Connecting Process Models, Object Life Cycles and Context-dependent Conditions Through Semantic Specifications

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Ralph Hoch, BSc
Registration Number 0405156

to the Faculty of Electrical Engineering and Information Technology
at the TU Wien

Advisor: Univ.Prof. Dr.techn. Hermann Kaindl
Second advisor: Dr.techn. Roman Popp

The dissertation has been reviewed by:
Univ.Prof. Dr. Óscar Pastor
Centro de Investigación en Métodos de Producción de Software (PROS)
Universidad Politècnica de Valencia, Spain.

Univ.Prof. Dr. Xavier Franch
GESSI Research Group
Universitat Politècnica de Catalunya (UPC), Spain.

Declaration of Authorship

Dipl.-Ing. Ralph Hoch, BSc

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 28th May, 2019

Ralph Hoch

Acknowledgments

First, I would like to thank my parents, who always supported me, morally and financially, and always encouraged me, to use my talents in every way possible. Also, moving to Vienna without their help, and the use of their car, would have been so much harder. Many trips with them to IKEA provided my first accommodation. Although I am stubborn and had countless arguments with them, they never lost their patience with me.

I also would like to thank my girlfriend, who helps me with words and deeds. I am pretty sure my parents are glad that somebody took pity and is willing to deal with my stubbornness on a daily basis. As she is also a fellow engineer I had many interesting discussions with her, which helped me during my studies. I am looking forward to many more great years together.

Last but not least, I would like to thank my advisor Prof. Hermann Kaindl, assistant advisor Dr. Roman Popp, and co-researcher Christoph Luckeneder. During our conversations, and sometimes disputes about details, I always found new perspectives to look at topics. They always were open minded for new ideas and a source of inspiration as I worked on my thesis.

Part of this research has been carried out in the ProREUSE project (No. 834167), funded by the Austrian FFG, and FeatureOpt project (No. 849928), funded by the Austrian BMVIT (represented by the Austrian FFG).

Abstract

Conceptual models of process-centric software can be task- or artefact-centric. These approaches are mostly used in isolation and no formal connection exists. Tasks in (business) processes may be executed by services and operate on business objects (artefacts). In general, there is no formal definition of the actions a task performs and how this has effects on business objects. In essence, there is no connection between a task, its executing service(s) and business object(s). Moreover, business objects may not change their status arbitrarily but only in a well-defined order according to their object life cycle. Since there exists no defined connection between the models, they are often not consistent with each other, e.g., a process may perform an action that does not fit the object life cycle. Thus, *formally verifying consistency* of process models and object life cycles is desirable.

This thesis presents an approach to formally connect tasks in process models and object life cycles through *semantic specifications.* We declaratively specify the actions that a task performs, via pre- and post-conditions, and relate them to attributes of object life cycles, i.e., we *ground them in the object life cycle.* This establishes a well-defined connection between the conceptual models of processes and business objects and enables checking their consistency.

An additional complication is that in different contexts, defined by *business rules*, a task may have different conditions that have to be fulfilled for its execution, i.e., *tasks are context-dependent.* Thus, we propose semantically specified context-dependent conditions. The semantic specifications of tasks in processes are enriched with context information and their pre- and post-conditions are adapted accordingly. This allows checking the consistency of process models against object life cycles in a specific context.

In order to guarantee that the same (software) services can be reused for implementing tasks in different contexts, the specifications of the former must be in *subtyping* relationships with the specifications of the latter. Even a recursive application of this approach on different levels of abstractions is possible. On a higher abstraction level, a composition of tasks may be viewed as a single step in the process as long as the subtyping relationship is enforced.

In conclusion, the proposed approach enables to formally connect process models, object life cycles and context-dependent conditions and to verify if they are procedurally and logically consistent.

Contents

CHAPTER 1

Introduction

Modern (business) software applications are commonly developed with conceptual models as their foundation. Often the software relies on well-structured process specifications, which define the order in which functions, or more generally speaking tasks, may be executed. In the simplest case, tasks are only executed in the same sequential order each time. For most applications this approach is not sufficient and a more flexible one is desirable. This is especially true for business software applications in domains with much variability. In this case, the software may execute tasks in different order each time depending on some internal events. Such software applications should be flexible enough to support this kind of behavior and also enable the possibility to easily adapt the possible processes of tasks, i.e., they should utilize conceptual models of (business) processes. This approach is called *process-centric* software development.

In addition, (business) software typically operates on domain entities, e.g., an invoice, and, during execution, alter the states and values of these entities. In more general terms, these entities are objects that a software manipulates. As one can possibly imagine, not all manipulations are permitted at any given time. Typically, an object may only be altered in a well-defined manner and often changes to their attributes or states are not retractable, e.g., as soon as an invoice has been paid it cannot be "unpaid", it can only be refunded. The possible orders in which an object may be manipulated have to be defined in a conceptual model as well, i.e., a formally specified object life cycle has to exist.

For process-centric software and their business processes there exist two major modeling paradigms: *task-centric* and *artefact-centric*. In a task-centric approach the business process is comprised of tasks which perform actions. The actions of the tasks drive the process and do not have to be related to an artefact of the domain, e.g., a typical example could be a timer task. These tasks are often, but not exclusively, performed by (software) services, e.g., Web services. A current industry standard for modeling task-centric (business) proccesses is the Business Process Model Notation (BPMN). In contrast, artefact-centric models regard objects as "first class citizens", i.e., the data objects drive the process. In this case, all actions are performed on objects and change the state of one or more objects synchronously. Here the object life cycles are the main models the software operates on.

Both modeling approaches have their advantages and their use cases, but typically they are not used in combination.

1.1 Problem Statement

As mentioned above, two modeling approaches for (business) processes exist. Software applications that combine both approaches could use the best of both worlds. However, this not only raises questions on how to connect them, but also on how this connection may be verified for consistency.

One problem is, that both models do not have to be created by the same stakeholder. In general a process-designer is responsible for creating (business) process models, but one designer might not be familiar with both modeling approaches. This means, that a task-centric (business) process model might be created by a different person than the artefact-centric object life cycle model. In addition, a software developer may implement a service that executes a task in the (business) process and relies on elements of the object, e.g., specifies the pre- and postconditions for the service to be executed. All these involved stakeholders work with these models and thsu all these models have to be consistent with each other. Figure 1.1 illustrates this problem.

