a general introduction to workflow processes. Subsequently, we discuss the service-oriented architecture (SOA), which is currently a popular technology for realizing workflow process systems. Information about the Web service is covered within SOA, which will be included when discussing BPEL in the next subsection of process definition languages and notations. Besides BPEL, we also include two other process definition standards: BPMN and Petri Nets. Ontology and the object constraint language (OCL) are covered, as they are the major references when we build the foundation of the proposed framework. Finally, we discuss model driven engineering which helps to capture the project from the level of model abstraction and to better understand the whole project.

2.1. REQUIREMENTS AND CONSTRAINTS IN SOFTWARE ENGINEERING

Requirements and constraints have been vital to software development ever since the introduction of software engineering as a discipline of computer science in year 1968. They are used as definitions of what and how a software system intends to be, and the measurement of how successful the software system achieves its objectives [26]. While requirements give a general statement of what the system should be like, constraints are needed to explicitly provide the limitations and conditions under which the software requirements should be met [1]. Simply
speaking, requirements and constraints are talking about the same aspect of the system under study but at different granularity levels. Thus, some researchers make no distinction between requirements and constraints and use the two concepts interchangeably.

A simple classification divides requirements into two categories: functional and non-functional requirements. Functional requirements define functions of a software system, and the nature of the interactions between its components [27], where a function can be described as a set of inputs, the behavior, and outputs [28]. Software developers utilize use cases to describe and document the functional requirements from the aspect of how the system would behave in all the cases. In contrast to functional requirements, non-functional requirements restrict a software system in aspects, such as execution time, cost, portability, etc., other than specific behavior or functions, but some of them need to be reflected through the implementation of functions. One characteristic of non-functional requirements is that they can be expressed or defined in various ways. They are also referred to as goals/soft goals [29, 30] and qualities. Both functional and non-functional requirements can be general purpose and domain-specific. We name those requirements resulting from a specific application domain as domain-specific requirements. Examples of domain-specific constraints can be the legal restriction of the access to some information or some requirements that use domain specific terminology for definition. Domain-specific constraints are important because experts in the domain (e.g. users, clients and other stakeholders) are likely to consider them as obvious, and so frequently omit them. However, developers are usually unclear about these requirements.
2.2. Workflows

2.2.1. Workflows and Workflow Management

The concept of workflows originated from the notion of factory automation in manufacturing. However, modern understandings of workflow within the context of information systems started in 1970s and 80s, when the three pioneers Skip Ellis [31], Anatol Holt [32] and Michael Zisman [33] separately worked on the so-called office information systems driven by explicit process models [34]. Since then, workflows have been gaining considerable popularity in business. The Workflow Management Coalition (WfMC) defines workflow as “the computerized facilitation or automation of a business process, in whole or part” [13]. However, in the context of this dissertation, we adopt a more general definition of workflows as abstractions of multiple tasks which are executed and performed by a coordination among different processing entities [35, 36].

The fast development of workflow management systems, or the business process management systems, is another reason of the flourishing of workflow applications in business activities. A workflow management system is a software system, which is used to define, manage and execute workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic [13]. Usually, each workflow system is implemented around a precise process definition that it needs to follow. Several process definition standards have been proposed. While the business process execution language (BPEL) and the business process model and notation (BPMN) are the two prevailing process definition standards accepted both in business and IT realms, Petri nets and Yet Another Workflow Language (YAWL) have gained their acknowledge throughout the academia. More recently, the proliferation of workflow and business process management have been emerging as an important paradigm for facilitating scientific investigation. Several attempts have been made to apply BPEL in scientific
workflows [37, 38], and some specialized workflow systems have been developed to fit better the characters of scientific workflows [3, 4, 39].

2.2.2. **Workflow Management Lifecycle and Process Constraints**

Even though we adopt a more general definition for workflows, we agree that differences exist among different kinds of workflow management systems, such as the business process management and scientific workflow management systems. An interesting way to analyze the differences is from point of the lifecycle of the process management. Figure 2.1 is given by Wil van der Aaslt in [34], which identifies the difference between business process management and the traditional workflow management. Wil van der Aaslt claimed that business process management systems are more powerful than the traditional workflow management systems as they include the procedures for diagnosing the bottlenecks of the current process design. Similar lifecycle of the scientific workflow management is proposed in [40], as we can see from Figure 2.2.

![Figure 2.1. Workflow management lifecycle and business process management lifecycle [34]](image-url)

12
The lifecycle of the workflow process management is also the lifecycle of process constraints (as shown in Figure 2.3). Functional requirements define the behavior of processes, which directly controls the control flow of processes in the design phase. Furthermore, non-functional and domain-specific constraints should also be considered and defined at the very beginning of the lifecycle, since a process which fails to meet some crucial non-functional (e.g. security) requirements cannot be adopted in the real life. All the constraints should be deployed during the system configuration, and an important task of the constraint enforcement phase, corresponding to the diagnosing procedure of the process management’s lifecycle, is to monitor and analyze whether all the constraints are met during executions. The result of the constraint verification is further used to optimize the original process design, which in turn moves forward the evolution of processes.

Figure 2.2. Scientific workflow lifecycle [40]
2.3. SERVICE-ORIENTED ARCHITECTURE (SOA)

Different from the traditional object-oriented software analysis and design architecture, service-oriented architecture is based on discrete software resources which provide application functionality as services to other applications. To be more specific, services can be regarded as packaged software resources, which are well-defined, self-contained, standardized with the business functionality and independent from other services [41]. As their architectures aim at providing high level guidelines, they are independent of any specific implementations. There exists a wide range of implementation technologies, such as CORBA (Common Object Request Broker Architecture) and RPC (Remote Procedure Call). Web services are the most preferred implementation technology for realizing SOA’s promises of maximum service sharing, reuse and interoperability [42].

Web services are self-contained, self-describing, modular applications that can be published, located, and invoked to provide functionality to other software systems via network connections [43, 44]. A Web service meets all the three essential properties of a service in SOA: (i) an SOA-based service should be self-contained, meaning that the service maintains its own
state; (ii) the interface of the service should be limited to platform independent assertions; and (iii) services in SOA can be dynamically located, invoked and combined [45]. Proposed by W3C, the three technology standards of Web Service Description Language (WSDL) [46], Simple Object Access Protocol (SOAP) [47] and Universal Description, Discovery, and Integration (UDDI) [48], form a basic XML-driven SOA. WSDL is XML-based, machine readable, and it describes the functionality of a Web service in terms of its interfaces. SOAP provides a protocol specification for exchanging structured information among Web services on the Internet. The platform independent, XML-based UDDI specifies a mechanism for both service resource providers and consumers to register and locate Web service applications.

While traditional workflow process technologies mainly focus on the disciplines and strategies that endorse the idea of modeling and automatically running a business in terms of its end-to-end processes, SOA is an architectural approach to system development that builds, delivers and encapsulates reusable services that different applications can share them in a loosely coupled and highly interoperable manner [49]. Even though workflow process management systems can be and have been effectively deployed and execute without SOA, the combination of the two have become the most favored infrastructural approach to counter the challenges of the changing environment and the need to reduce cost and increase efficiency [50].

2.4. **PROCESS DEFINITION STANDARDS**

Process definition is a representation of process elements and their relationships from different perspectives and levels of workflow processes. For example, process designers can define what a workflow process looks like, that is to give a graphical overview of the skeletal structure of the process. Process developers can define the workflow process in the execution level, which is how the workflow process can be realized in computation systems. Consequently, researchers
have proposed graphical representation standards, execution standards, as well as the standards combining both the graphical design and runtime execution. In the rest of this section, we mainly discuss the following three process definition standards: (i) BPEL, the business process execution language, which is currently the most popular process execution standard; (ii) BPMN, the business process modeling and notation, which is a widely accepted graphical representation standard in both business and IT areas; and (iii) Petri nets, which is based on rigorous mathematical foundations [51] and provide both the graphical representation of processes and the process execution guidelines.

Throughout the brief but concise introduction of the three process definitions and their basic elements, we will have a better understanding of the issues addressed in [19, 52], that almost all the prevailing process definition standards lack the capability of describing additional process constraints, such as the non-functional and domain-specific ones. But constraints, for example that a task must be executed on a specific host due to licensing restrictions and that two tasks need to be performed by humans in close geospatial proximity, need to be specified as part of the process definition during both design and development [20, 53].

2.4.1. BPEL

Business Process Execution Language (BPEL), short for Web Services Business Process Execution Language (WS-BPEL), originated from two separate standards, namely Web Services for Business Process Design (XLANG) [54] from Microsoft and Web Services Flow Language (WSFL) [55] from IBM. BPEL is currently the most influential XML-based process definition language for specifying and executing business processes in the Web service environment [49].

The main elements of BPEL are process, activity, partner link, compensation handler, fault handler, event handler, and correlation set [56]. A BPEL process is made of activities, and BPEL
activities can be either basic or structured. Basic activities are the atomic components in business processes, and they are realized through Web services. Structured activities are the ones used for control flow resembling control structures in conventional programming languages (e.g. switch, while, pick, flow and scope). The partner link elements are used to connect the BPEL process with internal and external Web services [57], and each link is defined by a partner link type and a role name. While partner link is a BPEL element defined in the BPEL process file, partner link types and roles are specified in WSDL files of each service involved in the conversation. Partner link type determines the relationship between processes, which is further determined by specifying the port type provided by each service to receive messages. Each role specifies one port type in the WSDL file.

Compensation handler and fault handler are the two mechanisms for error handling in BPEL. A compensation handler is a container for the activities that perform compensation actions, that is to reverse the state changes of some transactions if the overall business transaction fails or is cancelled. Fault handlers deal with errors and exceptions returned by outside Web services, and they can be used with compensation handlers to perform the reversal work once some unexpected situation happens. The event handler is used to handle events coming to the business process, where an event can be a request that requires a response or an asynchronous event requiring no response. BPEL engines use a mechanism called correlation to track the multiple, long-running exchanges of messages, used to recognize and match up the messages being exchanged based on the previous messages. Thus, BPEL provides a correlation set to relate the data between different messages belonging to the right process instance for a particular conversation. In other words, correlation sets enable BPEL engines to route messages to the correct process instances that the Web service is processing.
2.4.2. BPMN

BPMN was initially proposed in 2004 by the Business Process Management Initiative (BPMI) as an industry standard for process modeling. It is a graphical, flowchart-based business process modeling language, which has been gaining popularity both in the business area and IT community [58] over the past few years. A one-to-one correspondence between BPMN and XPDL (XML Process Definition Language [59]) is also provided in order to facilitate the storage or “serialization” of BPMN diagram in the XML format. Since one of the original goals for BPMN was to visualize BPEL processes, an integral part of the BPMN specification it the mapping from the graphical notation elements to BPEL [15]. However, due to some irreconcilable differences between BPMN and BPEL, it is sometimes impossible to faithfully generate BPEL code from BPMN’s graphical representation [49, 60].

The core set of BPMN elements are flow objects (Figure 2.4), connecting object (Figure 2.5), swimlanes (Figure 2.6) and artifacts (Figure 2.7). Flow objects contain three core elements of event, activity and gateway. An event, represented by a circle as shown in Figure 2.4–a, is something that happens during the course of a business process and affects the flow of the process, causing some impact on the process. Activities (Figure 2.4–b) are the representations of the actual work the participants in the process perform. Like BPEL, activities can be atomic and non-atomic (structured or compound). Gateways, represented by diamond shapes (Figure 2.4–c), are used to control the divergence and convergence of sequence flows in a process. Flow objects are connected to create the skeletal structure of a business process by connecting objects. There are also three kinds of connecting objects: sequence flows, message flows and associations (from a to c in Figure 2.5 respectively). Sequence flows are used to show the order that activities are performed; message flows indicate the flow of messages between two different participants that send and receive
messages in a cooperative business process. Associations are adopted to link artifacts with flow objects, such as showing the inputs and outputs of an activity.

BPMN utilizes the concept of swimlanes as a mechanism to organize activities into separate visual categories according to their functions or performers. Pools (Figure 2.6–a) separate self-contained processes according to their participants or business entities, and no sequence flow is allowed to cross the boundary of pools. Lanes (Figure 2.6–b) further divide a pool according to the functionality or roles of the actives, and they can extend the entire length of a pool, either vertically or horizontally. BPMN allows modelers to extend the basic notation through artifact elements. The current version of BPMN pre-defines three types of artifacts, as shown in Figure 2.7. Data objects show how data is required or produced by activities, and groups are used to documentation or analysis purposes. Annotations provide a way to document additional text information for a process diagram.

![Diagram](image)

**Figure 2.4. BPMN core components: flow objects**

![Diagram](image)

**Figure 2.5. BPMN core components: connecting objects**
2.4.3. **PETRI NETS**

Originated by Carl Adam Petri [61], Petri nets are a class of modeling tools, which have a well-defined mathematical foundation and an intuitive graphical feature. On the one hand, the graphical feature of Petri nets facilitates the visual communication between process designers and process developers. On the other hand, the powerful mathematical formalism makes it possible to set up mathematical models of the behavior of the system [62], that enables the qualitative analysis of the process model, including the correctness of the flow of the process, the presence of mutual exclusion wherever shared resources are used, and others. [63].
A Petri net is a directed graph consisting of three structural components: places, transitions, and arcs. Places, shown in circles, define possible states or conditions of the system while transitions, which are drawn as bars and representing events that may change the states of the system. Arcs are connecting places and transitions. However, arcs connecting places to places and transitions to transitions are not permitted. In that sense, a Petri net is a bipartite graph. Dynamic behaviors of the system are represented by tokens, which appear as black dots in places.

A nonnegative integer, named a marking, can be assigned to a place in a Petri net. If a marking assigns the nonnegative integer \( n \) to place \( p \), we say that there are \( n \) tokens on \( p \). For example, the marking of \( p_1 \) in Figure 2.8 is 2. Based on the aforementioned classical formalism of Petri nets, many extensions have been introduced. For example, colored Petri nets [64] enable the distinction between tokens by allowing them to have a data value. In timed Petri nets [65], a notion of time is applied to the classical formalism to analyze the temporal behavior of systems.

![Petri net example](image)

Figure 2.8. Petri nets: an example

### 2.5. Ontology and OWL

The term of ontology originated from the realm of philosophy, but has been applied in many different ways. In computer science area, ontology formally represents “a specification of a conceptualization” [66]. Ontologies provide common vocabularies to define and encode concepts within a domain, as well as the knowledge that spans domains. Knowledge can also be described in ontologies with different degrees of structure, ranging from simple taxonomies, to metadata.
schemes and to logical theories [67]. Besides the structural relationships, ontologies also include general relationships or linkages among the knowledge. The characteristics of unambiguity, reusability and extensibility make ontologies more and more popular in various realmes of disciplines, such as artificial intelligence, software engineering, bioinformatics, etc.

There are many languages used to define ontologies. Most of them have semantics based on logic, so that detailed, accurate, consistent, sound, and meaningful distinctions can be made among the classes, properties, and relations [67]. The Web Ontology Language (OWL) [68], based on description logic, is now the prevailing ontology language in computer science, as it aims to publish and share domain knowledge, and to make it easier for machines automatically to process and integrate information via the Web. Furthermore, OWL has become a W3C recommendation for the Semantic Web, since 2004.

Based on Resource Description Framework (RDF [69]) and RDF Schema [70], OWL provides more facilities in expressing meanings and semantics than RDF. It is actually a family of three language variants of increasing expressive power: OWL Lite, OWL DL and OWL Full. The second version of OWL, OWL 2, extends the original OWL with a small but useful set of features, such as additional property and qualified cardinality constructors, extended data-type support, simple meta-modeling, and extended annotations, for which effective reasoning algorithms are now available [71].

However, OWL 2 language is still not able to express all relations. For example, OWL 2 cannot express the relation between individuals with which an individual has relations (e.g., child of married parents) [72]. In this case, the Semantic Web Rule Language (SWRL), similar to rules in Prolog and the DATALOG language, is introduced to extend the expressiveness of OWL. Figure 2.4 is the ontological representation of Parent and Child, and both classes are sub-classes of the
Person class. By appending a SWRL rule, shown in Table 2.1, to the OWL definition of Person, Parent and Child, we can easily tell whether an individual of Person is a Parent or not.

![Ontological representation of Parent and Child](image)

Figure 2.9. Ontological representation of Parent and Child

Table 2.1. SWRL rule example: inferring a Person is a Parent

| Person(?p), hasChild min 1 Person(?p) -> Parent(?p) |

2.6. Object Constraint Language

Object Constraint Language (OCL), is a formal, typed, declarative and side-effect free specification language used to define a variety of constraint expressions that complement the Unified Modeling Language (UML) models [9]. Graphical modeling notations, such as UML, are suitable for modeling structural aspects of a domain (limited types of constraints, such as multiplicity constraints can be provided as textual annotations or adornments). However, graphical notation’s expressiveness usually sacrifice in order to keep the complexity of the notation manageable. Moreover, it is difficult for graphical models to represent some typical constraints, such as pre- and post-conditions of operations. This is why designers regard a formalized, textual constraint language as a necessary complement to a graphical notation.

OCL can be used to specify invariants on classes, association, and other elements in a UML model, and it can also describe pre- and post-conditions and constraints on operations and attributes. OCL is also a navigational language that can handle association end points in UML.
models. Figure 2.10 illustrates a simple class diagram consisting of two classes and their relationships. Sample constraints illustrating the above-mentioned features of OCL, and showing how OCL helps to increase the expressiveness of UML models are shown in Table 2.2, Table 2.3, and Table 2.4.

Table 2.2. OCL example: invariants

<table>
<thead>
<tr>
<th>context</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv</td>
<td>self.stockPrice() &gt; 0</td>
</tr>
</tbody>
</table>

Each OCL expression should be written in a context of an instance of a specific class or other UML element, and the key word context is used to indicate the specification (Table 2.2, Table 2.3, Table 2.4). The reserved word self is used to refer to the contextual instance (Table 2.2), and an equivalent formulation playing the role of self is an alias designated to the context as shown in Table 2.4. Invariants, followed by the keyword inv, determine constraints that must be true for all instances of a class (Table 2.2). Pre- and post-conditions are constraints associated with an operation or some behavioral features of a class, where pre-conditions determine constraints assumed to be true before the operation is executed, and post-conditions are constraints satisfied

---

1 The example is converted from http://cs.ulb.ac.be/public/_media/teaching/infoh302/oclnotes.pdf?id=teaching:infoh302
after the operation is executed (Table 2.3). The reserved word `result` denotes the result of an operation, and is often used in specification of post-conditions. OCL is also a navigational language; it can be used to handle constraints on associations in UML models. In the example shown in Table 2.4, the OCL expression navigates from the object `Company` to the object `Person` through the role name of the association.

Although OCL was initially introduced as the complement to UML, it now has become a key component of the Object Management Group’s (OMG) standard recommendation for transforming models, which is a core aspect in the model driven engineering (MDE). The navigational feature of OCL enables it to navigate through other models (e.g., navigate through XPath in XML models). Besides, it can also serve as an assertion language or even used as a side-effect free programming language for making requests on different kinds of models.

Table 2.3. OCL example: pre- and post-conditions

<table>
<thead>
<tr>
<th>context</th>
<th>Company::income(): Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre</td>
<td>self.age &gt;= 18</td>
</tr>
<tr>
<td>post</td>
<td>result &lt; 10000</td>
</tr>
</tbody>
</table>

Table 2.4. OCL example: navigating associations

<table>
<thead>
<tr>
<th>context</th>
<th>c: Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv</td>
<td>c.manager.isUnemployed = false</td>
</tr>
</tbody>
</table>

2.7. **Model Driven Engineering (MDE)**

Model-driven engineering (MDE) [73] is a software development methodology. Instead of focusing on designing algorithms for the computation, which is adopted by the traditional software engineering methodologies (e.g., object-oriented software development), MDE works on exploiting abstract representations of knowledge in different domains, or “models”. The model
driven architecture (MDA) standard has been proposed by OMG. MDA is a specific incarnation of the MDE approach [74], which can help us to understand the essence of MDE.

The fast evolution of middleware platforms (e.g., J2EE and .NET) and the increasing need to incorporate Web-based front ends has led to the realization that not all systems written in obsolete programming languages can be changed to support new standards [73, 75]. Working on the improvement of productivity in the long-run, software developers and researchers found that it is a way to reduce the rate at which software artifacts become outdated by raising the level of abstraction at which primary software artifacts are written [76]. Thus, MDA is a solution proposed for such problems. By identifying and separating the specification of system functionality or the so called platform independent model (PIM) from the specification of the implementation of that functionality on a specific technology platform, or the platform specific model (PSM), MDA stands at a higher abstraction level than the traditional object-oriented style of software engineering. The same PIM then can be realized on multiple PSMs through the auxiliary mapping standards, or through point mappings to specific platforms [75]. Although it would be the best if the mappings between PIMs and PSMs are automated, MDA does not exclude manual or semi-automatic ways of mapping techniques. MDE extends the the idea of building a more general and abstract model for some specific models (e.g., PIM for PSM), and introduces the concept of meta-model. Trying to work at a higher abstraction level, or following a more “meta” direction, MDE has provided guidelines for a new generation of software engineering, and also stimulated some ideas in our research.
Chapter

3. Motivating Examples

In order to illustrate our proposed constraint framework, we briefly describe two motivating examples. The first example involves geospatial constraints (e.g. distances between tornados and nearby hospitals) in an emergency reaction process, and the second one is about a common scenario of avoiding nepotism in job interviews. Definitions of both processes are simplified and trimmed down from real world applications (e.g. the tornado emergency reaction workflow process [77]), and such simplification helps us to better focus on the constraints of interest instead of the process definitions, and to better understand how the framework model, monitor, verify and enforce each constraint in its whole lifecycle. As mentioned in the introduction, one major objective of our framework is to concentrate on process constraint modeling without being restricted to a specific business process definition language (e.g., BPMN or BPEL). Here, we use flowcharts to represent the control flow of both processes.

3.1. Prevention of Nepotism in Job Interviews

The first motivating scenario comes from a common and natural restriction in the real world: that is to avoid the nepotistic relationships in every workplace. An efficient way of preventing nepotism is to avoid job interviewees being judged by their relatives in job interview processes, which we generalize as a simple workflow consisting of two sequential tasks (shown in Figure 3.1). Constraints can be set as the restriction to the roles of the job applicant and the interviewer. Instead of doing the tedious, prolonged background check manually, semantic ontological reasoners can help to detect the potential nepotistic relationships automatically as the
verification of the constraint, once semantic rules of kinship are properly defined. This particular scenario demonstrates several advantages of introducing ontology within the process constraint framework, that the ontology does not only clearly defines the meanings and relationships among process constraint concepts, but also supports inference and reasoning for the constraint verification.

![Figure 3.1. A Simple Job Interview Workflow Process](image)

3.2. **TORNADO EMERGENCY REACTION WORKFLOW PROCESS**

Figure 3.2 shows an abstraction of the tornado emergency reaction workflow process. Weather service organizations gather tornado information from sensors and local residents to detect and assess the risk. If it is not a tornado, they keep gathering weather information and detecting potential emergencies. If real tornados are spotted, emergency management agencies issue tornado alarms at once, and inform nearby hospitals and shelters to get prepared at the same time. Safety issues need to be carefully considered in all emergency reaction processes. For example, hospitals and shelters that are distributed along the tornado’s moving path are not considered to be safe, and some of them even need to be evacuated. However, for the efficiency purpose of rescuing the potential wounded, hospitals and shelters should not be too far away from where the disaster happens.

It is obvious in this case that geospatial constraints, such as the distance between a hospital and the tornado, are the facts that would impact an execution instance of such an emergency workflow process. On the other hand, however, those constraints are determined by the context information of the emergencies, which is usually not available until the execution of the workflow process. Throughout this thesis, the example of the emergency reaction workflow process will help
us to find out: (i) how the framework defines the related constraints together with the process definition; (ii) how the execution of processes is monitored for runtime information; (iii) how the framework dynamically evaluates and verifies the geospatial constraint by calculating the distances between a tornado and a hospital nearby; and finally (iv) what the framework will do if constraints cannot be met and exceptions are raised.

![Tornado Emergency Reaction Workflow Process](image)

**Figure 3.2. Tornado Emergency Reaction Workflow Process**

### 3.3. Glycomics Scientific Workflow

GlycoQuant IDAWG™ workflow has been used by scientists at the Complex Carbohydrate Research Center at the University of Georgia to perform quantitative glycomics analysis. One part of the workflow can be represented as four sequentially connected tasks shown in Figure 3.3. The raw data are produced by the mass spectrometer experiment. However, transferring raw data directly over the open Internet may not be feasible due to the large data size (e.g., in gigabytes). Instead, it is preferable to transfer the data pre-processing task to the computer storing the raw data, and then only transfer the segmented and encoded data back to more powerful servers for further computational analysis. In the meantime, when the raw data’s file size is small and is not sensitive from the point of view of the security, it is allowable to send these data directly to the remote server, which has more powerful hardware. In order to accommodate these requirements and impose minimum changes to the existing systems, it is desirable to have process constraints control these dynamic behaviors.
Figure 3.3. Glycomics Scientific Workflow
4.1. CONSTRAINTS MODELING AND REPRESENTATION

Considerable amount of work has been done on modeling non-functional requirements (NFRs) within the general software engineering research. Logic systems, such as first order logic, can be used to express NFRs [78]. NoFun, a formalized language to facilitate quantitative analysis of NFRs, has been introduced in [79]. A framework consisting of two languages, the process-NFL and the product-NFL, was proposed in [80] for building non-functional software architecture both in software developing phase and for the final software products.

Graphical notation has also been used in modeling requirements. For example, in [29], the NFR Framework proposed softgoal interdependency graph (SIG) to represent NFRs and their decompositions, as well as to keep track of the impact of various decisions through labels on SIG [20]. Yu [81] proposed the i* framework, which utilized graphical representation to model NFRs during the early phases of software development. While NFR Framework and i* framework focused only on NFRs, [82] presented a methodology to link NFR graphs with UML, the standard object-oriented analysis and design modeling language. KAOS [83] proposed using the AND/OR graph to relate requirements to operations, so that there would be no difference between functional and non-functional requirements.

Researchers have also focused on modeling of NFRs and constraints within specific types of software systems, such as the workflow systems. In [84], the authors proposed an approach based on extending a role activity diagram (RAD) process model to more explicitly describe
desirable operational goals through the analysis and evaluation of NFRs. By introducing artifacts, called the Operating Condition and the Control Case added to BPMN, [85] illustrates how the two modeling artifacts help to better represent and discover NFRs at an early phase of the business development life-cycle.

In summary, the formalized language representations help developers to more easily express and document constraints, while the graphical notations provide an intuitive way for eliciting and visualizing them. Frequently, a given constraint or an NFR can be stated differently using different vocabulary [79], which may lead to imprecise and ambiguous specifications. This is a difficult problem for the reported modeling approaches. In [82], the authors dealt with this problem by embedding specific keywords in their modeling framework to control the concepts and vocabulary used by developers. However, it is next to impossible to reach a consensus on a good set of constraint concepts expressive enough to cover a wide variety of application domains.

Recently, ontologies have been proposed to promote the consensus regarding the constraints concepts and relationship among them. In [86], Dobson, et al. developed an ontology to model all non-functional aspects in service-centric systems, based on which, he and his colleagues later presented a domain-independent ontology for NFRs and illustrated its application in a business trip service [87]. DAML-QoS [88] is another example of using ontology to model quality of service for web services. Much work has been focused on building ontologies for QoS and/or NFRs in business workflows and Web service applications, and various domain-specific requirements can be found in the implementation of scientific workflows. However, based on a rapidly developing interest in scientific workflows, there is an increasing need for a general-purpose constraints specification framework that can model both non-functional and other domain-specific requirements. Our work addresses these issues by introducing an extensible process
constraint ontology, as well as a process constraint language. We also propose a software framework suitable for the development and execution of workflows incorporating constraints and non-functional requirements.

4.2. PROCESS CONSTRAINT VERIFICATION

General software system development has four phases: (i) model design, (ii) code development, (iii) system deployment, and (iv) runtime execution. Models developed recently not only fulfill documentation purposes, but also are treated as first-class artifacts in the whole development process. Thus, model-driven engineering (MDE) has attracted much attention from both academia and industry. Model-driven constraint engineering has been proposed in [11] to address the lack of support for constraint modeling. It specifies textual constraints specified using the OCL language, parameterizes them in a computer-aided software engineering tool, and then transforms them into platform-dependent constraints via model transformation. Although this approach identifies several constraint patterns, it does not address the issue of constraint verification, which is the very next step in the MDE process. In [89], the authors introduce a representation model by extending BPEL with UML graphical models to define some of the QoS requirements within the process definition, for the purpose of static Web service composition in the grid environment.

Constraint verification is often divided into (i) static verification during design and deployment phase and (ii) dynamic verification during runtime execution. Static verification during design has traditionally been based on formal verification, such as model checking and theorem proving. It is often very complex, time-consuming and does not scale up well in distributed systems. Several approaches have been proposed to address static verification during the deployment phase. In [90], the author concentrates on quantifiable non-functional requirements
(NFR) using automata-based modeling and verification of NFRs (e.g., security algorithm) in composite Web services. In [91], constraints on Web service’s input and output messages are treated as instances in ontology and used to aid a Web service composition via semantic graph transformation. A dynamic discovery method is also proposed for Semantic Web services in [92].

Dynamic verification or runtime verification has attracted much attention in the last few years because it works directly with the system under study and has the advantage of avoiding the complexity of traditional formal verification methods. Monitoring-oriented programming (MOP) [93] and its related software library JavaMOP [94] has gained wide recognition. MOP supports runtime property monitoring and quantifiable requirement verification by combining requirement specification and actual system implementation. It defines various aspects related to the requirements, translates constraints into actual Java classes and integrates them within the actual implementation using aspect-oriented programming (AOP) techniques. Although MOP framework works well in monitoring, tracing the execution of programs and keeping track of the changes of quantitative properties, it only handles parametric verifications and does not have a general constraint model to handle more complex constraints.

While the methods reported in [93] and [94] work at the level of Java classes and are not specific to any workflow process applications, [95] and [96] focus on the runtime/dynamic monitoring and checking requirements for composed Web services based on BPEL. A monitoring rule language is proposed in [95], which separates the procedures of business logic design and the requirements or the monitoring rules designs. By weaving the monitoring rules and the corresponding monitoring services into the target BPEL process, it achieves the purpose of checking and verifying the requirements during the execution of the BPEL process. Unlike [95], the framework proposed by [96] is totally non-intrusive. [96] uses event-calculus for specifying
the requirements that need to be monitored, and events are then observed at runtime. It stores the event information into a database and the verifications of requirements are done based on integrity constraint checking in temporal deductive databases. The method discussed in [95] is the “inline” monitoring defined in [93], which embeds the monitoring code within the system under study. The approach in [96] is a typical representation of the “outline” monitoring. The outline monitoring mechanism is more flexible than the inline one, as it can monitor the context information of execution and provide some additional supervision and management at the same time. As it is less intrusive, it is less responsive in discovering runtime exceptional situations than the inline monitoring. That is why we use both inline and outline mechanism for monitoring in our process constraint framework.

4.3. Exception Handling

Exception handling (e.g., service failure or network outage) has been more widespread in the SOA environments than desktop applications and needs to be specially addressed. For the general UML model, the existing OCL specification does not explicitly indicate how to detect or handle a constraint violation (i.e., exception) and leaves the detection and handling of run-time violations to the software developer [97].

Different approaches have been proposed to address the issue of exception handling within the context of workflow process system. Some of them are based on the exiting process definition standards, while others introduce new models for representing the business logic. In order to deal with the unexpected situations or the so called deviations in [98], Luo et al. built an exception-aware workflow systems by using justified event-condition-action (JECA) rules to model the business logic of workflow processes. Besides some automatic mechanisms, such as ignoring subtle deviations, re-executing problematic tasks and replacing tasks by changing the original
JECA rules, their work also supports the human-involved ways to decide what to do when exceptions happen. It further proposes an ontology aided case-based reasoning (CBR) system, which can help to make decisions based on previous exception handling experience.

Adams et al. introduced an artifact named “Exlets” in [99] for exception handling within the context of YAWL [16]. Exlets are used to manage the exception services, which are separate components from the YAWL defined business logic, and can be executed in different workflow execution engines as long as they follow the service-oriented paradigm. Exlets can be checked before and after a task execution, which correspond to the attached pre- or post-condition constraints. The idea of our approach is similar to the “Exlet” artifacts in YAWL, that exception handling can be defined and executed as some external means other than embedded within the same workflow engine of the normal flow. As YAWL provides no intrinsic exception handling mechanism, there is no backup solution if the external exception services unfortunately fail.

The approaches discussed above, including what we proposed within the process constraint framework, can be classified as the throw-and-catch scheme, since they are all based on pre-defined exceptions and provided corresponding handlers. Yet, there exist other means of dealing with exceptions, different from the prevailing throw-and-catch scheme. Ardissono et al. presented an exception diagnosis framework in [100, 101]. Driven by the fact the Web service processes often span several business partners and are executed in different partner nodes, they proposed a distributed error-handling model, which assumes that every participant within the service process has a local “diagnoser”. All messages exchanged among the participants go through the diagnoser component, and log information of the local service is used for analyzing the reasons for the exceptional situation. For failures that are possibly caused by errors in another service, a global diagnoser is introduced to deal with such situation. As the messages are no longer directly
exchanged between Web services, a new message exchange protocol has been proposed, which is a leading reason why such a distributed mechanism has not been widely adopted in today’s applications.
CHAPTER

5. PROCESS CONSTRAINT REPRESENTATION MODEL

In this chapter, we will introduce the representation model of process constraints, which consists of a process constraint ontology (ProContO) and a process constraint language (PCL). Generally speaking, while ProContO helps designers and software developers to reach a consensus regarding the constraint specification vocabulary and to share the knowledge about process constraints, PCL provides a concise and convenient approach to express constraints along with the process definition.