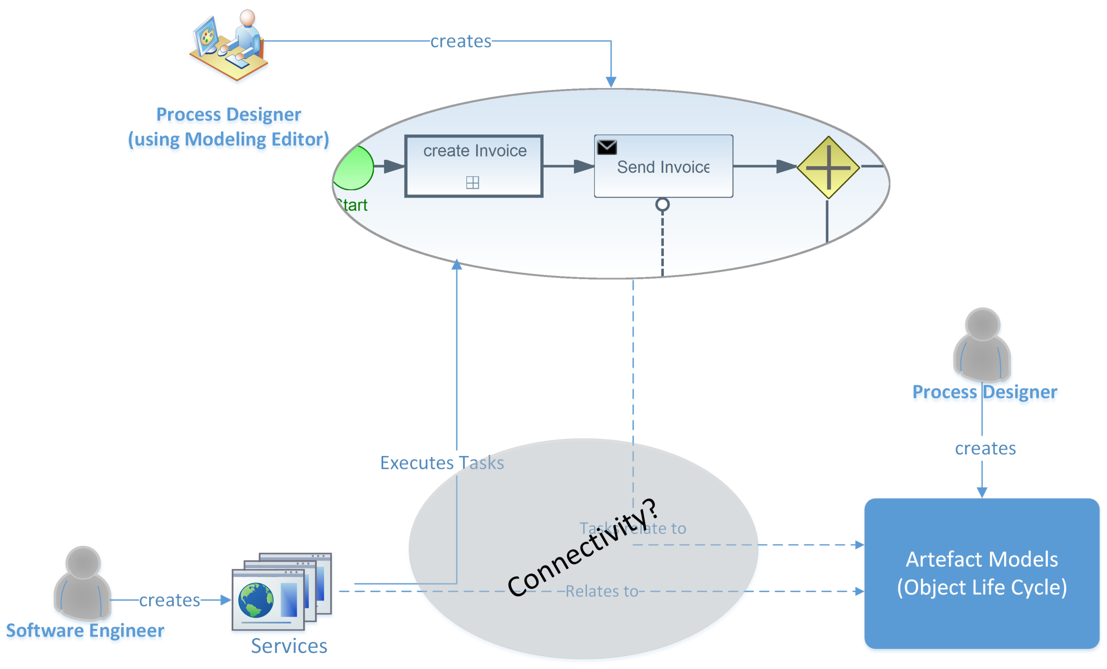


Figure 1.1: Connectivity and Semantic Specification Issues between different Conceptual Models

Another issue that arises is, that implementations of services may be used in different (business) processes. Even more so, the services may be employed in different contextual situations and the executing tasks may have different pre- and postconditions. The *reuse* of service implementations is crucial and should be facilitated. That is, each task should have its own *context-dependent semantic specification.*

The main research questions that arise are:

1. How can task-centric and artefact-centric models be connected?
2. Are semantic service specifications sufficient for task-centric process models?
3. How can a software service be reused in a different context without changing its implementation/specification?
4. Is it possible to decouple semantic specifications from process knowledge?
5. Is formal verification of process models against object life cycles possible?

1.2 Motivation

There exists a variety of publications in the process modeling domain, but only a fraction considers both major modeling approaches. Often transferring a model from one paradigm to the other is discussed, e.g., by Meyer and Weske [68], or their advantages are compared, but combining them both is almost never addressed. This thesis tries to bridge the gap between different conceptual models and provide the means to formally verify their *consistency.* By utilizing semantic specifications we provide the means for a formal specification of the tasks and their behavior in relation to objects in the domain. More specifically, we use formal defined object life cycles to relate tasks of (business) processes as well as their implementing services to artefacts.

Although task specifications using pre- and post-conditions are already well understood, there is no indication for relating them specifically to formal models of object life cycles. In addition, there is no indication that they can be useful for providing the means to connect tasks to objects and verify their consistency. The same applies for their implementing services. (Web) services can already be semantically specified, but their relation to other conceptual models is rarely addressed. In particular their relation to tasks of (business) processes is typically only considered as an implementation. The implemented tasks are not seen as separate entities with their own formal specification, apart from mappings of local process variables to parameters of the services, e.g., input and output mapping in BPMN. Figure 1.2 illustrates this issue.

In this thesis, we address these issues and provide meaningful insight on how different conceptual models may be connected and their consistency formally verified by semantic specifications.

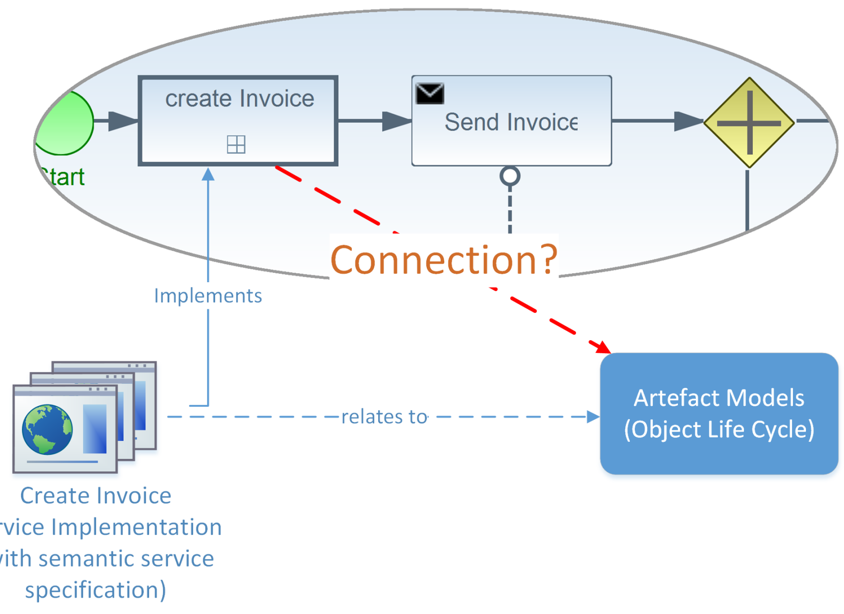

create Invoice
Send Invoice
Start
Connection?
Implements
relates to
Artefact Models (Object Life Cycle)
Create Invoice Service Implementation (with semantic service specification)

Figure 1.2: Lack of Semantic Specifications of Tasks in Process Models

1.3 Running Example

Throughout this thesis, we use a running example to illustrate problems with existing approaches and our solutions to them. We decided to base our running example on an existing reference process from the literature. The running example, shown in Figure 1.3, is based on a simplified version of the "payment handling" process of [98, p. 108], which we use here as a reference process.

This payment handling process involves two participants, the *Delivery Company* and the *Customer Company*. Each participant has its own process that is executed in its own content. The processes are synchronized at specific points illustrated by dotted lines. Each process is started independently and only at the synchronization points information is exchanged. One process may only continue if it receives the information necessary to proceed from the other process. For illustration purposes we omit the data flow of objects and messages from the figure.

The process of the delivering company starts with the creation of an *Invoice*. This is symbolized by the "Create Invoice" step in the figure. Afterwards, the created Invoice is passed to the "Transmit Invoice" step where a message containing the Invoice is transmitted to the customer company. The delivering company process continues immediately and proceeds to the "Receive Payment From Customer" step. There it awaits the payment of the Invoice, i.e., the process does not continue until the payment is received.