When we have been designing the representation model for process constraints, one objective has been to concentrate on process constraint modeling without tying it to a specific business process definition language (e.g., BPMN or BPEL). The are advantages of working with just the common business process concepts. First, constraint designers (e.g., business analysts or scientist) can focus on the constraints without knowing the actual process implementation. The hierarchy rooted at the top-level ontological class ProcessElement contains the common information needed to describe a business process, e.g., performer, activity, and others. Furthermore, because of the extensibility of ProContO, new properties or classes can be easily added to the hierarchy of ProcessElement as necessary. Finally, software developers can map ontological ProcessElement instances from ProContO to concrete business process definitions (e.g., BPMN or BPEL).

Another major objective in proposing a representation model comprised of ontology and language is to promote constraints as first-class citizens in our end-to-end process constraint
framework. *ConstraintAttributes*, which are the basic components of constraints, serve as the basic class in the UML diagram for PCL, as shown in Figure 5.1. Simple constraints (e.g., response time), represented by the *SimpleAttribute* class, are usually defined as data properties or single object properties in ProContO. When it comes to constraints that require complex calculations (e.g., a throughput ratio) or semantic reasoning (e.g., constraints guarding against potential nepotism), *ConstraintOperations* are used to delegate the computation to an external Web service or a semantic reasoner (e.g., a SWRL reasoner). Inputs to a *ConstraintOperation* and the corresponding output are all represented as *ConstraintAttributes* that reuse the ontological concepts existing in the domain ontology. In summary, all of the simple quantifiable constraints, inputs/outputs of the constraint operations use constraint attributes and can appear everywhere in PCL expressions. By elevating constraints to the level of first-class citizens, our process constraint framework is both expressive and extendable.

![Figure 5.1. UML model for PCL: promoting constraints as first-class citizens](image)

Such an ontology and language-combined representation model can be regarded as a powerful supplement to process definition standards. ProContO and PCL also serve as the main
building blocks for the subsequent phases of constraint verification in the end-to-end process constraint framework. The outline of this chapter is as follows: Section 5.1 discusses the ProContO ontology in detail and Section 5.2 provides guidelines on how to define process constraints using PCL. More specifically, we first present the structural outline of the ontology, followed by the further descriptions of the each top level class of ProContO in four separate subsections. The syntactic definition of PCL is given in EBNF (Extended Backus-Naur Form) and the PCL definitions of constraints related to the motivating examples in Chapter 3 are also provided.

5.1. PROCESS CONSTRAINT ONTOLOGY (ProContO)

Specification and handling of process constraints is an integral part of a well-designed workflow application. The motivating examples present some types of constraints that can be found in many other processes with similar types of requirements. We believe that ontologies offer the requisite expressive power to define the knowledge about process constraints, their classification and relationships among them, as well as the mechanism of connecting them to process components.

The outline of the ProContO ontology is depicted in Figure 5.2. The class of ProcessElement defines components in process definitions (activities, data objects and performers). A ProcessElement can be restricted in a process definition by many constraints, which are composed using ConstraintAttributes, such as the execution time of an activity, the network or geospatial position of an activity, or the size of an input data object. Complex ConstraintAttributes need to be computed by suitable operations (e.g., a throughput ratio) or using semantic reasoning (e.g., constraints guarding against potential nepotism), and ConstraintOperations are used to delegate the computation to external Web services or semantic reasoners.
Input arguments to a ConstraintOperation and the corresponding output are all represented as ConstraintAttributes that reuse the ontological concepts existing in the domain ontology.

While the three above mentioned ontological classes (ProcessElement, ConstraintAttributes and ConstraintOperation) are adequate to provide the semantic meaning of a process constraint, the exceptional situations when a constraint cannot be satisfied in the runtime execution should also be considered. As a consequence, we introduce Exception as another top-level class in the ontology. As shown in Figure 5.2, the four classes and their relationships serve as the backbone of the ProContO ontology. The extension and expansion of the ontology by adding new concepts under these top-level concepts is easy and convenient by following the concise backbone structural guidelines.

![Figure 5.2. Backbone of the Process Constraint Ontology (ProContO)](image)

5.1.1. PROCESS ELEMENTS

Process definition languages or notations available today, such as BPEL and BPMN, use different names for the elements of a workflow process, but their functions and relationships are similar. By following the BPMN specification, ProContO uses Activity class to represent the atomic work or a task within a workflow process. Each Activity is executed by a processing entity named Performer, and an Activity may consume some input Data and produce the output Data (Figure 5.3). Even though ProContO includes only these three most common and fundamental elements of ProcessElement, new concepts or classes can be easily added under ProcessElement when deemed necessary.
5.1.2. **Constraint Attributes**

*ProcessElements* can be described by attributes, such as the execution time of an *Activity* and the size of a *Data* object. These attributes can later be used in defining process constraints. For example, a requirement can be set as “the execution time of the computational task should be less than 10 minutes”. We define the *ExecutionTime* of an *Activity* under a super class named *ValueAttribute* (as shown in Figure 5.4), since the *ExecutionTime* can be simply described as an integer or float value followed by a proper unit (e.g. 30 milliseconds, 5.0 seconds, etc.). It is similar to the *SizeAttribute* for the *Data* object. There are also attributes that cannot be just defined using the elementary data types (e.g. integers, Boolean values, strings, etc.). As an example, consider the domain specific constraints in the motivating example of the tornado emergency reaction workflow process, the *GeoLocation* of the tornado and its nearby hospitals are represented by their longitudes and latitudes.

*ProcessElements* are connecting to *ConstraintAttributes* by various object properties (e.g. *happensAt* relationship between *Activity* and *GeoLocation* in Figure 5.4), and data properties (e.g. *hasSize* and *hasDuration* properties). They should be all defined as subproperties of the *hasAttribute* as defined in the ontology’s structural outline.
We incorporate the Metric class in the ProContO ontology, as it is frequently meaningless to define a numerical constraint without giving its unit. For example, the ExecutionTime of an Activity may be specified in milliseconds or hours. Defining various units of measure is important but goes beyond the scope of this dissertation. For our framework, we reuse the concepts and definitions in a metric ontology named QoSOno presented in [86].

5.1.3. CONSTRAINT OPERATIONS

While some constraints (e.g. execution time of a task and the size of input data) can be easily expressed as simple properties attached to ProcessElements, some constraints can only be defined through mathematical computations or semantic reasonings. The ConstraintOperation class in the ProContO ontology has been introduced to deal with those complex constraints.

Considering the domain specific constraints in the tornado emergency reaction process, the actual geographical locations of tornados and the nearby hospitals are represented as coordinates consisting of longitude and latitude values. Distances between the emergency and hospitals can be calculated based on their longitudes and latitudes. The Distance operation, as shown in Figure 5.5, takes GeoLocations of the two tasks as its input parameters, and produces a double number (an instance of the ValueAttribute class) as its result. ConstraintOperations can also produce other
ConstraintAttributes as their outputs rather than the elementary data types (represented by the ValueAttribute class). The Buffer operation is such an example. It is used to calculate a polygonal buffer area surrounding danger of a tornado.

![Diagram](image.png)

Figure 5.5. Examples of ConstraintOperation class and the relationship between ConstraintOperation and ConstraintAttribute

5.1.4. EXCEPTIONS

Exceptions may occur at any moment (i.e., before, during or after) with respect to the following two situations: (i) the execution of a specific ConstraintOperation or (ii) the evaluation of a ConstraintAttribute. Taking the motivating example of the tornado emergency reaction workflow process, one constraint can be specified on the distance between the tornado and the hospitals. For safety purposes, the distance between the two is set not to exceed 30 miles; but for efficiency purposes, hospitals should not be more than 50 miles away from where the tornado emergency happens. Under certain unexpected circumstances (e.g., there are no available hospitals under such a constraint), an exception should be raised and signaled, and then the control of the process execution flow should be handed over to the associated exception handler, if one has been defined. Ontological definitions of this constraint and the corresponding exception are given in Figure 5.6.
The general exception concept can be modeled as a quadruple $E$(Context, RelatedEntity, Type, Handler), and its elements are as explained below:

- **Context**: It includes the contextual and runtime information about where this exception occurs, e.g., current running task, parent business process, and the performer. All the context information depends on the runtime and some of them may be missing or not available due to the severity of exception (e.g., electricity outage).

- **RelatedEntity**: It represents which entity triggers the exception or violates the constraint and its range can be either ConstraintAttribute or ConstraintOperation.

- **Type**: It is the exception’s type, which can be user-defined (e.g., the `HospitalNotFoundException` in Figure 5.6), or pre-defined to represent some common causes, which may lead to unexpected situations (e.g., failure, fault or error, as discussed in [98]). The type information determines the severity and the extent of the
exception. For example, in most cases, exceptions of ordinary types or errors can be treated as recoverable, but failures are unrecoverable.

- **Handler**: It is an exception handler, which should be invoked, if specified by a process designer. Some general-purpose exception handlers are also provided, such as a process error logger. A handler can also be used to change the flow of execution, for example, to stop the instance execution when the 3DES algorithm is not being used.

The advantages of introducing the exception as a top-level ontological class are two-fold. First of all, it promotes sharing of knowledge and reuse of exception types and, most importantly, exception handlers. Secondly, the Context and RelatedEntity can be useful to identify the cause of the fault/error/exception when one is raised, and thus, a better exception handling mechanism can be provided with the help of a context-aware monitor, which will be discussed in the next chapter.

5.2. **PROCESS CONSTRAINT LANGUAGE (PCL)**

After the control and data flows are defined and related constraint concepts are elicited in the ProContO ontology, the next step involves specifying constraints and connecting them to the corresponding elements in the process definition. The Process constraint language (PCL) has been created as to achieve the purpose. It serves as a declarative specification language for formulating and documenting process constraints, including non-functional and domain specific ones. PCL constraint expressions are attached to the process design and ultimately deployed to the enactment service for execution. By following a similar syntax as Object Constraint Language (OCL), PCL is a useful supplement to the expressiveness of current process definition languages or notations.
### 5.2.1. PCL Expressions

A PCL expression is a logical assertion of a constraint, which evaluates to a Boolean value (true or false). Table 5.1 shows the syntax of PCL expressions defined in EBNF (Extended Backus-Naur Form). A literal is the smallest expression in PCL, which can be a string, a number, or a name. A name is an identifier referring to a concept in the constraint ontology or a name of an activity in the process definition. Larger expressions are formed with the use of unary and binary operators, shown in Table 5.2. The operators include the usual arithmetic and logical operators, as well as the navigation operators.

| expression | ::= | logical_expr |
| logical_expr | ::= | relational_expr {logical_op relational_expr} |
| relational_expr | ::= | arithmetic_expr [relational_op arithmetic_expr] |
| arithmetic_expr | ::= | unary_expr {arithmetic_op unary_expr} |
| unary_expr | ::= | [unary_op] navigation_expr |
| navigation_expr | ::= | primary_expr [navigation_op name] |
| primary_expr | ::= | (“” expression “”) | if_expr | constraint_call | literal |
| if_expr | ::= | “if” expression “then” expression “else” expression “end if” |
| constraint_call | ::= | name (“” [constraint_params] “”) |
| constraint_params | ::= | expression {“,” expression} |
| literal | ::= | string | number | name | “true” | “false” |

| Operator | ::= | “not” | “and” | “or” | “xor” |
| logical_op | ::= | “==” | “>” | “<” | “>=” | “<=” | “<>” |
| relational_op | ::= | “!” | “*” | “/” |
| arithmetic_op | ::= | “.” | “→” |
| navigation_op | ::= | “.” | “→” |

The two navigation operators are used to traverse relationships in the ontology. Starting with an expression identifying an ontology class as the left operand, the “.” operator is used to

---

47
access the class on the other side of the association (e.g., t1.executionTime, where t1 is the name of an activity), while the “→” is used to traverse a specific relationship to reach the other class (e.g., t1→hasOutput would access t1’s output data). By convention, the right operands of navigation operators should start with a lower case letter. A constraint call is an invocation of a ConstraintOperation defined in the constraint ontology. By convention, the name should start with a lower case letter.

5.2.2. CONSTRAINT DECLARATION

Connections between constraints and processes are specified in the context definition part. The keyword context is used to indicate which ProcessElements in the process definition are involved in the constraint specification. As the name of an activity can be long, activity aliases can be introduced at the same time. A constraint includes one or more conditions, which are either invariants, pre-, or post-conditions. The syntax of constraint declaration is shown in Table 5.3.

Table 5.3. PCL Constraint Declaration

| constraint_declaration | ::= “constraint” name |
| context_definition | context { condition } |
| | { exception_declaration } |
| context_definition | ::= “context” [alias “.”] name |
| | {“,” [alias “.”] name} |
| condition | ::= constraint_type [name] |
| | expression {“,” expression} |
| | {“raise” exception_type } |
| constraint_type | ::= “inv” | “pre” | “post” |
| exception_type | ::= name {“,” [alias “.”] name} |
| exception_declaration | ::= “exception” “when” name |
| | “then” expression {“,” “when” name “then” expression} |

An invariant condition must hold during workflow execution. More specifically, an invariant condition is checked before and after the execution of all activities listed in the context definition of a constraint. In case not all of the constraint attributes used in the expression are
available (have been established) due to the relative ordering of activities determined by the process control flow, the assertion is considered true. Pre-conditions define the required state of some process elements before they start to execute. If more than one activity is declared in the context definition, the constraint expression specified within the **pre** clause need to be verified at the starting point of each activity instance. Again, the constraint is trivially asserted as true if some attributes used in the constraint specification are not available. It is similar with post-conditions, which are verified at the ending point of each activity instance.

One objective of PCL is to integrate constraint definition with process control and data flow definitions. Therefore, a name given to a constraint in the declaration not only facilitates the documentation and serialization. At the same time, assigning a name to a constraint makes it easier to connect the constraint to its corresponding exception handling methods.

### 5.2.3. Exception Declaration and Handling

As shown in Table 5.3, the **raise** keyword is use to declare the type of exception to be raised in case the constraint is not satisfied. Exception handlers, specifying what should be done when an exception is triggered and signaled, are declared in the **exception** clause. Table 5.4 gives a complete definition, including the declaration of a user defined exception and its corresponding handler, of the GeoLocationProximity constraint in the tornado emergency reaction workflow process. An exception handler named **extendDistance** is defined in this case. It can be realized the same way as constraint operations. Once no hospital has been found from the distance of 30 to 50 miles, the constraint will fail and as a compensation, the exception handler service will be called.

<table>
<thead>
<tr>
<th>constraint</th>
<th>GeoLocationProximity</th>
</tr>
</thead>
<tbody>
<tr>
<td>context t:</td>
<td>TornadoDetection, h:</td>
</tr>
<tr>
<td></td>
<td>HospitalPreparation</td>
</tr>
</tbody>
</table>
The declaration of an exception in a constraint definition is optional. Whether an exception and its corresponding handler are explicitly declared affects how the process constraint framework would deal with such an exception in the runtime execution. If the exception and its handler are given in the constraint definition, when such an exception occurs during runtime, the process constraint framework will first invoke the user defined exception handler. If not, the process constraint framework triggers its build-in default exception that logs the contextual information and directly interacts with the concrete workflow execution engine for compensation. The mechanism of how our proposed process constraint framework deals with exceptions in runtime execution will be further explained in the following chapters.

While developing PCL, our objective has been to follow the principle of design-by-contracts [102] and then enforce the adherence to the constraints during the whole software development lifecycle. The introduction of exception declarations and their handling to PCL syntax promotes design-by-contracts even further. With the help of exception handling definition, PCL not only enhances the documentation of process designs and their runtime verification, but also provides a way to specify how to react to exceptions raised during runtime.
6. PROCESS CONSTRAINT VERIFICATION

The business process development lifecycle is often divided into five phases: design, analysis, implementation, deployment, and execution [103]. In order to focus on the constraint modeling, we do not distinguish between design and analysis and assume the service implementation is readily available to be composed into more complex business process. Therefore, the whole lifecycle is simplified into three major phases, namely, design, deployment and runtime execution. Constraint specifications introduced during the design phase must be maintained and enforced during the rest of the process lifecycle in MDE. In order to provide an end-to-end constraint verification framework, we should support the following: (i) static verification during design phase, (ii) static verification during the deployment phase, and (iii) dynamic verification within the runtime environment, as shown in Figure 6.1. After process constraints have been described in ProContO and defined in PCL, along with the business process definition, the additional steps must ensure that (i) the constraints are both syntactically and semantically correct, (ii) the constraint operations are valid and working properly, and (iii) the conformance to these constraints within the runtime environment. The first two steps focus on the syntactic and semantic checks in the design and deployment phases, respectively, while the last step monitors the runtime execution of business processes and ensures these constraints are satisfied. In the following sections, we will discuss the various verification steps in greater detail.
6.1. Static Verification

In accordance with our constraint representation model, each process constraint in PCL includes the following syntactic constructs: context, conditions and exceptions. These constituents must be checked first for their syntactic and semantic correctness before the whole constraint can be deployed along with the business process and properly evaluated and verified within the runtime environment.

6.1.1. Syntax Verification

We have developed a specialized parser to process the constraints coded in PCL and verify their syntactic correctness. Several alternative approaches are available for this purpose. The first one is to develop a hand-coded lexical analyzer and parser, as typically done for traditional general-purpose languages (e.g., C++/C, Java). The second approach is to use a parser generator, such as YACC and ANTLR. However, using this approach poses complications as it is difficult to modify the generated parser to fit our needs. The third approach, implemented in the Scala programming language, is to use an internal domain-specific language, which consists of a library of parser combinators [104] and maps them to the grammatical constructs of the target domain-specific language.

We have selected the last approach based on the combinator parsing as most suitable to our needs. Consequently, our PCL parser is almost directly based on its ENBF grammar and uses the
existing parser libraries provided by Scala. If a constraint contains lexical and/or syntactic errors (e.g. misspelling of PCL key words or missing definition of variable in Table 6.1), they are detected in this phase. All the aforementioned constituents of constraints written in PCL can be extracted from the specifications source code and translated to our internal constraint representations and later used in the subsequent verification steps.

Table 6.1. PCL syntax verification example -- missing definition of variable "a"

<table>
<thead>
<tr>
<th>constraint ***</th>
</tr>
</thead>
<tbody>
<tr>
<td>context t1: Task1, t2: Task2</td>
</tr>
<tr>
<td>pre distance(a.location, t2.location) &gt;50 mile</td>
</tr>
</tbody>
</table>

6.1.2. SEMANTIC VERIFICATION

After syntax verification, semantic verification is performed to ensure the correctness with respect to both the ontology and PCL definitions. Three types of semantic verifications are listed below:

- **Declarations**: The existence of ontological classes and entities is checked. SPARQL queries are used to ensure a match between the declarations in PCL and the classes in ProContO. If a declaration cannot be found in the ontology, it indicates that a mistake may exist on either side, i.e., an incorrect context declaration in the PCL specification or a missing class/property in the ProContO ontology. As the ontology grows larger and more complex, such mistakes will be hard to detect and consequently ignored at the design phase, because they can be syntactically correct and semantically consistent from the perspective of the ontology.

- **Navigability**: The validity of the navigation paths needs to be verified, too. A common mistake is for an ontological class to access a property that is not defined or cannot be
reached from the specified path. Both cases are illustrated in the highlighted statements in Table 6.2 and Table 6.3.

- **Reachability**: Reachability within the business process flow is more difficult to verify because it involves both constraint and process definitions. First, the constraint's invariants and pre/post-conditions along with business process definition are inspected and then the conjunction of an activity’s “inv ∩ post” and its subsequent activity’s “inv ∩ pre” is checked to determine whether the reachability is met. For complex systems, it is often impractical to perform a complete reachability checking for the constraints. At the static verification phase, the constraint attributes are not available yet and reachability verification falls within the scope of formal verification, which often involves the exploration of all the possible states of the constraint attributes. This formal verification approach is prone to the state explosion problem and runtime verification gains more population recently. However, in this stage, we focus on the constraint attributes that appear in two consequent activities and evaluate the conjunction expression.

Table 6.2. PCL semantic verification example -- no concept "location" declaration in ProContO

```
constraint ***
  context t1: Task1, t2: Task2
  pre distance(t1.geoLocation, t2.location) > 30 mile
```

Table 6.3. PCL semantic verification example -- non-navigable from "geoLocation" to "location" in ProContO

```
constraint ***
  context t1: Task1, t2: Task2
  pre distance(t1.geoLocation.location, t2.geoLocation)
```
It is worth noting that semantic verification is a very broad research area and these three cases (declarations, navigability and reachability) only partially address this issue. We plan to add more semantic checks in the future.

6.1.3. Service Specification Verification

When constraints are evaluated within the runtime environment, our framework must dynamically invoke various constraint operations. Therefore, the service specification of the constraint operation must be checked during the static verification phase, before the whole business process is deployed to the runtime environment.

As constraints are “first-class citizens” in our framework, the inputs and outputs of constraint operations must involve constraint attributes already defined in the process constraint ontology. The semantic Web services that implement these constraint operations also reuse the same semantic definition for their inputs and outputs. This approach is similar to [91], which adds semantic description to the input and output messages of Web services in order to enable automatic Web service composition. However, in our framework, constraint attributes serve not only as input and/or output parameters to a Web service, but are also used in constraint specifications and the subsequent verification steps.

Table 6.4. Service specification verification example – incomplete parameter list

<table>
<thead>
<tr>
<th>constraint ###</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>context</strong> t1: Task1, t2: Task2</td>
</tr>
<tr>
<td><strong>pre</strong> distance(t1.geoLocation) &gt;50 mile</td>
</tr>
</tbody>
</table>

The input constraint attributes used in the constraint operations defined in PCL are matched with those specified in the constraint operation ontology first. Then, the number and order of the input parameters are checked. Finally, the output constraint attributes are checked. A substantial
amount of research on Web service discovery and composition has been devoted to semantic annotation and mediation between Web service interfaces. However, our framework simply raises a verification error for a syntactic mismatch found between ontology and PCL declaration and does not involve Web service discovery and mediation. Once the service specification verification is finished, the whole procedure of static verification phase is complete.

6.2. Dynamic Verification

All of the verifications described in the previous section are performed before the business process is executed within the deployment environment, thus are classified as static verifications. The focus of this section is on dynamic or runtime verification. In order to address the main issue of enforcing the satisfaction of process constraints during runtime execution, our framework utilizes an underlying monitoring module and a runtime verifier. While the former collects contextual runtime information, the latter performs runtime verification and handles exceptions.

6.2.1. Context-aware Monitoring

After constraints have been deployed along with the business process in a heterogeneous SOA-based runtime, they are continuously evaluated against every property process state change. This is accomplished by the monitoring library, which is a key enactment module and vital to the successful deployment and execution of business workflows and web services compositions.

There are two alternative mechanisms for monitoring: inline and outline. Inline monitoring embeds the monitoring code or executable binary within the SUS and runs within the same CPU process. In order not to invade the code base of the actual implementation, aspect-oriented programming (AOP) [105] technique is often used and inline monitoring can provide most flexible monitoring solution with high efficiency. On the other hand, outline monitoring keeps tracking the SUS’s behavior from a different CPU process or even remotely. It provides not only context
information but also additional support for supervision and management on SUS. Such capability of supervision is very beneficial in exception handling. When some exception occurs, an outline monitor can even react to the situation at hand, where the specific exception handler cannot resolve the constraint exception. The obvious disadvantage for outline monitoring is that some overhead is introduced due to constant “heart-beat” operations. However, from the perspective of our context-aware monitoring module, we argue that both monitoring methods are complimenting each other, and a combination of the two enables us to better evaluate the constraint, understand the contextual information when one constraint is violated and, ultimately, handle the ensuing exception more properly.

With these requirements in mind, our proposed underlying monitoring module, shown in Figure 6.2, uses both inline and outline monitoring methods, which are implemented as parts of the online context-aware monitoring module. From the inline monitoring sub-module, we can obtain service execution information (e.g., constraint operations service, business processes), while from the outline monitoring sub-module, various contextual information not available from inline monitoring can be collected, e.g., service health checking, geo-location, and QoS measurement metrics. Also, an online verification algorithm is used to handle property change events so that the monitoring module does not rely on a static dataset.

Figure 6.2. Context-aware monitoring module
6.2.2. **Constraint Verification and Exception Handling**

After SUS’s contextual information is obtained from the underlying monitoring framework, dynamic verification is performed against the corresponding process constraints.

The very first step in dynamic verification is to populate ProContO with instances against the constraint definitions in the ontology. Such a procedure is important because the evaluation of simple constraint attributes and constraint operations needs up-to-date context information. It is easy to verify simple constraint attributes as a quantifiable property (e.g. response time). For the more complex computational constraint operations (e.g. distances between a tornado and its nearby hospitals), the actual implementation of the constraint operation is dynamically calculated and the output, as a constraint attribute, replaces the original constraint operation in PCL’s condition, which is represented as a Boolean expression.

Furthermore, dynamically populating the instances in the ontology is especially required for constraints that need to be inferred and reasoned against the ontology by a SWRL reasoner. The inference on SWRL rules can be undecidable if the named individuals are not bound [106]. As these named individuals will not be available until runtime execution, we should dynamically populate the ontology and this will let the related SWRL reasoner generate desired result. For example, a prerequisite constraint in an interview process, which is represented in PCL as shown in Table 6.5, indicates that no nepotism is allowed between interviewers and interviewees. Once an instance of the interview process starts for an applicant Mary Smith, the uncle-niece relationship between Mary Smith and the interviewee, John Smith, can be easily detected based on the SWRL rules shown in Table 6.6, as there exists an instance, Bob Smith, in the ontology and Bob is related to Mary by the father-daughter relationship, and Bob is the brother of John. This example demonstrates additional advantages of introducing ontology as the foundation in our constraint
representation model. It not only clearly defines the meaning of the relationships among the process constraint concepts during the static verification, but also supports inference and reasoning at runtime.

Table 6.5. Constraint of nepotism restriction in interview process

<table>
<thead>
<tr>
<th>constraint</th>
<th>NepotismNotAllowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>context</td>
<td>t1: JobApplication, t2: JobInterview</td>
</tr>
<tr>
<td></td>
<td>x: t1.performer, y: t2.performer</td>
</tr>
<tr>
<td>pre</td>
<td>not isRelative(x, y)</td>
</tr>
</tbody>
</table>

Table 6.6. SWRL rules for family relationship in ProContO

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasParent(?x1, ?x2) ∧ Man(?x2) → hasFather(?x1, ?x2)</td>
</tr>
<tr>
<td>hasParent(?x1, ?x2) ∧ hasBrother(?x2, ?x3) → hasUncle(?x1, ?x2)</td>
</tr>
<tr>
<td>Declaration(ObjectProperty(isRelative))</td>
</tr>
<tr>
<td>subObjectPropertyOf(hasFather, hasParent)</td>
</tr>
<tr>
<td>subObjectPropertyOf(hasParent, isRelative)</td>
</tr>
<tr>
<td>subObjectPropertyOf(hasBrother, isRelative)</td>
</tr>
<tr>
<td>subObjectPropertyOf(hasUncle, isRelative)</td>
</tr>
<tr>
<td>hasFather(Mary, Bob)</td>
</tr>
<tr>
<td>hasBrother(Bob, John)</td>
</tr>
</tbody>
</table>

Once all the needed constraint attributes are available, Boolean expressions are evaluated to determine whether the entire constraint is satisfied or not. When the dynamic verification is completed, then we can determine whether the constraints are satisfied and which exception handler to execute, if a constraint is violated. The exception handlers are implemented the same way as constraint operations, and can be dynamically invoked by the runtime verifier. It is worth noting again that the process constraint framework does not depend on any specific definition language (e.g., BPMN, BPEL, etc.) or process execution engine. Therefore, when a certain constraint is violated, the verifier first invokes the corresponding user-defined exception handler. If the constraint cannot be satisfied after the execution of the exception handler, runtime verifier interacts with the exception handling mechanism of the workflow process execution engine for
compensation. For example, it can signal an intermediate event to the BPMN engine or throw an application fault to the BPEL engine, so that the concrete execution engine can react to such event or fault and invoke the corresponding fault handling within the business process. If no defined exception handler can be found in the PCL specification, the runtime verifier uses the default built-in exception handler, which logs the contextual information and directly interacts with the concrete workflow execution engine for compensation.
CHAPTER

7. PROTOTYPE IMPLEMENTATION

7.1. SERVICE COMPONENT ARCHITECTURE (SCA) AND ENTERPRISE INTEGRATION PATTERNS

7.1.1. SERVICE COMPONENT ARCHITECTURE (SCA)

Since the introduction of service-oriented architecture (SOA), it becomes widely accepted in both industry and academia. It not only provides a standard approach to register/discover/invoke Web services, which abstracts business logic by making it less dependent on the actual implementation language. Nowadays, Java, C++, C#, PHP, or Python can all be used as programming languages in implementing Web services. Although SOA developers can consume Web services without their implementation details, they still need to consider numerous details of the endpoints, bindings and protocols when composing Web services. Faced with this common obstacle, the Service Component Architecture (SCA) specification has been proposed to address this issue in order to maintain programming language and application environment neutrality.

OASIS defines SCA as “a vendor-, technology-, language-neutral model for the creation of business systems using SOA by the composition and deployment of new and existing service components” [107]. For the purpose of maintaining programming language and application environment neutrality, SCA supports various interface, service implementations, protocols and binding possibilities [108], including:

- **Interface**: WSDL port type, Java interface
- **Service/component implementation**: Java class, BPEL process, BPMN, Python, Spring
- **Binding**: JMS, Web Service, RMI/IIOP, SOAP, REST
• **Protocols**: HTTP, HTTPS, GET, POST, FTP

SCA comes into play by hiding the details of interface, implementation, binding and protocols. Built upon existing SOA standards and specifications, SCA aims at promoting SOA to the next level. It abstracts the endpoints of the services and hides protocol implementation from SOA developers. The building blocks provided in SCA are listed below [108]:

• **Composite** - A composite is used to assemble SCA elements into logical groups and contains a set of components, services, references and the wires that interconnect them, plus a set of properties.

• **Component** - A Component is the basic element of business function in an SCA assembly; components are combined into complete business solutions by SCA composites.

• **Service** - A Service represents an addressable interface of an implementation.

• **Reference** - A reference represents a dependence that an implementation has on a service provided by another component.

• **Promotion** - Promotion is used to connect different components across composites.

• **Wire** – A Wire is used to connect different components within a composite.

A simple SCA Hello World example is illustrated in Figure 7.1. The whole process uses the SwitchYard tooling within Eclipse IDE. The whole process includes a simple routing service and a Java bean, both exposed as SCA services. The Java bean is a Hello World example. The Camel routing service acts as a service broker to the Java bean. It receives an HTTP request, routes it to the referenced Java bean implementation, and responds back to the requester.
This simple example illustrates the SCA’s powerful capability of hiding implementation and language details; this feature has been widely utilized in our PCF framework. The SCA composite includes two SCA services, each backed by an SCA component, respectively. Although the referenced SayHello service is implemented as a Java bean in this example, it can be a BPEL process, BPMN process or a Drools-based process, as long as each one is promoted as an SCA service. In the meantime, the HTTP binding is just one of the many supported bindings and the others may include SOAP, REST, FTP, JMS (Java Message Service), etc. In summary, an SCA service has several advantages over the traditional plain services:

- SCA services can be easily integrated with other SCA services and thus the service composition including many processes defined by various process definition languages is feasible. For example, in SCA architecture, the composition of a BPMN process and a BPEL process is achievable.

- SCA services are under management of SCA runtime environment, which has a built-in service registry, version control, support of monitoring and auditing.

- SCA service is enriched with additional powerful features, e.g., built-in transformer, interceptor and validator.
• SCA service can be easily integrated by other remote applications that are compatible with SCA specification via SCA reference concept.
• SCA services are cluster-ready, which means they already consider the issues of scalability and fault tolerance.

Given all the benefits provided by the SCA architecture, it has been a natural step for our proposed PCF framework to adopt the SCA principles. From our perspective, PCF aims at providing a general constraints model without relying on a specific process definition language. It not only requires the neutrality of process definition details (e.g., definition language, programming implementation, etc.), but also requires an easy approach to integrate remote constraint operations with various business activities. SCA specification addresses these two features properly. Built on top of the SCA architecture, the PCF framework is capable of handling a variety of processes and their specific runtime environments, as well as the integration of constraint operations into the associated processes.

Several open source SCA implementations are available, e.g., Apache Tuscany and JBoss SwitchYard. Oracle and IBM also integrate SCA specifications into their own application server offerings, namely, the Oracle SOA Suite and the WebSphere Process Server. We have chosen the SwitchYard project from JBoss Inc., largely because it is open source, under active development and well documented. According to the latest stable version v1.1 (at the time of this writing), it supports business process management and service orchestration of BPMN 2, BPEL 2.0, and some other features, e.g., Camel routing, Drools rule engine. In our PCF framework, SwitchYard is used as the underlying process engine for Camel routing, BPEL workflow engine and BPMN process engine. It controls the development, deployment and execution of workflows. Furthermore, the
PCF framework utilizes SwitchYard’s SCA technology to enable the interaction between a workflow and its associated constraint operations.

### 7.1.2. Camel and Enterprise Integration Patterns

Camel\(^2\) is the kernel of SwitchYard runtime environment. It is designed to address the service integration functionality in the distributed computing environment and already implements most of the enterprise integration patterns. The SwitchYard runtime integrates it and achieves two goals: (i) provides gateway bindings for services and references; Camel endpoints can function as protocol adapters and expose services hosted in SwitchYard; (ii) provides a robust, easy-to-configure, and feature-rich routing engine. Camel routes can be defined using Java DSL or XML and deployed within SwitchYard to handle pipeline orchestration between SwitchYard services\(^{[109]}\). Exposing SwitchYard as a protocol adapter, e.g., exposing a JMS service as a SOAP Web service, is a very useful feature, however, our focus has been on Camel’s seamless support of Enterprise Integration Patterns.

Enterprise Integration Patterns (EIP) is defined as a “a consistent vocabulary and visual notation to describe large-scale integration solutions across many implementation technologies” in\(^{[110]}\). These patterns are proposed to address the various scenarios occurring in the service integration domain. The PCF framework utilizes three patterns extensively: (i) wiretap, (ii) message broker and (iii) content-based router in order to achieve the purpose of context-aware monitoring and integration of process, constraint operations and exception handlers. By combining these three patterns, PCF introduces minimal invasion to the existing process definition language and integrates remote constraint operation with ease. In the following sections, the three patterns will be discussed in greater detail.