The process of the customer company starts and proceeds to the first step "Receive Invoice". This step blocks the process until an invoice is transmitted by the delivering

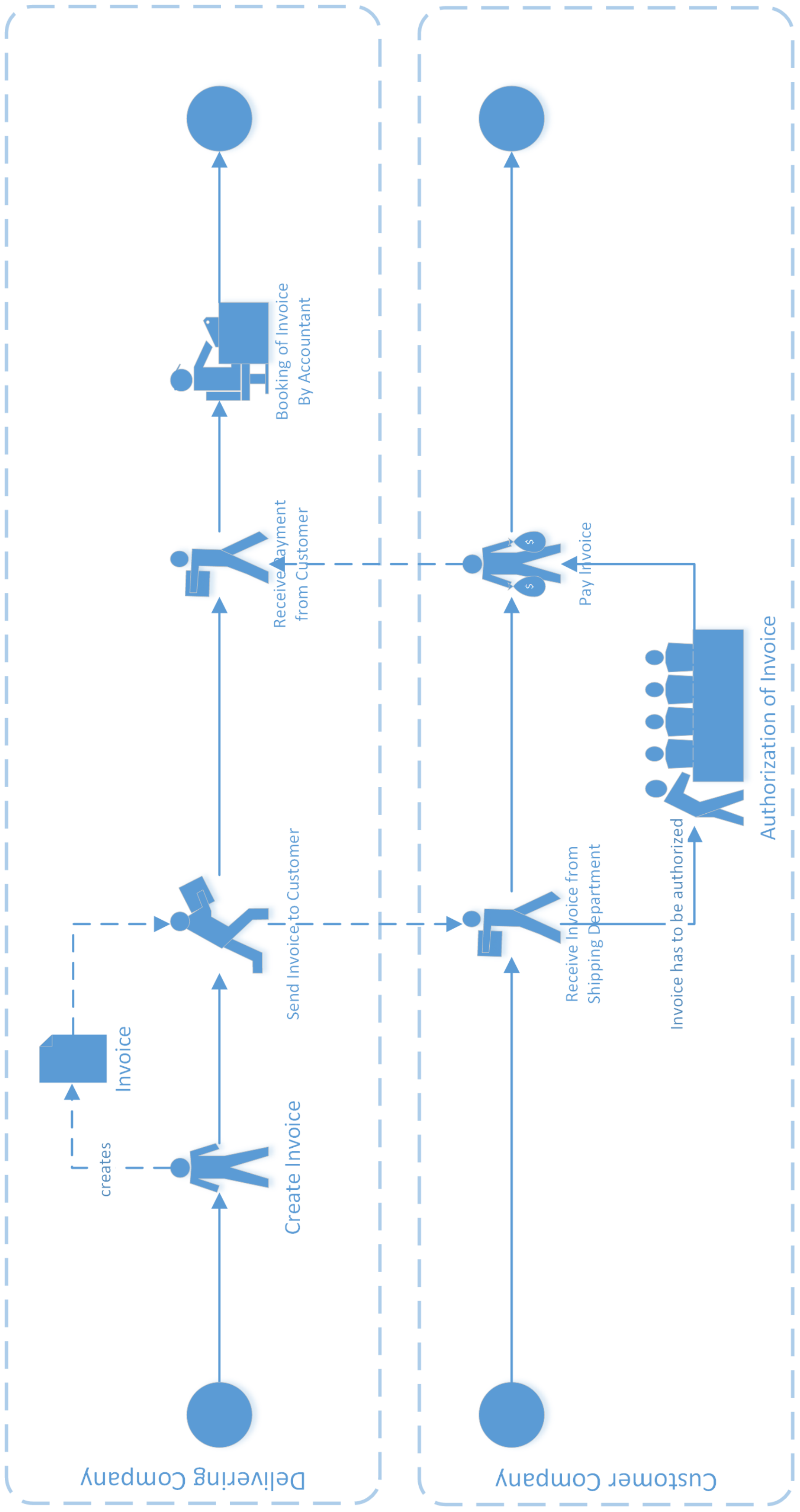

Delivering Company
Customer Company
Create Invoice
creates
Invoice
Send Invoice to Customer
Receive Payment from Customer
Booking of Invoice By Accountant
Receive Invoice from Shipping Department
Invoice has to be authorized
Authorization of Invoice
Pay Invoice

Figure 1.3: Payment Handling Process, Including Authorization

company, i.e., until it receives an invoice. As soon as the delivering company, via "Transmit Invoice", transmits a invoice, the customer company process continues. The next step involves making a payment for the invoice. While the payment is unconditional in the reference process of [98, p. 108], ours actually includes a conditional authorization according to an informally given (business) rule. This rule is given in (Business) Rule 1.1 and states, that all invoices that have an amount greater than a threshold have to be authorized before a payment can be made.

(Business) Rule 1.1. *Authorization Required For Payment If Amount Greater Than Threshold*
"If the amount of an invoice is greater or equal than a threshold level, its payment has to be authorized."

This authorization replaces the simple payment step of the reference process with a whole process part. Depending on the amount, the step "Authorize Invoice" is executed or not. Subsequently the "Pay Invoice" step is reached and the payment of the invoice is made. This payment is also sent to the delivering company. After making the payment the customer company process finishes.

At this point the delivering company process receives the information of the payment and continues. The final step performed is "Book Invoice" where the invoice is booked in the accounting. Finally, the process of the delivering company finishes.

1.4 Structure of the Thesis

Following this introduction, Chapter 2 provides background information for the conceptual approach presented in this thesis. The chapter is split into several sections on various topics. It starts with an overview on (business) process models, including major modeling paradigms, standardization approaches and implementing technologies, and their use for software applications. We present Semantic Service Specifications briefly as (Web) services are typically used to perform actions in (business) process models. In addition, an introduction to (business) rules and business artefacts, including object life cycles, is given. Lastly, we present the two approaches used for formal verification in this thesis, Model Checking and Fluent Calculus. Alongside this background information, we provide an overview on state-of-the-art approaches in this research topic.

Chapter 3 introduces semantic service specifications and shows how they are used for service composition. It describes how service specifications are represented in the *Fluent Calculus* and how its implementation in *FLUX* can be used for formal verification. Finally, it shows a *verification and validation mismatch* when using semantic service specifications for service composition in different business context [47].

In Chapter 4, a solution to the identified verification and validation mismatch is proposed. We present *semantic task specifications* and show how they are utilized to solve the problem of over-specifying services [46, 87]. In addition, we show how they make services and their semantic specifications *reusable*. To this end, we use conditional (business) rules to express context information, i.e., they represent *context-dependent* *conditions*. This enables us to *reuse* services and their specifications in different *contexts*. Finally, we show how our approach can be applied *recursively* [50].

Chapter 5 describes the conceptual models of our object life cycles and their use for formal verification [48]. We introduce the definition of semantic specifications *without knowledge of the process* and how these specifications are *logically grounded* in formal *object life cycles* and their attributes.

In Chapter 6 we combine all parts presented in the previous chapters (semantic task specifications, object life cycles, context-dependent conditions) and show how they can be used for *formal verification*. We illustrate how these models are transformed to Finite State Machines (FSMs) and how a *model checking tool* can be used for *consistency checking*. To this end we also present how (business) rules can be formalized for model checking [90].