---

The **wiretap pattern** introduces additional message channel to consume and inspect all messages that travel across the channel. It does not interfere with the normal execution flow and makes it suitable for the monitoring purpose.

![Diagram of wiretap pattern](image)

**Figure 7.2.** Enterprise integration - wiretap pattern [110]

The **message broker pattern** is a common approach to hide implementation details from the originating application and perform some specific routing logic. From the point of view of the originating application, the target service remains the same while the actual implementation may have been changed based on some requirements. In our PCF framework, the interaction of constraint operations is hidden via message broker pattern.

![Diagram of message broker pattern](image)

**Figure 7.3.** Enterprise integration - Message broker pattern [110]
The **content-based router pattern** is used to route messages to the correct recipient based on the message content. A common use case is to perform different operations based on the given input. For example, the execution flow will be routed to a pdf processor or doc processor depending on the document type. In our PCF framework, the condition of content-based router is determined by the constraint operation. If the constraint operation is satisfied, the request is routed to the original service; otherwise, it is routed to the pre-defined exception handler or aborted if no exception handler is specified.

![Figure 7.4. Enterprise integration - Content-based pattern](image)

By adopting the SCA principles and following the enterprise integration patterns throughout the whole PCF framework, the framework has achieved two goals: (i) successful constraints integration into the process engine, and (ii) keeping the interference of the actual process execution flow to the minimum. The service template is illustrated in Table 7.1. The template integrates together the original services, the constraint operation and the exception handler. Thanks to the power of the SCA architecture, this template is general enough to integrate any services or tasks that are compatible with the SCA specification, without being limited to a specific process definition language. In the PCF framework, this integration template is used for any services/tasks that have a constraint associated with them. The three motivating examples have been implemented by utilizing this template and will be discussed in more detail at the end of this chapter.
Table 7.1: Service Integration Template for PCF framework

```java
from("entry point")
.log("Received message for 'entrypoint' : ${body}")
.setProperty("originBody", body())
// Send context information to context monitor using wiretap pattern
.log("wiretap to context monitor")
.wireTap("switchyard://ContextMonitorInvoker")
// Apply content based router pattern and invoke constraint operation
.log("Invoke constraint operation...")
.to("constraint operation")
// Content-based router
.choice()
 .when(body().contains("true"))
  .log("go to original service........")
  .setBody(property("originBody"))
  .to("original service")
.otherwise()
 .log("constraint operation failed!!!!")
 .to("exception handler");
```

If a constraint operation is related to a certain SCA service, a new Camel routing component is added to serve as a broker or proxy to the original SCA service. Following the service integration template mentioned in Table 7.1, the whole process of linking a constraint operation with a common SCA service is illustrated in Figure 7.5. A monitoring service can be easily connected with the underlying process via wiretap pattern, and no side-effects will be introduced.

In order to achieve runtime verification with minimum intrusion into the existing processes and their involved activities, a combination of the message broker pattern and content-based router pattern is used. A routing service has been added via the message broker pattern to the original service that requires constraint validation before the actual service invocation. If a post-condition is needed, the constraint can also be verified by adding it to the end of the service execution. Then, the content-based router pattern has been applied and the router to determine if to route to the original service or the specified constraint exception, based on the result of the constraint operation.
7.2. ARCHITECTURE OF THE PCF FRAMEWORK

The architecture of our PCF framework is illustrated in Figure 7.6 and Figure 7.7. The major components include the PCL parser, semantic verifier, operation service verifier, context-aware monitor, runtime verifier, and the exception handler. PCF has been coded mainly in the Java programming language, while the Scala programming language has been used to develop the PCL parser. Because Java can easily interact with Scala classes, the Scala and Java code have been combined to take full advantage of the two languages. The rest of the PCF framework leverages SOA and the JavaEE suite provided by the JBoss Enterprise Application Platform (EAP).
The PCF framework is built upon the concept of SOA (service-oriented architecture) and it adopts the major principles in SCA (Service Component Architecture). All the involved entities in the PCF framework, which can be generally divided into processes and constraints, e.g., the processes (defined in BPEL and/or BPMN) and their constituent task/activities, constraint operations and exception handlers, all begin as simple SCA components, which are then promoted to SCA services to facilitate their easy composition and integration.
Figure 7.7. Process constraint framework prototype architecture – Example illustration

On the process side, the processes (e.g., Camel route, BPEL, BPMN) and its constituent tasks/activities (e.g., Web services and REST Web services) can be integrated within the SCA runtime engine with ease. All the tasks/activities are first developed as SCA components and then promoted to SCA services. They can be composed into various processes or workflows, e.g., BPMN or BPEL, and referenced as SCA references.

On the constraints side, the context-aware monitor, constraint operations and exception handlers within the PCF framework are also promoted to SCA services and wired to form a process, following the service integration template. This does not interfere with normal execution flow or pose any modification to the existing service or task implementation. The previous approaches included writing the customized code using instruction interceptors or aspect-oriented programming techniques. However, these techniques are too specific and tied to certain programming language (e.g., Java or C++) and if the implementation were to be changed, the corresponding implementation of constraint operations would have to be modified, as well. Rather than focusing on the level of the executable binary code, we have concentrated solely on the service...
level. By adopting the SCA principles, our PCF framework is capable of addressing a wider range of service integration situations.

The major components in the PCF framework include (i) the ProContO ontology and its management implementation, (ii) the PCL language, (iii) the Context-aware monitor and (iv) the static and dynamic verifiers for constraint handling.

7.2.1. ProContO Ontology Management Implementation

OWL API has been used to navigate and query the ProContO ontology. The ProContO ontology is easy to extend and incorporate domain-specific knowledge. ProContO keeps track of the context information and serves as a knowledge base for constraint operations. Furthermore, a semantic reasoner can also be utilized to perform semantic inferencing in constraint verification. Consequently, process constraints become more expressive and capable to address more complex constraints than the typical quantifiable requirements (e.g., quality of service matrix) used in many research papers.

Due to the easy extensibility of the ontology, a domain-specific ontology can be incorporated in the ProContO ontology with ease. In the motivating example of the job interview, the nepotism ontology as been imported into ProContO, as shown in Figure 7.8; the whole ontology is shown in Appendix B. The isRelative constraint is defined as a disjunction (OR) of relationships, namely, hasParent, hasChild, hasSibling, hasAunt, hasUncle, hasNephew, hasNiece, and hasCousin. Corresponding SWRL rules have been defined, as shown in Table 7.2. Constraint operations can query ProContO for the names of interviewer and interviewee first and then invoke the Pellet reasoner to discover the relationship between the interviewer and interviewee and determine if these two persons are related or not. The capability of performing semantic reasoning
by constraint operations makes PCF more suitable for expressing more complex constraints than when using the existing approaches.

![Nepotism ontology](image)

**Figure 7.8: Nepotism ontology**

**Table 7.2: Nepotism SWRL rules**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Man(?y), hasChild(?x, ?y) -&gt; hasSon(?x, ?y)</td>
<td></td>
</tr>
<tr>
<td>hasParent(?x, ?y), hasSister(?y, ?z) -&gt; hasAunt(?x, ?z)</td>
<td></td>
</tr>
<tr>
<td>hasParent(?x1, ?y1), hasParent(?x2, ?y2), hasSibling(?y1, ?y2) -&gt; hasCousin(?x1, ?x2)</td>
<td></td>
</tr>
<tr>
<td>Woman(?y), hasSibling(?x, ?y) -&gt; hasSister(?x, ?y)</td>
<td></td>
</tr>
<tr>
<td>Man(?y), hasParent(?x, ?y) -&gt; hasFather(?x, ?y)</td>
<td></td>
</tr>
<tr>
<td>Man(?z), hasChild(?y, ?z), hasSibling(?x, ?y) -&gt; hasNephew(?x, ?z)</td>
<td></td>
</tr>
<tr>
<td>Woman(?y), hasParent(?x, ?y) -&gt; hasMother(?x, ?y)</td>
<td></td>
</tr>
<tr>
<td>Man(?y), hasSibling(?x, ?y) -&gt; hasBrother(?x, ?y)</td>
<td></td>
</tr>
<tr>
<td>hasBrother(?y, ?z), hasParent(?x, ?y) -&gt; hasUncle(?x, ?z)</td>
<td></td>
</tr>
<tr>
<td>Woman(?y), hasChild(?x, ?y) -&gt; hasDaughter(?x, ?y)</td>
<td></td>
</tr>
</tbody>
</table>
7.2.2. PCL Language Processor

The syntax verification is the very first step in the static verification and thus a parser is required. The PCL parser is based on an approach called parser combinatory and implemented in the Scala programming language. It utilizes the existing simple parsers provided by Scala (e.g., JavaTokenParsers, RegexParsers, etc.) and combines them to parse a more complex process constraint language expressed in EBNF. A small portion of the parser implementation is listed in Table 7.3 and illustrates how the individual parsers (e.g., ident, context, conditions) have been combined together to form a complete PCL parser. The full implementation can be found in Appendix C.

Table 7.3. PCL parser implementation snippet in Scala

```scala
def pcl: Parser[PCLConstraint] =
PCLConstraint.Keyword_constraint ~> ident ~ context ~ conditions ~
  opt(exception_handler_clause) ^^ {
    case constraintName ~ context ~ conditions ~ exceptionHandler => {
      val result = new PCLConstraint(constraintName, context, conditions,
        exceptionHandler)
      result
    }
  }

def context ...

def conditions ...

def exception_handler_clause ...
```

With the use of the PCL parser, process constraints written in PCL are first parsed into a corresponding Java object model. Its UML diagram is shown in Figure 5.1. The subsequent static verification will be discussed in Section 7.2.4.

Assuming the static verification has been completed, the PCL processor will continue to collect information from multiple involved parties: the constraint language, process definition.
language, and service definition language (e.g., WSDL). From the constraint language, it collects the information for the constraint operation and the associated process task/activity. From the SwitchYard process configuration (switchyard.xml), it first collects the SCA service definition and then generates an additional Camel routing in Java. The mapping between the constraint language and the process definition is straightforward by following the service template specified in Section 7.1.2. There is no need to modify the original service implementation or process definition.

7.2.3. CONTEXT-AWARE MONITOR AND EXCEPTION HANDLER

The context-aware monitor provides the knowledge base for both dynamic verification and exception handling. We argue that a useful monitor needs to fully observe the runtime environment and understand the internal program context with the help of both inline and outline monitors. In order to achieve such context-awareness, our monitor extensively utilizes messages and a series of remote monitors have been implemented as SwitchYard services (i.e., compatible with SCA specification) and deployed along with the constraint operation services within the process runtime engine (via a wiretap pattern). Before dynamic verification can be executed, each involved activity is responsible for sending out the context information to the context-aware monitor, which updates the corresponding fields in ProContO. Therefore, the constraint operations can query ProContO for the most up-to-date contextual information and determine whether to execute the original service.

An exception handler is integrated into the process using a similar approach as a constraint operation. When the constraint is not satisfied and an exception handler is specified, it is invoked as specified in the Camel routing. The implementation of the exception handler is dependent on the process engine. For BPEL, it addresses how to compensate for the exception and how to link the exception handler defined in PCL with the compensation in BPEL.
7.2.4. **Static Verification and Dynamic Verification**

The PCF framework performs static verification at the deployment time and dynamic verification, namely, the enactment and enforcement of constraints at runtime.

Static verification is performed after the constraint language has been parsed and before process/workflow is deployed into the runtime production environment. Because the PCL language processor collects the information from all the involved parties, it understands not only the constraint side, but also the process side. Several static verifications can be considered, e.g., syntax verification, semantic verification and service specification verification. Our current implementation mainly addresses the syntax verification. Other static verification will be added in the future.

Dynamic verification is performed in the runtime environment. ProContO serves as a knowledge base and is updated by the context-aware monitor. A constraint operation is integrated as an SCA service and extracts the contextual information as input from ProContO. The result of the constraint operation is used as the input to the content-based router, which determines whether the control flow will continue within the original service or the exception handler.

7.3. **Processes and Constraint Implementation**

The proof-of-concept applications for the three motivating examples have been implemented using the SwitchYard tooling: (i) the Job Interview process has been defined as a BPMN process, (ii) the emergency reaction workflow has been defined using the BPEL process definition language, and (iii) the glycomics workflow has been defined using Camel routing. The implementations are discussed in more detail in the following sections.
7.3.1. JOB INTERVIEW PROCESS IMPLEMENTATION

The Job Interview process is defined in BPMN, as shown in Figure 7.9, and it is implemented in SwitchYard, as shown in Figure 7.10. Job application service task takes an interviewee as input and chooses an interviewer as output. Job interview service task conducts the actual interview procedure and gives the result of whether the interviewee has passed the interview. The two service tasks are implemented as SwitchYard service tasks, which are extension widgets of the standard BPMN task and can be consumed by BPMN process directly. The JBoss BPM runtime supports the BPMN2 specification and is integrated in the SwitchYard runtime as a BPMN process engine.

![Figure 7.9. Job interview process in BPMN representation](image)

One of the common requirements during an interview process is to avoid nepotism. Our approach to tackle this issue is to use a constraint, namely, the pre-condition in the JobInterview task is specified as interviewer and interviewee cannot be relatives, as listed in Table 6.5.

By integrating the constraint operations, the whole BPMN process in SwitchYard is an SCA service and references the JobApplication task as a starting point. Because the JobInterview service task is related to a constraint, its task is configured as a Camel routing service. Following the previously specified integration template, the updated and augmented JobInterview service task is, in fact, a message broker service, and uses the same interface as the original JobInterview service. The constraint operation (NepotismChecker), context-aware monitor and exception handler (FailedConstraintOperationLogger) are all SCA services and integrated together using the
Enterprise Integration Patterns implemented in Camel routing. The detailed Camel route is shown in Table 7.4

Table 7.4: Camel Routing in JobInterview service task

```java
from("switchyard://JobInterview")
 .log("Received message for 'JobInterview' : ${body}")
 .setProperty("origInBody", body())
 // Send context information to context monitor using wiretap pattern
 .log("wiretap to context monitor")
 .wireTap("switchyard://ContextMonitorInvoker")
 // Apply content based router pattern and invoke constraint operation
 .log("Invoke constraint operation...")
 .to("switchyard://NepotismCheckerInvoker")
 .choice()
 .when(body().contains("false"))
  .log("go to original job interview.......")
  .setBody(property("origInBody"))
  .to("switchyard://JobInterviewService")
 .otherwise()
  .log("constraint operation failed!!!!")
  .to("switchyard://FailedConstraintOperationLoggerInvoker");
```

NepotismChecker queries ProContO and retrieves the names of the interviewer and interviewee, then performs semantic reasoning using Pellet and OWL API. The capability of
performing semantic reasoning illustrates the strength of the PCF framework in handling more complex constraints.

7.3.2. TORNADO EMERGENCY REACTION PROCESS IMPLEMENTATION

Tornado Emergency Reaction Workflow is defined using the BPEL process definition language. It consists of four services, namely, TornadoDetection, TornadoIssueAlarm, HospitalPreparer, and ShelterActivator. After a tornado is detected, the other three services will be triggered in parallel to achieve fast reaction and prompt response. The constraint or the requirement is that before notifying a hospital, it should not be close to the Tornado’s location. A BPEL process is depicted in Figure 7.11 and its implementation in SwitchYard is shown in Figure 7.12.

![BPEL process for emergency reaction workflow](image)

Figure 7.11. BPEL process for emergency reaction workflow
Following the service integration template, the integration procedure remains the same, an augmented Camel route is created, which serves as a message broker to the original service. The context-aware monitor is integrated via the wiretap pattern, and the constraint operation (DistanceChecker) is integrated as the entry point to the content-based router pattern. If the DistanceChecker returns a true result, the original request message is routed to the original HospitalPreparer service. Otherwise, the control flow is processed by exception handler (FailedConstraintOperationLogger in this case).

7.3.3. **Glycomics Scientific Workflow Implementation**

Glycomics Workflow involves four services, namely raw data producer, data processor, simulation optimization performer and result visualizer. The requirement is that the data processor service should be able to select a fast or slow candidate service dynamically, based on the size of the input raw file. If the file is too large, it should pick a fast candidate service. This service selection constraint is represented in PCL as an if-then-else clause. Meanwhile, the service selection is feasible in the service integration template as it is natively supported by the content-
based router pattern. The process definition is shown in Figure 3.3 and the SwitchYard implementation is shown in Figure 7.13.

The Glycomics Workflow is implemented based on Camel routing services\(^3\). As mentioned earlier, Camel allows us to define routing and mediation rules in a variety of domain-specific languages and includes implementation of most enterprise integration patterns. On one hand, Camel has been extensively used in SwitchYard to provide a routing service among various SCA services due to its great support of enterprise integration patterns. On the other hand, Camel also supports service orchestration and is great for implementing the data flow. Therefore, Camel routing has been utilized in the PCF framework to develop the Glycomics Workflow. The workflow follows a common pipeline pattern and the process definition, written in the Camel Java domain-specific language, is listed in Table 7.5.

---

\(^3\) http://camel.apache.org
Table 7.5. Glycomics workflow in Camel

| from("switchyard://GlycomicsWorkflow")  |
| .log("Received message for 'GlycomicsWorkflow' : ${body}" ) |
| // Invoke raw data producer service  |
| .to("switchyard://RawDataProducer")  |
| .log("created raw file with size ${body}G") |
| // Invoke data processor service  |
| .to("switchyard://DataProcessorService")  |
| // Invoke SimulationOptimizationExecutor service  |
| .to("switchyard://SimulationOptimizationExecutor")  |
| // Invoke ResultVisualizer service  |
| .to("switchyard://ResultVisualizer")  |
| .log("finish glycomics workflow and quit.....") |

By applying the template shown in Section 7.1.2, the constraint operation is integrated into the Glycomics Workflow process. The corresponding Camel route is listed in Table 7.6. The whole integration remains the same as in the previous two motivating examples, and the only difference is the content-based router, which is used to consider the service selection in this case, based on the result from the constraint operation (FileSizeChecker).

Table 7.6. Camel route for data processor service

| from("switchyard://DataProcessorService")  |
| .log("Received message for 'DataProcessorService' : ${body}" ) |
| // Send context information to context monitor using wiretap pattern  |
| .log("wiretap to context monitor")  |
| .wireTap("switchyard:///ContextMonitorInvoker")  |
| // Apply content based router pattern and invoke constraint operation  |
| .log("Invoke constraint operation....")  |
| .to("switchyard:///FileSizeCheckerInvoker")  |
| .choice()  |
| .when(body().contains("true"))  |
| .log("go to slow data processor.....")  |
| .to("switchyard:///SlowDataProcessorInvoker")  |
| .otherwise()  |
| .log("go to fast data processor.....")  |
| .to("switchyard:///FastDataProcessorInvoker"); |
CHAPTER

8. EVALUATION

This dissertation presents PCF, a semantic end-to-end process constraint modeling framework. The framework has been evaluated from two perspectives. First, to assess the quality and validity of our proposed PCF framework, we have used the ProContO ontology as a knowledge base and adopted SCA principles to implement the proof-of-concept applications for the three motivating scenarios (presented in Chapter 3) using the Switchyard tooling and JBoss Enterprise Application Platform. Second, to evaluate the performance overhead concerns as a consequence of introducing constraint operations, we have measured the response time of tasks and the associated constraint operations in runtime deployed in a local area network to simulate a distributed computing environment.

8.1. QUALITY EVALUATION

The quality of PCF framework is discussed in the following sections:

1. Powerful modeling capability of process constraints:

Most of the previous research on requirement and constraint modeling has been focused on the quantifiable constraints, e.g., response time, latency, throughput etc. Furthermore, without the help of an ontology, it was impossible to conduct reasoning based on the contextual runtime information. On the one hand, the PCF framework supports the general modeling of various kinds of constraints, from non-functional (e.g. Glycomics workflow example) to domain-specific ones (e.g., emergency reaction example). On the other hand, the wide adoption of ontology and semantic technique in PCF framework contributes to increased expressiveness and extensibility to
accomplishing a general constraint model, an abstract meta-model for general process definitions and the capability to model and create more complex constraints than just quantifiable ones.

Being capable of working at a high-level of abstraction of processes and their constraints enables us to model various process components and process constraints without worrying about the underlying process definition languages or runtime execution platforms.

Based on the proposed general constraints model, a process can be represented by the ontological concept of ProcessElement and the following aspects can be addressed in the constraint and further verified in the runtime: performer, activity and data. This powerful representation feature enables us to address more complex constraints than most previous research approaches that only focused on the QoS metrics or quantifiable constraints, i.e., the data aspect in our process model. The constraints on the process elements can be represented in PCL and they can be used separately or combined with other process elements. The process elements and their related constraints of the three motivating examples are listed in Table 8.1.

<table>
<thead>
<tr>
<th>Process</th>
<th>ProcessElement</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Interview process</td>
<td>Performer</td>
<td>Interviewee and interviewer cannot be relatives</td>
</tr>
<tr>
<td>Emergency reaction</td>
<td>Data</td>
<td>Distance between tornado and hospital</td>
</tr>
<tr>
<td>workflow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Glycomics workflow</td>
<td>Data, Activity</td>
<td>Based on file size, various services will be selected</td>
</tr>
</tbody>
</table>

2. An end-to-end constraint modeling and verification framework

Most of the previous research on requirements and constraints has been focused on the constraint modeling phase and has not addressed the issues of constraint deployment or verification, not to mention the integration with the underlying process runtime engine. However, the PCF framework covers the whole life cycle of a constraint from the design phase to runtime execution,
including potential constraint-related error handling. As illustrated in Figure 8.1, the life cycle of a process constraint overlays the life cycle of process development. It accelerates the evolution of processes in various phase of the whole life cycle. In particular:

(i) It facilitates the design of constraints alongside with the process definitions, which makes the model more accurate in the first place.

(ii) The static verification before deployment, especially the syntax verification helps detecting errors in constraints written in PCL.

(iii) The enforcement of constraints within the runtime, during the execution phase enables the detection of constraint violations based on contextual runtime information, and exception handlers can be integrated with ease. Finally, in the future, it will enable workflow optimization based on the constraint verification results.

3. The support of multiple process platforms and implementation service languages

Following SOA principles, PCF framework is capable of integrating Web services developed in various programming languages and protocols. Furthermore, adopting SCA principles makes the PCF framework independent of specific process definition languages and runtime environments. Consequently, the proposed PCF framework can map the constraints to various process definition and execution platforms, such as BPEL, BPMN and Camel routing.
4. A powerful service integration template

Most of the previous research on dynamic verification has been conducted based on a stand-alone server, unable to scale to a distributed environment. In order for verification code to be executed by the runtime, it usually has been either hard-coded within the actual business logic or aspect-oriented programming has been utilized to instruct the binary executable file. The first approach has not been viable or portable. The second approach has not been intrusive, but not widely supported.

Following enterprise integration patterns, we have proposed a service integration pattern and successfully applied it to the three motivating examples. The emergency reaction workflow and job interview process use a quantifiable constraint and a constraint that requires semantic reasoning, respectively, while the glycomics workflow uses a constraint that specifies service
selection. Although these constraints differ, all four parties related to a constraint definition, including the original task/activity, constraint operation, the context-aware monitor and the exception handler, are “glued” together using the same integration template based on the Camel routing provided in the SwitchYard tooling. This approach focuses on the service level and is built upon industry specifications (SCA and SOA), and thus it is widely supported by various software vendors. Furthermore, it is not intrusive and does not change the service implementation.

8.2. PERFORMANCE EVALUATION

One of the major concerns on dynamic (runtime) verification is its performance, an issue which is more difficult to address in a distributed environment. In our PCF framework, in order to connect a plain task/activity associated to a specific constraint within a runtime system, we have leveraged the service integration template, as well as three enterprise integration patterns, namely, message broker, wiretap and content-based router.

The overheads of introducing constraint operations and applying the integration patterns are as follows:

(i) The performance overhead of initializing a message broker and routing the message is low and can be ignored, in practice.

(ii) The execution of the wiretapped branch is in parallel with the normal process flow and poses no performance impact to the normal execution of the original service.

(iii) The performance overhead of content-based router is solely dependent on the execution time of the constraint operation and the time used in routing the message to the constraint operation and original service can be safely ignored.

(iv) Under certain circumstances, heavy load placed on the system may lead to many unpredictable situations, e.g., the system may fail to allocate the message broker or
it may take longer to route messages to the desired endpoint. However, this is due to the lack of system resources; these issues go beyond the scope of our research.

In summary, time used in message routing and service integration can be ignored safely and the performance overhead introduced by the PCF framework is largely dependent on the execution time of constraint operations. Some constraint operations may have very low operating cost, e.g., the determination of a file size or computation of the geospatial distance between two geo-locations (e.g., hospital and tornado), while other constraint operations may have a much higher computational cost. For example, some algorithms used in semantic reasoning may be in the N-EXP Time class, and thus, the reasoning on nepotism between an interviewer and interviewee can easily takes quite a long time to finish or even fail to produce a result with large size of the ontology. We believe that the overhead introduced by executing constraint operations is acceptable in view of the advantages of incorporating constraints within processes. Especially for the long-running processes or tasks, e.g., human interview tasks or simulation and optimization activities, these advantages are more apparent and certain level of performance overhead is regarded as acceptable.

In order to evaluate the performance of the PCF framework, we simulated the three motivating examples and choose a normal distribution as a distribution for constraint operations’ and tasks’ response time. The normal distribution is defined as \( f(x, \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \) where \( \mu \) is the mean or expected value and \( \sigma \) is the standard deviation. After estimating the mean and standard deviation for a normal distribution to be used as a simulation of service and constraint operations’ response times, the three processes have been executed 10,000 times to collect the average response time and determine the impact of adding constraint operation to the whole process’s execution time. The results are shown in Table 8.2 (units are in seconds).
<table>
<thead>
<tr>
<th>Processes</th>
<th>Tasks</th>
<th>Mean (μ)</th>
<th>Standard deviation (σ)</th>
<th>Average response time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Job interview</strong></td>
<td>Job application</td>
<td>4</td>
<td>2</td>
<td>3.52</td>
</tr>
<tr>
<td></td>
<td>Job interview</td>
<td>10</td>
<td>3</td>
<td>9.40</td>
</tr>
<tr>
<td></td>
<td>Nepotism checker</td>
<td>2</td>
<td>1</td>
<td>1.52</td>
</tr>
<tr>
<td></td>
<td>Failed Constraint Operation Logger</td>
<td>1</td>
<td>1</td>
<td>0.638</td>
</tr>
<tr>
<td><strong>Emergency reaction</strong></td>
<td>Tornado Detector</td>
<td>3</td>
<td>2</td>
<td>2.57</td>
</tr>
<tr>
<td></td>
<td>Shelter Activator</td>
<td>7</td>
<td>3</td>
<td>6.49</td>
</tr>
<tr>
<td></td>
<td>Hospital Preparer</td>
<td>5</td>
<td>2</td>
<td>4.46</td>
</tr>
<tr>
<td></td>
<td>Tornado Alarm Issuer</td>
<td>8</td>
<td>4</td>
<td>7.40</td>
</tr>
<tr>
<td></td>
<td>Distance Checker</td>
<td>2</td>
<td>1</td>
<td>1.54</td>
</tr>
<tr>
<td></td>
<td>Failed Constraint Operation Logger</td>
<td>1</td>
<td>1</td>
<td>2.83</td>
</tr>
<tr>
<td><strong>Glycomics workflow</strong></td>
<td>Raw Data Producer</td>
<td>10</td>
<td>5</td>
<td>9.50</td>
</tr>
<tr>
<td></td>
<td>Result Visualizer</td>
<td>3</td>
<td>3</td>
<td>2.65</td>
</tr>
<tr>
<td></td>
<td>Simulation Optimization Executor</td>
<td>8</td>
<td>4</td>
<td>7.63</td>
</tr>
<tr>
<td></td>
<td>Fast Data Processor</td>
<td>4</td>
<td>2</td>
<td>3.56</td>
</tr>
<tr>
<td></td>
<td>Slow Data Processor</td>
<td>20</td>
<td>10</td>
<td>19.67</td>
</tr>
<tr>
<td></td>
<td>File Size Checker</td>
<td>2</td>
<td>1</td>
<td>1.52</td>
</tr>
</tbody>
</table>

Based on data shown in Table 8.2, we can see that the services’ actual average response time basically matches the expected value of their specific normal distribution. The performance overhead for the processes is the constraint operations, namely, nepotism checker for job interview
process, distance checker for emergency reaction workflow, file size checker for Glycomics workflow, respectively.

The overall response time of the augmented services is roughly the sum of the constraint operation plus its own execution time. It is clear that the constraint operation causes the slow execution of the augmented service due to the extra communication and execution time for constraint operations. However, the added benefits brought by dynamic verification are also apparent:

(i) Process models incorporating constraints are considered to be more accurate. For example, without nepotism checker, there will no way to prevent nepotism in an interview process and unqualified interviewee may be recruited without being noticed.

(ii) It is more efficient from the point of view of the whole life cycle and the evolution of workflow processes as discussed in Figure 8.1. Service and constraint can work and evolve side-by-side without interfering with each other.

(iii) In many real world cases, the original activity/task is a long-running task itself, e.g., one hour for job interview or two hours for simulation optimization. Therefore, the performance loss in constraint operation can be neglected.

In summary, certain level of performance loss is expected, but it can be regarded as acceptable in view of the added benefits of incorporating constraints.
9. CONCLUSIONS AND FUTURE WORK

9.1. SUMMARY AND DISCUSSION

In the research area of software engineering, requirements and constraints are always an important research topic and have attracted considerable attention from both industry and academia. UML and OCL are two mature examples for modeling software systems and constraints from an object-oriented perspective. However, since OCL has not been designed as an executable specification language, enforcing OCL constraints at runtime remains a big challenge.

With the fast advancement of service-oriented architecture and the great need of distributed computing, more and more software systems are built based on the SOA standards and specifications. Several process definition languages and process engines have gained wide recognition, e.g., BPEL and BPMN. However, the exact same issue remains yet to be addressed when serious consideration is taken to add constraints to a process in a distributed environment, namely, how to model process constraints in the design phase and then verify and enforce them in a distributed process environment.

We have proposed a semantic end-to-end process constraint framework, which is composed of three major components: (i) the process constraint language (PCL), (ii) the process constraint ontology (ProContO), and (iii) the process constraint framework (PCF). While PCL and the ProContO ontology focus on the representational aspects of process constraint modeling, the PCF framework concentrates on the static verification in the design phase and dynamic verification of the constraints within the runtime environment.
Having the concept of model-driven engineering in mind, the ultimate goal of our proposed framework has been twofold. First, it provided an integrated representation model for both the process and process constraints by integrating the two with the use of an ontological model. Furthermore, the model is independent of any specific process definition language. Second, our framework allows the designers and implementers to maintain a process constraint’s consistency throughout its whole lifecycle, covering the phases of design, development and runtime verification and enforcement. In this way, process constraints evolve naturally alongside the process development and runtime management.

9.2. Original Contributions

The contributions of our research on the semantic end-to-end constraint framework described in this dissertation are as follows:

- *Process constraint ontology (ProContO):* it serves as the knowledge base for both the process and its associated constraints. It has four top-level classes, ProcessElement, ConstraintAttribute, ConstraintOperation and Exception. These four top-level classes complement each other and form a solid foundation that covers the major concepts representing knowledge of both the process and its constraints.

- *Process constraint language (PCL):* PCL is similar in its purpose to OCL. It has been defined by an EBNF grammar. Most importantly, PCL allows constraint designers to express process constraints in a simple, compact and intuitive way. By extracting ontological entities (e.g., process elements, constraint operations, constraint attributes and exceptions) from ProContO directly, PCL remains simple yet expressive with powerful navigational capabilities. All the participating process and constraint elements are incorporated within PCL statements.
• **Process constraint framework (PCF):** based on PCL and ProContO, PCF is a semantic end-to-end process constraint framework. It has three distinguishing features that existing approaches fail to address, or address only partially:

First, PCF is an end-to-end framework targeted for constraints modeling and development. It covers the whole lifecycle of process constraints, from design, development to runtime verification.

Second, the ontology and other semantic techniques (e.g., OWL and SWIRL) are extensively leveraged in PCF, so that not only the process constraint language becomes more expressive, but also constraint operations are capable of performing more subtle reasoning during runtime verification.

Third, PCF is built upon the service component architecture and does not rely on a specific process definition language or process runtime engine. By promoting all the services to SCA components and SCA services, PCF benefits from all the built-in features of SCA, e.g., runtime monitoring and access to the context information. Therefore, it can perform dynamic constraint evaluation by invoking remote constraint operations and react appropriately, based on the result on the service level.

9.3. **Future Work**

The current development of the PCF framework is still in its initial stage and has been realized only as a proof-of-concept implementation. As our proposed framework involves multiple research areas, the future work can be considered in the following directions:

• **ProContO:** Current ProContO is not yet well populated. Possible improvements may include (i) reusing an existing QoS ontology, (ii) providing better workflow modeling
by capturing more workflow’s context information and representing it as subclasses of ProcessElement.

- **Constraint verification**: Current implementation of constraint verification is preliminarily and more types of semantic verification can be added. Furthermore, runtime verification functionality can be improved, as well. Current implementation uses an online algorithm and only maintains a limited context information and the effectiveness has been sacrificed for the sake of efficiency. Future work may include a more robust algorithm to address both efficiency and effectiveness by keeping track of more context information and historical execution history.