Chapter 7 shows the results of our approach and presents its evaluation. We show how adaption of a (business) rule influences the verification of process models without the need of adapting them or referenced parts.

In Chapter 8 we discuss limitations and assumptions we made. In addition, we present ongoing as well as future research topics in this field and relate them to our approach. We lay out a roadmap of our next steps and how our approach will evolve.

Finally, the thesis ends with Chapter 9, where a conclusion is given.

CHAPTER 2

Background and Related Work

This chapter provides background information on all supporting concepts of our conceptual verification approach presented in this thesis. We give an overview, including supporting technologies, on these concepts and particular focus on their integration with our approach. We also present state-of-the-art approaches and related work in the research field addressed in this thesis.

We start with presenting conceptual (business) process models and their use for process-centric software. The two major modeling paradigms are introduced and current standardization approaches are discussed. In particular, we present BPMN, a modern modeling notation for business processes, and how our running example can be expressed with it.

Subsequently semantic service specifications are introduced, since (Web) services are commonly used to implement actions in (business) process models. Their formal specification is one pillar of our verification approach. We relate them to (business) process models and present current specification approaches.

(Business) Rules and Business Artefacts are introduced next. We classify the (business) rules used in this thesis and relate them to our verification approach. For business artefacts we use conceptual models including object life cycles.

Finally, we present the two verification approaches used in this thesis. First, the Fluent Calculus and its implementation FLUX is presented, where we give an overview of their fundamentals. Secondly, we introduce model checking.

2.1 Using Process Models for Process-centric Software

Software, and especially business software, is often used to execute tasks (activities), which create some sort of asset. Typically, these tasks are not executed in isolation, but are rather processed together, where tasks can provide outputs, which are then used as inputs for other tasks. The resulting composition of tasks creates a process that may be executed by software. However, this process is often only available hard-coded in the source code of software. As the requirements for the software change and the software evolves over time, also the source code has to be adapted [56]. Adapting the source code for each new requirement is not only time consuming, but also increases maintenance costs [15]. A flexible approach for executing processes of tasks without the need for adapting the source code would help to support the software evolution and to reduce costs [44, 89]. Hence, a separate specification of processes, that expresses the high-level functionality a software performs, is desirable. Such *process models* define the flow in which tasks are executed by the software.

A *machine-processable* specification of the process models is necessary to automatically process and execute them. The processable models are loaded by a *process-engine* and the embedded tasks are executed according to their specification. This leads to a *process-centric* approach to software applications, where the process models are the driving force of their execution. Figure 2.1 illustrates how a (simplified) *process-centric* approach can be realized.

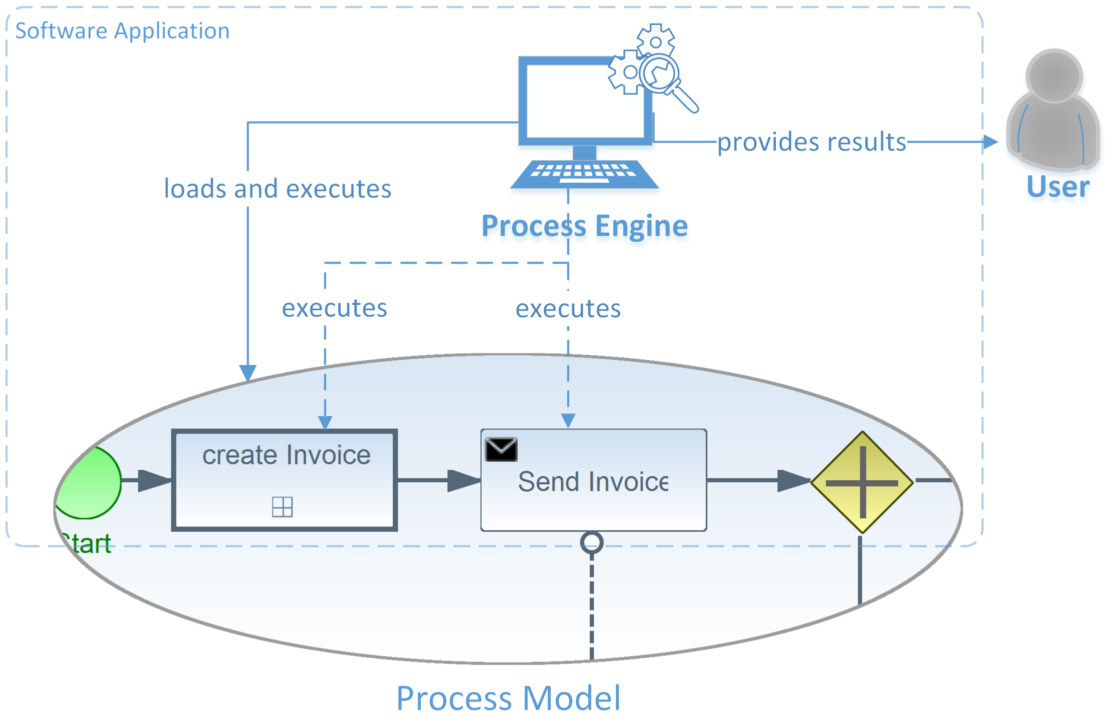


Figure 2.1: Simplified Process-centric Approach to Software

In a process-centric software, some tasks and their implementing *services* are invoked and executed automatically by the process-engine. To accomplish this, there has to exist a defined interface between the services and the engine. Services can be realized with various technologies and, in essence, they are just software components that are invoked. Recently, especially with the rise of microservice-based architectures [10, 99], Web Services have become the de facto standard for service implementations. Web Services are self-contained, self-describing components that can be invoked and provide some sort of functionality [39]. They provide a machine-processable interface description which enables machine-to-machine interaction over a network. The interface exposes the functionality of the service and can be consumed by others by sending messages [41]. There exist various ways [85] to implement and expose Web Services, e.g., Representational State Transfer (REST) [33] or Web Service Definition Language (WSDL) [24].

Software with an architectural design that heavily relies on (Web) services and is distributed across a network, is commonly referred to as implementing Service Oriented Architecture (SOA) [29]. SOA allows finding services through a discovery mechanism and invoke them during runtime, thus making the software very flexible as services can be exchanged with ease [120]. This approach is also commonly utilized in a microservice architecture [116, 28] where the software consists of several services instead of a single monolithic implementation. SOA facilitates *reusing* (Web) services.

However, there is still the problem in which order (Web) service are to be executed. In general there are two options available to organize (Web) services: *Choreography* and *Orchestration* [86]. In choreography, no central unit exists that would control and call all (Web) services, but the (Web) services rather act autonomously and communicate with each other via messages. This approach can also be combined with process models as shown in [119]. Orchestration, in contrast, uses a central unit that controls the execution of the (Web) services. This approach is closely related to process-centric software and process models as the software controls the flow of (Web) services. Most process-centric software applications work according to the orchestration paradigm.

So, combining process-centric software and their models with SOA and (Web) services, addresses the problem of organizing flow of services [51, 102]. It is important to note that SOA and process orchestration models are not equivalent, but rather use each other to enable process-centric software with distributed services [9, 32]. SOA facilitates reusing (Web) services in different processes, through a registry or a service repository, and models of processes orchestrate the order in which (Web) services are executed. Figure 2.2 schematically illustrates how process-centric software can be combined with (Web) services.

2.1.1 Conceptual (Business) Process Models

Specified process models are the foundation of flexible process-centric software applications. They specify the order in which tasks are executed and, by doing so drive the logic of a process-centric application. Davenport [25] describes a process as follows:

> “In definitional terms, a process is simply a structured, measured set of activities designed to produce a specified output for a particular customer or market. It implies a strong emphasis on how work is done within an organization, in contrast to a product focus’s emphasis on what.”

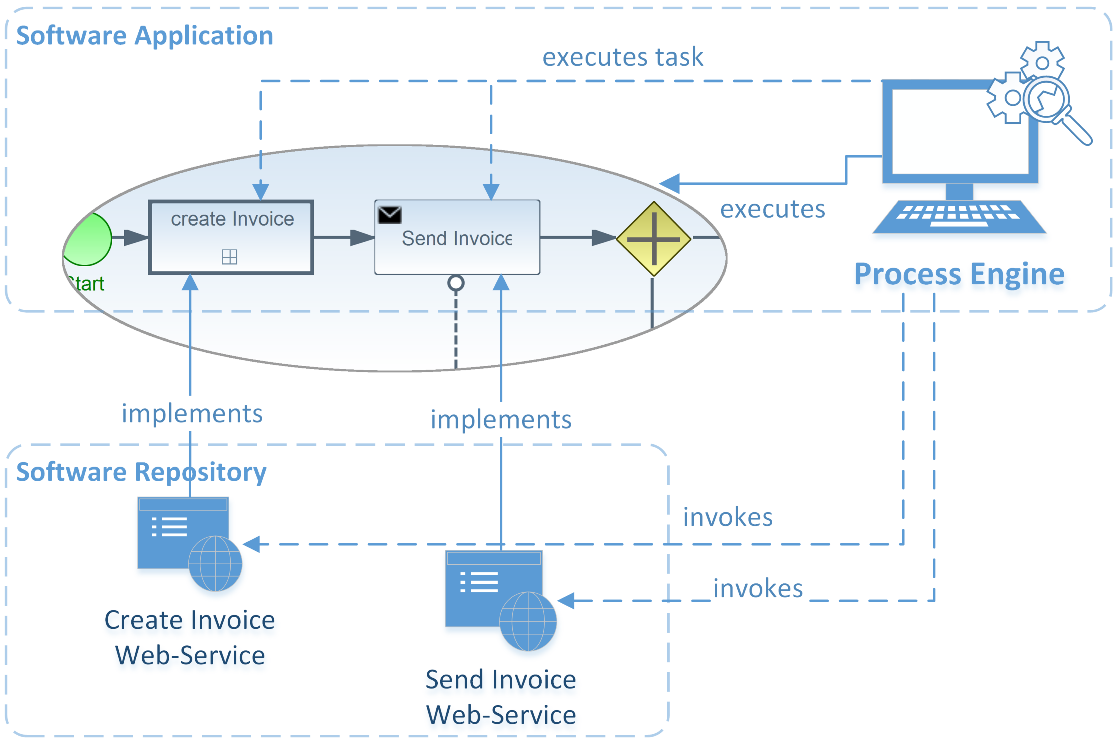


Figure 2.2: Combining Process-centric Software with (Web) Services

This also applies to so-called Business Process Models (BPMs). A BPM is a special kind of process model that is used in business applications of an enterprise and typically deals with creating or handling assets of the enterprise, i.e., it operates on *Business Artefacts*.

Process models, however, not only specify the flow of tasks that are executed, but also provide additional elements, which further define the process. Generally, process models specify in which order, by whom and with what tasks are being executed. That is, they provide the means to specify *roles* for operators as well as inputs and outputs of tasks. The order of execution is not fixed and may vary depending on the concrete enactment of the process. That is, a process model provides the means for conditional branching and, in some cases, parallel execution of tasks. Hence, each enactment of a process might take a different path to reach its end. In addition, process models typically provide some sort of event specification. They are used to handle internal events, e.g., a timer for errors, or to synchronize the process model with external models, e.g., waiting for an incoming message. Often there are many more features supported [14, 42], but for the sake of this thesis these features are sufficient.

Not all process models are constructed the same way and, depending on what they focus on, they can be differentiated. There are two major paradigms to (business) process modeling: *Task-centric* and *Artefact-centric* [68].

A task-centric approach is what is commonly assumed when talking about process models. Its emphasis is on connected tasks and what they are doing. For example, an invoice might first be created, then a check on possible delivery options is performed and finally it is sent to the customer. Figure 2.3 illustrates such a simplified task-centric process model.

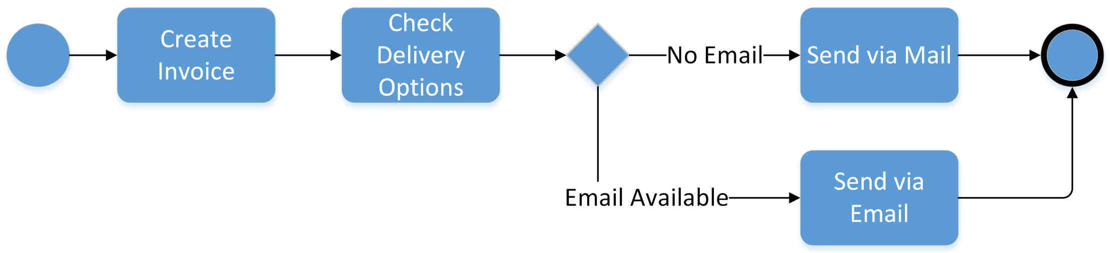


Figure 2.3: An example of a task-centric process model

As this process model illustrates, not all tasks are directly related to or advance the invoice, e.g., "Check Delivery Options". Moreover, the process only states in which order tasks should be executed, but not what tasks could possibly be executed. For example, it could be necessary to update the information on the invoice, which could be performed at any point before sending it to the customer. However, since the process model does not specify this option, it is also not available.

In contrast, an artefact-centric approach has its focal point on what can be done with an artefact. A similar process as described above, could be modeled in a artefact-centric approach as well. In this case, each action performed would be related to an artefact and its execution would advance the artefact. Figure 2.4 shows a simplified artefact-centric process model.