- **PCF framework implementation**: First of all, a better web user interface is needed in the design phase in order to provide better integration with the process definition, involved Web services and related constraints. Deeper integration with SwitchYard tooling can be provided and its built-in monitors and auditors can be utilized for a better context-aware monitor.
BIBLIOGRAPHY


APPENDICES

A. PROCESS CONSTRAINT ONTOLOGY (PROCONTO) IN OWL

<rdf:RDF xmlns="http://www.amy.pcf/constraint/proconto.owl#"
    xml:base="http://www.amy.pcf/constraint/proconto.owl"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<owl:Ontology rdf:about="http://www.amy.pcf/constraint/proconto.owl"/>
</rdf:RDF>
  </owl:ObjectProperty>

<!-- http://www.amy.pcf/constraint/proconto.owl#hasHandler -->
<owl:ObjectProperty

rdf:about="http://www.amy.pcf/constraint/proconto.owl#hasHandler">
  <rdfs:domain rdf:resource="http://www.amy.pcf/constraint/proconto.owl#ExceptionHandler"/>

  </owl:ObjectProperty>

<!-- http://www.amy.pcf/constraint/proconto.owl#hasInput -->
<owl:ObjectProperty rdf:about="http://www.amy.pcf/constraint/proconto.owl#hasInput">
  <rdfs:domain rdf:resource="http://www.amy.pcf/constraint/proconto.owl#Activity"/>
  <rdfs:range rdf:resource="http://www.amy.pcf/constraint/proconto.owl#Data"/>

  </owl:ObjectProperty>

<!-- http://www.amy.pcf/constraint/proconto.owl#hasOutput -->
<owl:ObjectProperty

rdf:about="http://www.amy.pcf/constraint/proconto.owl#hasOutput">
  <rdfs:domain rdf:resource="http://www.amy.pcf/constraint/proconto.owl#Activity"/>
  <rdfs:range rdf:resource="http://www.amy.pcf/constraint/proconto.owl#Data"/>

  </owl:ObjectProperty>

<!-- http://www.amy.pcf/constraint/proconto.owl#hasPerformer -->
<owl:ObjectProperty

rdf:about="http://www.amy.pcf/constraint/proconto.owl#hasPerformer">
<rdfs:domain rdf:resource="http://www.amy.pcf/constraint/proconto.owl#Activity"/>

<rdfs:range rdf:resource="http://www.amy.pcf/constraint/proconto.owl#Performer"/>
</owl:ObjectProperty>

<!-- http://www.amy.pcf/constraint/proconto.owl#hasType -->
<owl:ObjectProperty rdf:about="http://www.amy.pcf/constraint/proconto.owl#hasType">
  <rdfs:domain rdf:resource="http://www.amy.pcf/constraint/proconto.owl#Exception"/>
  <rdfs:range rdf:resource="http://www.amy.pcf/constraint/proconto.owl#ExceptionType"/>
</owl:ObjectProperty>

<!-- http://www.amy.pcf/constraint/proconto.owl#occursAt -->
<owl:ObjectProperty rdf:about="http://www.amy.pcf/constraint/proconto.owl#occursAt">
  <rdfs:domain rdf:resource="http://www.amy.pcf/constraint/proconto.owl#Exception"/>
</owl:ObjectProperty>
B. NEPOTISM ONTOLOGY

Prefix(:<http://www.w3.org/2002/07/owl#>)

Prefix(owl:<http://www.w3.org/2002/07/owl#>)

Prefix(rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)


Prefix(xsd:<http://www.w3.org/2001/XMLSchema#>)

Prefix(rdfs:<http://www.w3.org/2000/01/rdf-schema#>)

Ontology(<http://www.amy.pcf/nepotism/family.owl>)

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Aunt>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Aunt>)

ObjectIntersectionOf(ObjectUnionOf(ObjectMinCardinality(1

<http://www.amy.pcf/nepotism/family.owl#hasNiece>

<http://www.amy.pcf/nepotism/family.owl#Niece> ) ObjectMinCardinality(1

<http://www.amy.pcf/nepotism/family.owl#hasNephew>

<http://www.amy.pcf/nepotism/family.owl#Nephew>))

<http://www.amy.pcf/nepotism/family.owl#Woman>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Brother>)))
EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Brother>)
ObjectIntersectionOf(<http://www.amy.pcf/nepotism/family.owl#Sibling>
<http://www.amy.pcf/nepotism/family.owl#Man>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Child>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Child> ObjectMinCardinality(1
<http://www.amy.pcf/nepotism/family.owl#hasParent>
<http://www.amy.pcf/nepotism/family.owl#Parent>))

SubClassOf(<http://www.amy.pcf/nepotism/family.owl#Child>
<http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Daughter>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Daughter>
ObjectIntersectionOf(<http://www.amy.pcf/nepotism/family.owl#Woman>
<http://www.amy.pcf/nepotism/family.owl#Child>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Father>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Father>
ObjectIntersectionOf(<http://www.amy.pcf/nepotism/family.owl#Parent>
<http://www.amy.pcf/nepotism/family.owl#Man>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Man>))

SubClassOf(<http://www.amy.pcf/nepotism/family.owl#Man>
<http://www.amy.pcf/nepotism/family.owl#Person>)
Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Mother>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Mother>
ObjectIntersectionOf(<http://www.amy.pcf/nepotism/family.owl#Woman>
<http://www.amy.pcf/nepotism/family.owl#Parent>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Nephew>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Nephew>
ObjectIntersectionOf(ObjectUnionOf(ObjectMinCardinality(1
<http://www.amy.pcf/nepotism/family.owl#hasUncle>
<http://www.amy.pcf/nepotism/family.owl#Uncle>) ObjectMinCardinality(1
<http://www.amy.pcf/nepotism/family.owl#hasAunt>
<http://www.amy.pcf/nepotism/family.owl#Aunt>))
<http://www.amy.pcf/nepotism/family.owl#Man>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Niece>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Niece>
ObjectIntersectionOf(ObjectUnionOf(ObjectMinCardinality(1
<http://www.amy.pcf/nepotism/family.owl#hasUncle>
<http://www.amy.pcf/nepotism/family.owl#Uncle>) ObjectMinCardinality(1
<http://www.amy.pcf/nepotism/family.owl#hasAunt>
<http://www.amy.pcf/nepotism/family.owl#Aunt>))
<http://www.amy.pcf/nepotism/family.owl#Woman>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Parent>))
EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Parent> ObjectMinCardinality(1
<http://www.amy.pcf/nepotism/family.owl#hasChild>
<http://www.amy.pcf/nepotism/family.owl#Child>))

SubClassOf(<http://www.amy.pcf/nepotism/family.owl#Parent>
<http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Person>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Person>
ObjectUnionOf(<http://www.amy.pcf/nepotism/family.owl#Woman>
<http://www.amy.pcf/nepotism/family.owl#Man>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Sibling>))

SubClassOf(<http://www.amy.pcf/nepotism/family.owl#Sibling>
<http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Sister>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Sister>
ObjectIntersectionOf(<http://www.amy.pcf/nepotism/family.owl#Woman>
<http://www.amy.pcf/nepotism/family.owl#Sibling>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Son>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Son>
ObjectIntersectionOf(<http://www.amy.pcf/nepotism/family.owl#Man>
<http://www.amy.pcf/nepotism/family.owl#Child>))
Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Uncle>))

EquivalentClasses(<http://www.amy.pcf/nepotism/family.owl#Uncle>)

ObjectIntersectionOf(ObjectUnionOf(ObjectMinCardinality(1
  <http://www.amy.pcf/nepotism/family.owl#hasNiece>
  <http://www.amy.pcf/nepotism/family.owl#Niece>) ObjectMinCardinality(1
  <http://www.amy.pcf/nepotism/family.owl#hasNephew>
  <http://www.amy.pcf/nepotism/family.owl#Nephew>))
  <http://www.amy.pcf/nepotism/family.owl#Man>))

Declaration(Class(<http://www.amy.pcf/nepotism/family.owl#Woman>))

SubClassOf(<http://www.amy.pcf/nepotism/family.owl#Woman>
  <http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasAunt>))

ObjectPropertyDomain(<http://www.amy.pcf/nepotism/family.owl#hasAunt>
  <http://www.amy.pcf/nepotism/family.owl#Person>)

ObjectPropertyRange(<http://www.amy.pcf/nepotism/family.owl#hasAunt>
  <http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasBrother>))

SubObjectPropertyOf(<http://www.amy.pcf/nepotism/family.owl#hasBrother>
  <http://www.amy.pcf/nepotism/family.owl#hasSibling>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasChild>))
InverseObjectProperties(<http://www.amy.pcf/nepotism/family.owl#hasParent> <http://www.amy.pcf/nepotism/family.owl#hasChild>)

ObjectPropertyDomain(<http://www.amy.pcf/nepotism/family.owl#hasChild> <http://www.amy.pcf/nepotism/family.owl#Parent>)

ObjectPropertyRange(<http://www.amy.pcf/nepotism/family.owl#hasChild> <http://www.amy.pcf/nepotism/family.owl#Child>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasCousin>))

SymmetricObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasCousin>)

ObjectPropertyDomain(<http://www.amy.pcf/nepotism/family.owl#hasCousin> <http://www.amy.pcf/nepotism/family.owl#Person>)

ObjectPropertyRange(<http://www.amy.pcf/nepotism/family.owl#hasCousin> <http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasDaughter>))

SubObjectPropertyOf(<http://www.amy.pcf/nepotism/family.owl#hasDaughter> <http://www.amy.pcf/nepotism/family.owl#hasChild>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasFather>))

SubObjectPropertyOf(<http://www.amy.pcf/nepotism/family.owl#hasFather> <http://www.amy.pcf/nepotism/family.owl#hasParent>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasMother>))
SubObjectPropertyOf(<http://www.amy.pcf/nepotism/family.owl#hasMother> <http://www.amy.pcf/nepotism/family.owl#hasParent> )

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasNephew>))

ObjectPropertyDomain(<http://www.amy.pcf/nepotism/family.owl#hasNephew> <http://www.amy.pcf/nepotism/family.owl#Person>)

ObjectPropertyRange(<http://www.amy.pcf/nepotism/family.owl#hasNephew> <http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasNiece>))

ObjectPropertyDomain(<http://www.amy.pcf/nepotism/family.owl#hasNiece> <http://www.amy.pcf/nepotism/family.owl#Person>)

ObjectPropertyRange(<http://www.amy.pcf/nepotism/family.owl#hasNiece> <http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasParent>))

InverseObjectProperties(<http://www.amy.pcf/nepotism/family.owl#hasParent> <http://www.amy.pcf/nepotism/family.owl#hasChild>)

InverseFunctionalObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasParent>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasSibling>))

SymmetricObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasSibling>)}
ObjectPropertyDomain(<http://www.amy.pcf/nepotism/family.owl#hasSibling>
<http://www.amy.pcf/nepotism/family.owl#Person>)

ObjectPropertyRange(<http://www.amy.pcf/nepotism/family.owl#hasSibling>
<http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasSister>))

SubObjectPropertyOf(<http://www.amy.pcf/nepotism/family.owl#hasSister>
<http://www.amy.pcf/nepotism/family.owl#hasSibling>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasSon>))

SubObjectPropertyOf(<http://www.amy.pcf/nepotism/family.owl#hasSon>
<http://www.amy.pcf/nepotism/family.owl#hasChild>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasSpouse>))

SymmetricObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasSpouse>)

ObjectPropertyDomain(<http://www.amy.pcf/nepotism/family.owl#hasSpouse>
<http://www.amy.pcf/nepotism/family.owl#Person>)

ObjectPropertyRange(<http://www.amy.pcf/nepotism/family.owl#hasSpouse>
<http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(ObjectProperty(<http://www.amy.pcf/nepotism/family.owl#hasUncle>))

ObjectPropertyDomain(<http://www.amy.pcf/nepotism/family.owl#hasUncle>
<http://www.amy.pcf/nepotism/family.owl#Person>)
ObjectPropertyRange(<http://www.amy.pcf/nepotism/family.owl#hasUncle>
<http://www.amy.pcf/nepotism/family.owl#Person>)

Declaration(NamedIndividual(<http://www.amy.pcf/nepotism/family.owl#Alice>))

ClassAssertion(<http://www.amy.pcf/nepotism/family.owl#Woman>
<http://www.amy.pcf/nepotism/family.owl#Alice>)

ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasChild>
<http://www.amy.pcf/nepotism/family.owl#Alice>
<http://www.amy.pcf/nepotism/family.owl#Diana>)

ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasChild>
<http://www.amy.pcf/nepotism/family.owl#Alice>
<http://www.amy.pcf/nepotism/family.owl#David>)

ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasSibling>
<http://www.amy.pcf/nepotism/family.owl#Alice>
<http://www.amy.pcf/nepotism/family.owl#Alyssa>)

ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasSpouse>
<http://www.amy.pcf/nepotism/family.owl#Alice>
<http://www.amy.pcf/nepotism/family.owl#Bob>)

Declaration(NamedIndividual(<http://www.amy.pcf/nepotism/family.owl#Alyssa>))

ClassAssertion(<http://www.amy.pcf/nepotism/family.owl#Woman>
<http://www.amy.pcf/nepotism/family.owl#Alyssa>)
ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasChild>
<http://www.amy.pcf/nepotism/family.owl#Alyssa>
<http://www.amy.pcf/nepotism/family.owl#Cathy>)

Declaration(NamedIndividual(<http://www.amy.pcf/nepotism/family.owl#Ben>))

ClassAssertion(<http://www.amy.pcf/nepotism/family.owl#Man>
<http://www.amy.pcf/nepotism/family.owl#Ben>)

ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasChild>
<http://www.amy.pcf/nepotism/family.owl#Ben>
<http://www.amy.pcf/nepotism/family.owl#Emma>)

ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasChild>
<http://www.amy.pcf/nepotism/family.owl#Ben>
<http://www.amy.pcf/nepotism/family.owl#Eric>)

Declaration(NamedIndividual(<http://www.amy.pcf/nepotism/family.owl#Bob>))

ClassAssertion(<http://www.amy.pcf/nepotism/family.owl#Man>
<http://www.amy.pcf/nepotism/family.owl#Bob>)

ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasChild>
<http://www.amy.pcf/nepotism/family.owl#Bob>
<http://www.amy.pcf/nepotism/family.owl#Diana>)

ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasChild>
<http://www.amy.pcf/nepotism/family.owl#Bob>
<http://www.amy.pcf/nepotism/family.owl#David>)
ObjectPropertyAssertion(⟨http://www.amy.pcf/nepotism/family.owl#hasSibling⟩
⟨http://www.amy.pcf/nepotism/family.owl#Bob⟩
⟨http://www.amy.pcf/nepotism/family.owl#Ben⟩)

Declaration(NamedIndividual(⟨http://www.amy.pcf/nepotism/family.owl#Cathy⟩))

ClassAssertion(⟨http://www.amy.pcf/nepotism/family.owl#Woman⟩
⟨http://www.amy.pcf/nepotism/family.owl#Cathy⟩)

Declaration(NamedIndividual(⟨http://www.amy.pcf/nepotism/family.owl#David⟩))

ClassAssertion(⟨http://www.amy.pcf/nepotism/family.owl#Man⟩
⟨http://www.amy.pcf/nepotism/family.owl#David⟩)

ObjectPropertyAssertion(⟨http://www.amy.pcf/nepotism/family.owl#hasSibling⟩
⟨http://www.amy.pcf/nepotism/family.owl#David⟩
⟨http://www.amy.pcf/nepotism/family.owl#Diana⟩)

Declaration(NamedIndividual(⟨http://www.amy.pcf/nepotism/family.owl#Diana⟩))

ClassAssertion(⟨http://www.amy.pcf/nepotism/family.owl#Woman⟩
⟨http://www.amy.pcf/nepotism/family.owl#Diana⟩)

Declaration(NamedIndividual(⟨http://www.amy.pcf/nepotism/family.owl#Emma⟩))

ClassAssertion(⟨http://www.amy.pcf/nepotism/family.owl#Woman⟩
⟨http://www.amy.pcf/nepotism/family.owl#Emma⟩)
ObjectPropertyAssertion(<http://www.amy.pcf/nepotism/family.owl#hasSibling>
<http://www.amy.pcf/nepotism/family.owl#Emma>
<http://www.amy.pcf/nepotism/family.owl#Eric>)

Declaration(NamedIndividual(<http://www.amy.pcf/nepotism/family.owl#Eric>))

ClassAssertion(<http://www.amy.pcf/nepotism/family.owl#Man>
<http://www.amy.pcf/nepotism/family.owl#Eric>)

DLSafeRule(Body(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasParent>
Variable(<urn:swrl#x>) Variable(<urn:swrl#y>)))
ClassAtom(<http://www.amy.pcf/nepotism/family.owl#Man>
Variable(<urn:swrl#y>)))Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasFather> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>))))

DLSafeRule(Body(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasSibling>
> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>)))
ClassAtom(<http://www.amy.pcf/nepotism/family.owl#Man>
Variable(<urn:swrl#y>)))Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasBrother> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>))))

DLSafeRule(Body(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasParent>
Variable(<urn:swrl#x>) Variable(<urn:swrl#y>)))
ClassAtom(<http://www.amy.pcf/nepotism/family.owl#Woman>
Variable(<urn:swrl#y>)))Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasMother> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>))))
DLSafeRule(Body(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasChild> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>)))
ClassAtom(<http://www.amy.pcf/nepotism/family.owl#Man>)
Variable(<urn:swrl#y>))Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasSon> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>))))

DLSafeRule(Body(ClassAtom(<http://www.amy.pcf/nepotism/family.owl#Man> Variable(<urn:swrl#z>))
ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasChild>
Variable(<urn:swrl#y>) Variable(<urn:swrl#z>))
ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasSibling>
Variable(<urn:swrl#x>))
Variable(<urn:swrl#y>))Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasNephew> Variable(<urn:swrl#x>) Variable(<urn:swrl#z>))))

DLSafeRule(Body(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasChild>
Variable(<urn:swrl#x>) Variable(<urn:swrl#y>)))
ClassAtom(<http://www.amy.pcf/nepotism/family.owl#Woman>)
Variable(<urn:swrl#y>))Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasDaughter> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>))))