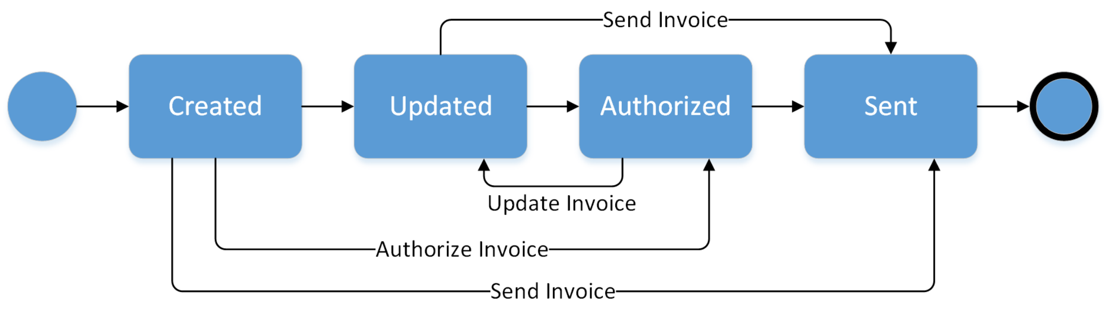


Figure 2.4: An example for an artefact-centric process model

As the figure points out, all possible options of subsequent tasks are available at all times. The focal point is on what can be done with the artefact rather on what should be done next [107]. In fact, such artefact-centric models are commonly used to express life cycle of artefacts. Each connection of two states in such a life cycle is performed by an action.

In essence, a task-centric approach focus on what *should be done and when*, and, in contrast, an artefact-centric approach puts its focus on what action *could be performed*. Which paradigm is a better fit depends on the specific use case and on the operationalization of the process model. There are approaches available that allow transforming a process model of one paradigm to the other [68].

There exist various modeling notations and languages that enable the specification of process models [70]. Ko et al. [55] did a survey on many of these languages.

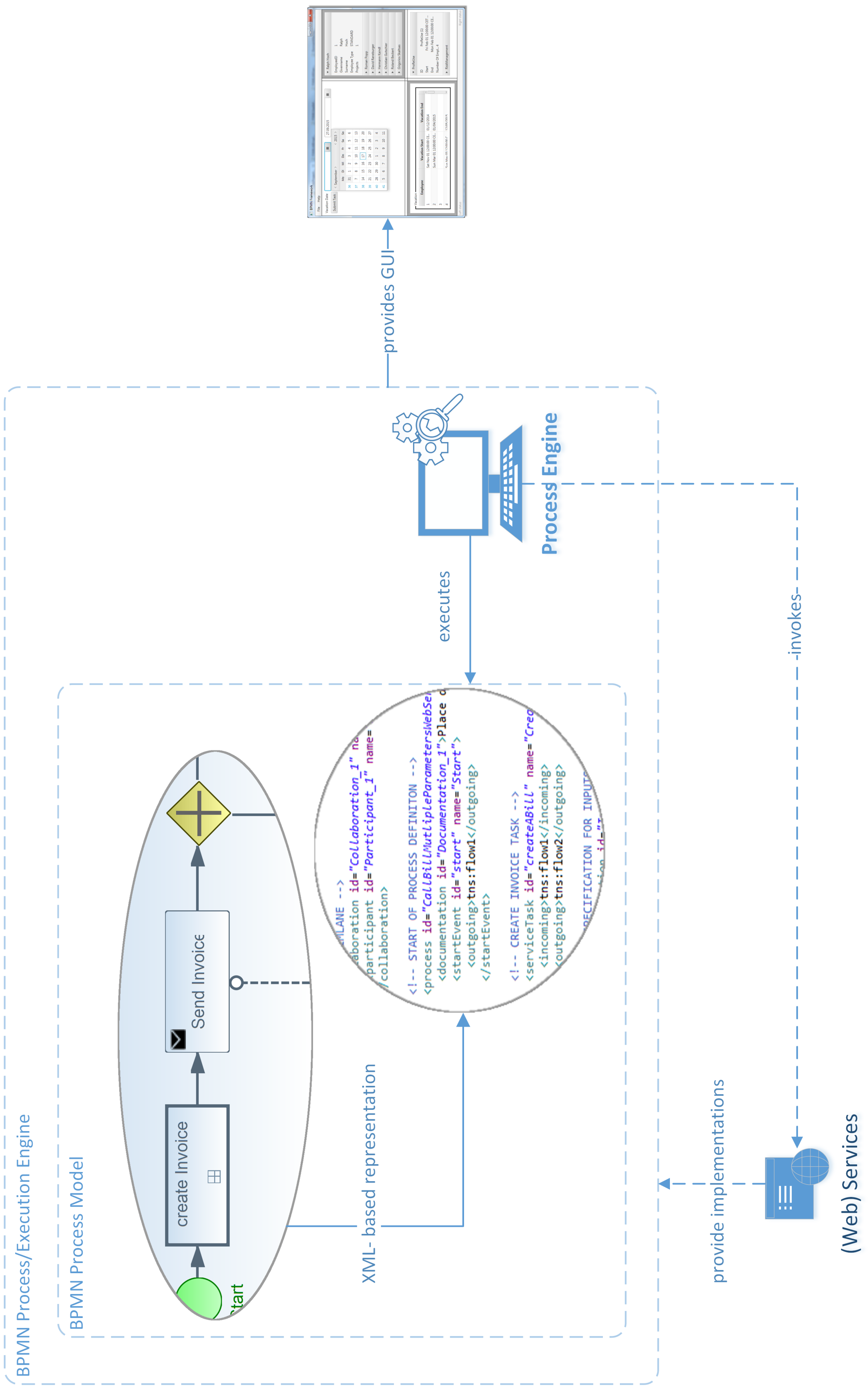
2.1.2 BPMN

BPMN [81] is a standard that provides a graphical notation, based on flow-chart techniques, for modeling processes. Early versions up to 1.2 only provided a graphical representation of the process models but lacked the possibility to execute the models by a process-engine. It utilized the XML Process Definition Language (XPDL) [113] standard to provide an interchangeable format that could be translated to Business Process Execution Language (BPEL) [79] [83]. BPEL is a standard execution language for BPMs with Web Services as their actions that uses a block-structured approach for its process models. This is in contrast to BPMN, which uses a directed-graph approach, thus making the translation of BPMN models to BPEL nontrivial [84].

In 2011 version 2.0 of BPMN was introduced, which addressed many shortcomings of earlier versions [23]. It introduced a meta-model, which provides a well-defined specification of the standard, extended the graphical notation for BPMs and introduced a machine-processable representation of the graphical presentation of BPMs based on Extensible Markup Language (XML). This machine-processable representation can be used by process-engines to execute BPMs alongside with their task implementations. The implementation of tasks can be provided by, but is not limited to, Web services. Figure 2.5 gives a schematic overview of how a BPMN 2.0 process execution engine operates.

There are many tools available that support BPMN, but to be fully BPMN-compliant four types of conformance (Process Modeling Conformance, Process Execution Conformance, BPEL Process Execution Conformance, and Choreography Modeling Conformance) have to be fulfilled. Most tools only support a subset of these and focus on Process Modeling Conformance and Process Execution Conformance [38]. Unfortunately, there is currently no reference implementation available, and models are often not interchangeable between tools as they are vendor specific [37].

We decided to use BPMN 2.0 as the modeling notation for our BPMs as it is the de facto standard for (business) process modeling that is supported by many vendors, e.g., IBM or Bonitasoft. In addition, BPMN has a built-in extension mechanism through which new elements can be introduced or standard elements can be extended. Further details on BPMN can be found in [2] or the official standard [81].



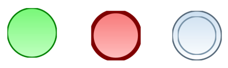
```
MLANE -->
aboration id="Collaboration_1" na
participant id="Participant_1" name=
/collaboration>

<!-- START OF PROCESS DEFINITON -->
<process id="CallBillMutlipleParametersWebSe
  <documentation id="Documentation_1">Place d
  <startEvent id="start" name="Start">
    <outgoing>tns:flow1</outgoing>
  </startEvent>

<!-- CREATE INVOICE TASK -->
<serviceTask id="createABill" name="Crea
  <incoming>tns:flow1</incoming>
  <outgoing>tns:flow2</outgoing>

ECIFICATION FOR INPUT
```

Figure 2.5: Schematic overview of a BPMN 2.0 process execution engine

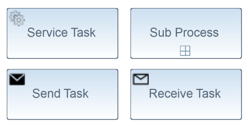
2.1.2.1 Elements of BPMN Models and their Graphical Notation

Most BPMN models consist of only a few basic elements. This is aligned with the intent behind BPMN to keep the representation of the models simple and clear. Overall, BPMN organizes elements in five categories with each category containing several groups of elements. The following list contains a short description of each group in the categories and also shows examples of their graphical representation on the right.

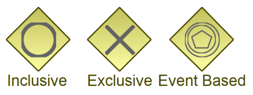
1. Flow Objects
   are essentially the nodes, which are connected to each other in one way or another, in a process model.
   - Events
     are something that occurs during execution of a process and they influence the flow of BPMs. Often they are triggered by external, collaborating entities or by internal triggers. They can have different roles depending on the event type. Typical elements are message receiving or error catching events. They are also used to signal the start or end of a process. Their representation is a circle with an embedded symbol (depending on the type of event).

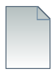
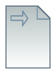
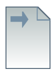
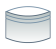
     

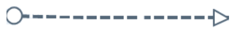
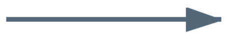
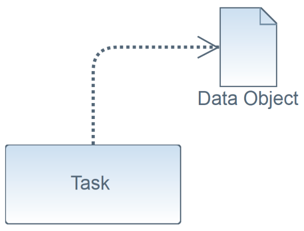
   - Activities
     are elements that perform some sort of work in a process. There are two distinct types of activities: *tasks* and *sub-processes*. A task is a basic or atomic activity, which is an activity that cannot be further divided. There are several specific *types* of tasks with different functionality, e.g., service-task, available. A sub-process is an activity which is composed of several activities. Activities are symbolized with rounded rectangles and a symbol representing its type.

   - Gateways
     influence the sequence flow of nodes in a process. Depending on the type of gateway the sequence flow can branch, fork, merge or join. Gateways can have multiple incoming (merge or join) or multiple outgoing (branch, fork) sequence flows and provide conditions on them. There is also a gateway for parallel execution of several sequence flows. Gateways are illustrated by a diamond shape with an internal marker (depending on the type of behavior control).

2. Data
   represent information about data that is used throughout the process.
   - Data Objects
     are elements used for storing information and can be passed from node to node. Often they are used to represent (business) artefacts that a process operates on, but this is not essential. They are symbolized by a sheet of paper.
     
   - Data Inputs
     are specific data objects that are used as inputs for nodes. They are symbolized by a sheet of paper with an incoming arrow.
     
   - Data Outputs
     are specific data objects that are used as outputs of nodes. They are symbolized by a sheet of paper with an outgoing arrow.
     
   - Data Stores
     are used as a storage for data objects. The graphical notation is a database symbol.
     

3. Connecting Elements
   are used to connect elements with each other.
   - Sequence Flows
     connect nodes with each other. They are the typical flow elements of a BPM as they indicate control flow from task to task. They can be enriched with data objects as well. Graphically they are represented by arrows.
     
   - Message Flows
     are used to model external relationships or communication with other processes. In addition, they provide the means for triggering events. They are symbolized by broken lines.
     
   - Associations
     connect additional elements, such as data objects or annotations, with other elements. They are represented by dotted lines.
     

4. Swimlanes
   help to organize BPMs and allow splitting one model into more than one process or participant.

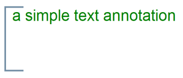
- Pools
  typically represent single processes in the model. They can be further divided by lanes to organize a process with several participating roles. They are represented by rectangles.

- Lanes
  organize a process, or in general a pool, with several participating roles. Each lane represents one role.

Participant
Lane 2
Lane 1

5. Artifacts
   provide additional information about the process.
   - Group
     A group is used to visually organize several elements in a single element. They are represented by a rounded, dotted rectangle.

   - Text Annotation
     gives additional information about elements in the BPM. Annotations are not processed by an execution engine.



There are some intricacies about some of these elements, which are necessary for understanding the remainder of this thesis. In particular, the activities need some more explanation. We utilize send- and receive-tasks throughout this thesis as they enable several processes to communicate with each other. A send-task sends asynchronously, i.e., it continues immediately, a message, via a message flow, to a collaborating process. The counterpart is a receive-task, which waits for an incoming message and only continues when it receives one. These tasks may also perform an additional action.

The flow of data objects can be visualized in BPMN models. However, this rather clutters the graphical model and thus we omit them from our models. Another reason is, that data objects are often only passed from one activity to another or transferred via a message. If the data flow is not obvious, e.g., by the title of an activity, we mention it in the corresponding description. Although there is no visual representation of the data flow, there is still a specification of the data flow in the XML representation of the BPM.

2.1.2.2 Extending Elements and Using Custom Elements in BPMN

BPMN allows extending its specification with custom elements as well as extending existing elements, since one specification for BPMs cannot possible satisfy all custom requirements in every domain. Each domain or application may have its own custom needs that a BPM has to fulfill and thus BPMN supports this in a controlled manner through its *Extension Mechanism*.

The extension mechanism can be used to make custom elements, e.g., the definition of a custom data type, available in the process model. If we consider our running example, then we could provide a formal datatype specification of our Invoice artefact. Even more so, we could provide additional information about such an element, e.g., references to an object life cycle. XML-based structures are preferred as they are easy to process and integrate nicely with the, also XML-based, BPMN specification. However, all types with a specification that the BPMN-engine understands, can be used. The same approach is used to import interface descriptions of WSDL Web Service or other BPMN documents.

Listing 2.1 shows an example of importing a custom element. It imports a specification of an Invoice, given as an XML-based data structure, into the BPM.

Listing 2.1: Import statement for custom elements in BPMN

```
<import importType="http://www.w3.org/2001/XMLSchema"
  location="InvoiceDefinition.xsd"
  namespace="http://ict.tuwien.ac.at/InvoiceData"/>
```

BPMN allows extending existing elements as well. In this case, the existing elements are not replaced, but only extended by custom properties or attributes. That is, both the elements without extension and the elements with extension can be used in the same BPM. This is accomplished by an *ExtensionDefinitions* tag, which is available on all elements. The definition of the extension has to be imported first and then may be used as needed. We utilize this extension definition, for example, to provide additional information for tasks on their prerequisites.