DLSafeRule(Body(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasSibling>
> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>)))
ClassAtom(<http://www.amy.pcf/nepotism/family.owl#Woman>
Variable(<urn:swrl#y>)) Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl #hasSister> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>)))))

DLSafeRule(Body(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasParent> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>)))
ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasBrother> Variable(<urn:swrl#y>))
Variable(<urn:swrl#y>)
Variable(<urn:swrl#z>)) Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl #hasUncle> Variable(<urn:swrl#x>) Variable(<urn:swrl#y>) Variable(<urn:swrl#z>)))))

DLSafeRule(Body(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasParent> Variable(<urn:swrl#x1>) Variable(<urn:swrl#y1>)))
ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasParent> Variable(<urn:swrl#x2>) Variable(<urn:swrl#y2>)))
ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasSibling> Variable(<urn:swrl#y1>)
Variable(<urn:swrl#y2>)) Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl #hasCousin> Variable(<urn:swrl#x1>) Variable(<urn:swrl#x2>))))

DLSafeRule(Body(ClassAtom(<http://www.amy.pcf/nepotism/family.owl#Woman> Variable(<urn:swrl#z>)))
ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasChild> Variable(<urn:swrl#y>) Variable(<urn:swrl#z>))
ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasSibling> Variable(<urn:swrl#x>))
Variable(<urn:swrl#y>))Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasNiece> Variable(<urn:swrl#x>) Variable(<urn:swrl#z>)))))

DLSafeRule(Body(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasSister> Variable(<urn:swrl#y>) Variable(<urn:swrl#z>)))

ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasParent> Variable(<urn:swrl#x>))

Variable(<urn:swrl#y>))Head(ObjectPropertyAtom(<http://www.amy.pcf/nepotism/family.owl#hasAunt> Variable(<urn:swrl#x>) Variable(<urn:swrl#z>)))))

)
C. PROCESS CONSTRAINT LANGUAGE (PCL) PROCESSER

```scala
package pcf.pcl
import java.util.StringTokenizer
import scala.util.parsing.combinator.JavaTokenParsers
import models.pcf.pcl._
import scala.collection.mutable.ListBuffer
import org.slf4j.LoggerFactory

class PCLParser extends JavaTokenParsers {
  val logger = LoggerFactory.getLogger(getClass());

  def pclParse(input: String): PCLParseResult = {
    val parseResult = parseAll(pcl, input)
    parseResult match {
      case Success(x, _) => new PCLParseResult(Some(parseResult.get), None)
      case NoSuccess(err, next) => {
        val error = 
          "failed to parse PCL input " + 
          "(line " + next.pos.line + ", column " + 
          next.pos.column + "):
" + 
          err + 
          "n" + 
          next.pos.longString + "n"
        logger.debug(error)
        new PCLParseResult(None, Some(error))
      }
    }
  }

    conditions ~ opt(exception_handler_clause) ^^ {
    case constraintName ~ context ~ conditions ~ exceptionHandler => {
      val result = new PCLConstraint(constraintName, context, conditions, exceptionHandler)
      result
    }
  }

  def exception_handler_clause: Parser[PCLExceptionHandler] = 
    PCLConstraint.Keyword_exception ~ PCLConstraint.Keyword_when ~ ident ~ PCLConstraint.Keyword_then ~ ident ^^ {
    case keywordException ~ keywordWhen ~ exception ~ keywordThen ~ handler => {
      val pclException = new PCE_exception(exception)
      val result = new PCLExceptionHandler(pclException, handler)
      result
    }
  }
}
```
def context: Parser[PCLContext] = PCLConstraint.Keyword_context ~> repsep(process_element, ",") ^^ {
  case list => {
    var buffer = new ListBuffer[PCLProcessElement]()
    list.foreach(e => buffer += e)
    new PCLContext(buffer.toList)
  }
}

// Assume all processElements defined in context are activity elements
def process_element: Parser[PCLProcessElement] = opt(ident <~ ":") ~ ident ^^ {
  case alias ~ name => alias match {
    case Some(alias) => new PCLProcessElement(alias, name)
    case None => new PCLProcessElement(name)
  }
}

def conditionType: Parser[String] = PCLConstraint.Keyword_condition_Pre | PCLConstraint.Keyword_condition_Post | PCLConstraint.Keyword_condition_Inv | failure("unsupported condition type")

def conditions: Parser[List[PCLCondition]] = rep(condition)^

def condition: Parser[PCLCondition] = conditionType ~ (expression | if_expression) ~ opt(exception_clause) ^^ {
  case conditionType ~ expr ~ exceptions => {
    logger.debug("-----> condition:" + conditionType + ", exceptions" + exceptions)
    val result = PCLCondition(conditionType, expr, exceptions)
    result
  }
}

def exception_clause: Parser[List[PCLException]] = PCLConstraint.Keyword_raise ~> exceptions ^^ {
  case list => {
    val result = new ListBuffer[PCLException]()
    list.foreach(exceptionType => {
      result += new PCLException(exceptionType)
    })
    result.toList
  }
}

/**
 * rep1sep: at least match one exception type
 */
def exceptions: Parser[List[String]] = rep1sep(ident, ",") ^^ {
  case list => {
    list
  }
}

def expression: Parser[PCLEExpression] = logical_expression

def logical_operator: Parser[String] = "and" | "or" | "xor" | failure("unsupported logical operator")

def logical_expression: Parser[PCLEExpression] = relational_expression ~
rep(logical_operator ~ relational_expression) ^^ {
  case left ~ list => {
    if (list.size == 0) {
      // Only one relational_expression
      left
    } else {
      var exp: BinaryExpression = null
      list.foreach(opExpr => {
        val op = opExpr._1
        val right = opExpr._2
        if (exp == null) {
          exp = new BinaryExpression(op, left, right)
        } else {
          val nextExp = new BinaryExpression(op, exp, right)
          exp = nextExp
        }
      })
      exp
    }
  }
}

def relational_expression: Parser[PCLEExpression] = additive_expression ~
rep(relational_operator ~ additive_expression) ^^ {
  case left ~ list => {
    if (list.size == 0) {
      left
    } else {
      var exp: BinaryExpression = null
      list.foreach(opExp => {
        val op = opExp._1
        logger.debug("relational_operator = "+ op)
        val right = opExp._2
        logger.debug("right = "+ right)
      })
      exp
    }
  }
}
if (exp == null) {
    exp = new BinaryExpression(op, left, right)
} else {
    val nextExp = new BinaryExpression(op, exp, right)
    exp = nextExp
}
}

def additive_expression: Parser[PCLEExpression] = multiplicative_expression ~
rep(add_operator ~ multiplicative_expression) ^^ {
  case left ~ list => {
    if (list.size == 0) {
      left
    } else {
      var exp: BinaryExpression = null
      list.foreach(opExpr => {
        val op = opExpr._1
        logger.debug("add_operator = " + op)
        val right = opExpr._2
        logger.debug("right = " + right)
        if (exp == null) {
          exp = new BinaryExpression(op, left, right)
        } else {
          val nextExp = new BinaryExpression(op, exp, right)
          exp = nextExp
        }
      })
      exp
    }
  }
}

def multiplicative_expression: Parser[PCLEExpression] = unary_expression ~
rep(multiply_operator ~ unary_expression) ^^ {
  case unary ~ list => {
    if (list.size == 0) {
      unary
    } else {
      var exp: BinaryExpression = null
      list.foreach(opExpr => {
        val op = opExpr._1
        logger.debug("multiply_operator = " + op)
        val right = opExpr._2
        logger.debug("right = " + right)
        if (exp == null) {
          exp = new BinaryExpression(op, left, right)
        } else {
          val nextExp = new BinaryExpression(op, exp, right)
          exp = nextExp
        }
      })
      exp
    }
  }
}
val right = opExpr._2
logger.debug("right = " + right)
if (exp == null) {
    exp = new BinaryExpression(op, unary, right)
} else {
    val nextExp = new BinaryExpression(op, exp, right)
    exp = nextExp
}
}

exp
}
}

def unary_operator: Parser[String] = "not"
    failure("unsupported unary operator")

def multiply_operator: Parser[String] = "*"|"/"
    failure("unsupported multiply operator")

def add_operator: Parser[String] = "+"|"-"
    failure("unsupported add operator")

def relational_operator: Parser[String] = "<>"|"="|"<="|">="|">"|"<"|
    failure("unsupported relational operator")

def unary_expression: Parser[PCLExpression] = opt(unary_operator) ~ primary_expression
  ^ { case op ~ primary_expression => { op match { case Some(unaryOp) => { logger.debug("unary op = " + unaryOp) val result = new UnaryExpression(unaryOp, primary_expression) result } case None => { primary_expression } } } } }

def if_expression: Parser[PCLExpression] = ("if" ~> expression) ~ ("then" ~> expression) ~
    opt("else" ~> expression) <~ "endif" ^ { case cond ~ default ~ alt => new IfExpression(cond, default, alt) }
def primary_expression: Parser[PCLExpression] = if_expression | floatingPointNumberParser | attributeParser | operationParser |
literalParser | "(" ~> expression <~ ")" | failure("unsupported primary expression")

def operationParser: Parser[PCLExpression] = ident ~ "(" ~> repsep(primary_expression, ",", ) <~ ")") ^^ {
  case operation ~ list => {
    logger.debug("operation = " + operation)
    val constraintOperation = new PCLConstraintOperation(operation, list)
    new Func(constraintOperation)
  }
}

/**
 * constraint attribute is in the form of "a->b.c->d.location" with navigation operators (".", ",->")
 * Refer match multiple occurrence of dot
 * http://stackoverflow.com/questions/4739759/how-to-match-repeated-patterns
 */
def attributeParser: Parser[PCLExpression] = "w+\((\.|->)w+)\".r ^^ {
  case literal => {
    literal match {
      case s: String => {
        logger.debug("attribute literal = " + s)
        /*
         * Refer split a string, but also keep all the delimiters
         * a->b.c->d.location produces ("a" "->" "b" "," "c" "->" "d" "." "." "location")
         */
        val WITH_DELIMITER = "((?<=\%1$s)|(?=%1$s))"
        val words = s.split(String.format(WITH_DELIMITER, ".\.|->")
        val path = words.toList
        // path.foreach(println)
        new Var(new PCLConstraintComplexAttribute(path))
      }
    }
  }
}

def literalParser: Parser[PCLExpression] = (ident | stringLiteral) ^^ {
  case literal => {
    literal.toLowerCase() match {
      case "true" => new Constant(PCLConstraintConstantAttribute[Boolean](true))
    }
  }
}
case "false" => new Constant(PCLConstraintConstantAttribute[Boolean](false))
  case _ => new Constant(PCLConstraintConstantAttribute[String](literal))
}
}
}

def floatingPointNumberParser: Parser[PCLExpression] = floatingPointNumber ^^ {
  case number => {
    new Constant(PCLConstraintConstantAttribute[Float](number))
  }
}
}
D. PCF FRAMEWORK PROJECT

Download the whole pcf project from source control repository and the project includes the following sub-projects, as listed in Table D.1.

Table D.1. Project list for PCF framework

<table>
<thead>
<tr>
<th>Project name</th>
<th>Packaging</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>pcf</td>
<td>war</td>
<td>include pcl parser and web ui deployed in JBoss EAP</td>
</tr>
<tr>
<td>pcf-operations</td>
<td>jar</td>
<td>include constraint operations and exception handling operations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>deployed in SwitchYard runtime</td>
</tr>
<tr>
<td>pcf-processes</td>
<td>jar</td>
<td>include all the business processes and scientific workflows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>deployed in SwitchYard runtime</td>
</tr>
<tr>
<td>pcf-services</td>
<td>jar</td>
<td>include all the services that are used to compose the processes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>deployed in SwitchYard runtime</td>
</tr>
<tr>
<td>Switchyard-helloworld</td>
<td>jar</td>
<td>include a simple switchyard project</td>
</tr>
<tr>
<td></td>
<td></td>
<td>deployed in SwitchYard runtime</td>
</tr>
</tbody>
</table>
E. PCF FRAMEWORK DEVELOPMENT ENVIRONMENT SETUP

The steps to set up the development environment are listed as follows:

1. Install Java, Ant and Maven and configure them properly.
   a. JDK version 7
   b. Ant version 1.9.3
   c. Maven version 3.2.1

2. Install SopaUI to test Web services.

3. Eclipse IDE
   a. Eclipse IDE Kepler v4.3

4. Install SwitchYard tooling in JBoss tools plugins
   a. Install from Switchyard repository
   b. Refers to the switchyard tooling installation guide
      (https://docs.jboss.org/author/display/SWITCHYARD/Installing+Eclipse+Tooling)
   c. Add an update site
      (http://download.jboss.org/jbosstools/updates/stable/kepler/integration-stack/aggregate/4.1.4.Final/) to eclipse repository
   d. Install all the features and plugins related to BPEL, BPMN and SwitchYard from this update site:
      i. JBoss Business Process and Rules Development
      ii. JBoss Integration and SOA Development
      iii. Jboss BPEL Editor
      iv. Switchyard project support
   e. Optional approach is to install jbosstools from Eclipse Marketplace
5. Install Jboss EAP 6.1
   a. Download jboss-eap-6.1 (NOT jboss-as-6.1).
   b. Note: JBoss EAP is Enterprise Application Platform, while JBoss AS is community application server. Jboss EAP 6.1 uses Jboss AS 7.2 internally.
   c. Extract to a directory
   d. (optional) create a admin user and password with add_user.sh

6. Install Switchyard 1.1 runtime to Jboss EAP
   a. download switchyard-installer-1.1.0.Final.zip
   b. install switchyard 1.1 runtime following the guide
      (https://docs.jboss.org/author/display/SWITCHYARD/Installation+Guide)
   c. (optional) install bpel console

7. Set up SwitchYard runtime in Eclipse
   a. Open Eclipse jbosstools
   b. Add a new server runtime (Windows -> Preference --> Server).
   c. SSclects JBoss EAP server runtime 6.x and points the folder to the existing EAP installation directory where SwitchYard is just installed.
F. SIMPLE SCA TUTORIAL

1. BPEL process as SCA components

   SwitchYard supports BPEL development and deployment by utilizing Jboss's Riftsaw project (http://riftsaw.jboss.org/). Riftsaw is based on Apache ODE project (http://ode.apache.org/). Switchyard tooling supports the following features:

   a. Graphical BPEL 2.0 editor and ODE deployment configuration editor.
   b. WS-BPEL 2.0 OASIS standard (http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf)
   c. Native JBossWS stack and CXF Web Service stack.
   d. UDDI registration of BPEL endpoints, and Runtime UDDI Endpoint lookup as preview feature.

2. BPMN process as SCA components

   SwitchYard supports BPMN2 via the integration of JBoss BPM Suite (http://www.jboss.org/products/bpmsuite/overview/). The BPM component is a pluggable container and has integrated jBPM 5. SwitchYard supports the following features:

   a. Graphical BPMN2 editor
   b. SwitchYard service task as a BPMN task extension

3. Camel routing through Enterprise Integration Pattern

   Camel is a library to define routing and mediation rules and works with a variety of binding protocols. Another powerful feature for Camel library is its native support and implementation of the EIPs. Enterprise Integration Patterns (EIPs) are a set of patterns and best practices in the area of enterprise integration.
Camel is not a process runtime in the real sense. However, due to its support of some common process runtime services (e.g., data transformation, data mediation, service monitoring, and orchestration, etc.), it can be used to build simple workflow manager and utilizes the EIPs to compose the services into a complete workflow.

4. Constraint operations as SCA components

All the constraint operations (and exception handler operations) are first developed as a plain Web service and then promoted to an SCA component. In this way, constraint operations and exception handlers can be referenced directly by other SCA services.

A simple implementation of the constraint operation service invoker is listed in Table F.1. This Java bean is a remote invoker to an SCA remote service. Because the current version of SwitchYard cannot send HTTP or SOAP request in Camel routing directly, we cannot route the constraint operation to an HTTP endpoint directly and have to use a remote invoker mechanism provided by SwitchYard (https://docs.jboss.org/author/display/SWITCHYARD/Remote+Invoker). Basically, any SCA services can be invoked by name via sending request to the remote URL (e.g., http://localhost:8080/switchyard-remote). Therefore, the constraint operations are integrated successfully in PCF framework without any change in their original implementation.

Table F.1. A simple implementation for a constraint operation

```java
@Service(FileSizeCheckerInvoker.class)
public class FileSizeCheckerInvokerBean implements FileSizeCheckerInvoker {

    private static final Logger logger = Logger.getLogger(FileSizeCheckerInvokerBean.class);
    private static final String SWITCHYARD_REMOTE_URL = "http://localhost:8080/switchyard-remote";
    private static final String REMOTE_SERVICE_NAME = "FileSizeChecker";
    private static final String TARGET_NAMESPACE = "urn:edu.uga.cs:pcf-operations:1.0";

    @Override
    public String check() {
        String result = "";
        try {
```
SwitchyardRemote remote = new SwitchyardRemote(SWITCHYARD_REMOTE_URL);
result = remote.invoke(TARGET_NAMESPACE, REMOTE_SERVICE_NAME, content);
logger.info("filesize result = " + result);
} catch (IOException e) {
    logger.error("failed to save to file size checker", e);
}

return result;