Listing 2.2 shows a custom extension for service-tasks, which establishes a link to an external resource with additional information, e.g., a taxonomy.

Listing 2.2: *Select Vacation User Task* in running example

```
<serviceTask id="transmitInvoice" name="Transmit Invoice">
  <extensionElements>
    <ict:taskInformation
      id="transmitInoiceInformation"
      name="Additional Information for the Transmit Invoice Task">
        http://ict.tuwien.ac.at/taxonomy#TransmitInvoice"
    </ict:taskInformation>
  </extensionElements>
</userTask>
```

In this listing a new element *ict:taskInformation* is used to provide additional information. The definition of this element is actually not given in this listing, but is rather imported as shown in Listing 2.1. Elements may have several attributes, e.g. a name, and a content. We use the content for providing a link to an external resource.

2.1.2.3 Running Example in BPMN

Throughout this thesis we use BPMN to illustrate BPMs. Hence, we also have to provide a BPMN representation of our running example of Section 1.3. Overall, the processes shown in Figure 1.3 remain the same in BPMN as Figure 2.6 shows.

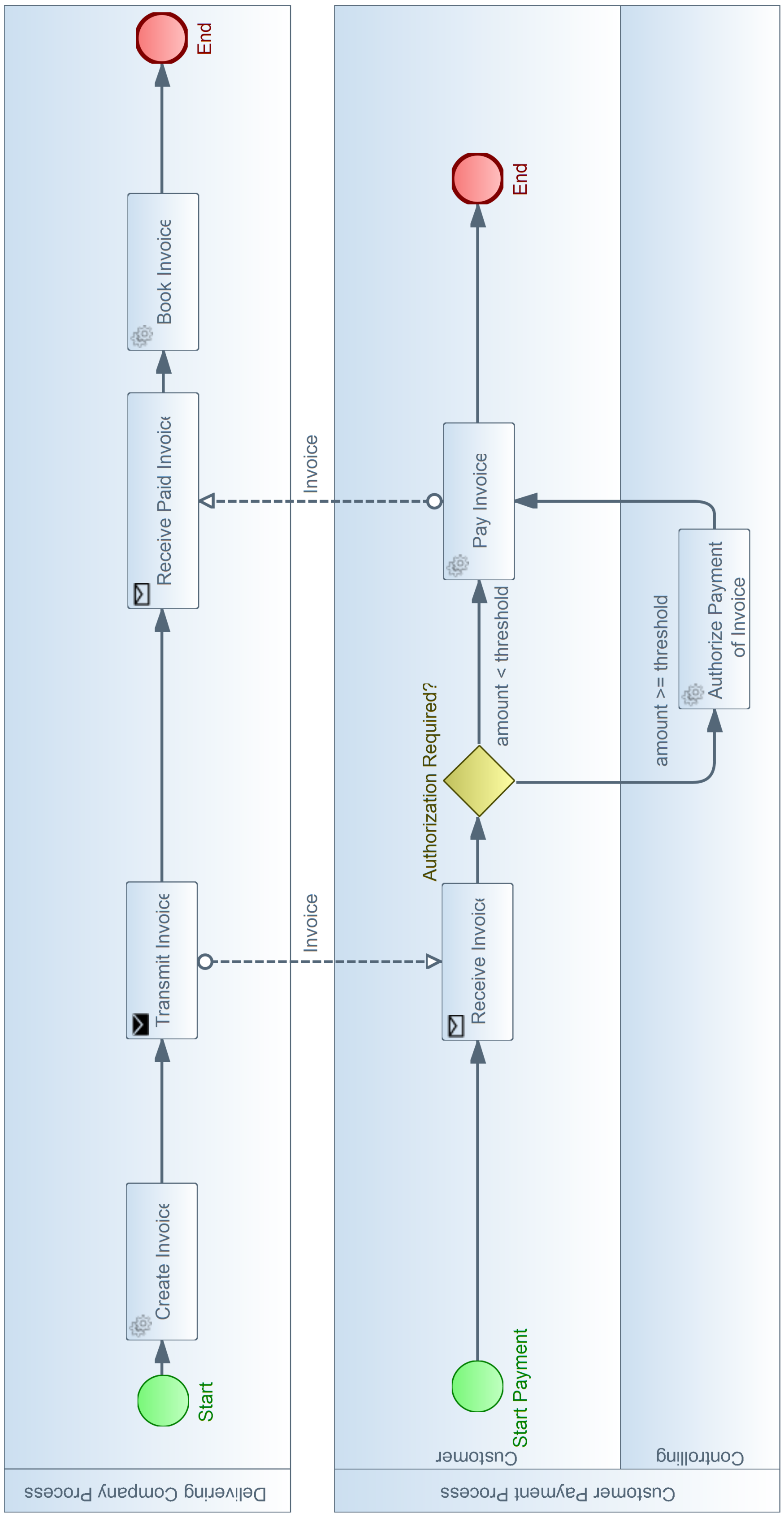


Figure 2.6: Payment Handling Process in BPMN, Including Authorization

Each participant of the running example is implemented by a single process (delivering company process and customer payment process) represented by a pool in BPMN and each pool has exactly one start and end event. All steps in the processes are represented by activities, more precisely tasks. We use *service-tasks*, implemented by Web services, for tasks that perform some form of action.

As mentioned before, the two participating process are operating synchronized and all synchronization operations are expressed via *send-* and *receive-tasks*. For example, the "Transmit Invoice" task is a send-task that sends a message from the delivering company process to the customer payment process. Since these tasks operate asynchronously the delivering company process continues immediately. However, the customer payment process is blocked in the "Receive Invoice" task until a message is received.

The conditional execution of the authorization step is accomplished by using an *exclusive-gateway*. Exclusive-gateways ensure that only one of its outgoing paths is executed at a time. The conditions for choosing the "right" path is expressed by the guard-conditions on the outgoing sequence flow of the gateway, e.g., amount < threshold. The "Authorize Payment of Invoice" is only executed if the amount of the invoice is greater or equal than the threshold.

The "Pay Invoice" and "Receive Paid Invoice" tasks are used to synchronize the two processes. Both processes end when they reach their one end event.

2.2 Object Life Cycles of Business Artefacts

Giving a clear definition of what a *Business Artefact* is, may differ depending on the environment. In Business Processes (BPs), businss artefacts are often considered to be the objects that are used within a specific domain and provide some sort of information. Nigam and Caswell [75] define business artefacts as:

> "Any business, no matter what physical goods or services it produces, relies on business records. It needs to record details of what it produces in terms of concrete information. Business artifacts are a mechanism to record this information in units that are concrete, identifiable, self-describing, and indivisible. We developed the concept of artifacts, or semantic objects, in the context of a technique for constructing formal yet intuitive operational descriptions of a business."

Generally speaking, business artefacts are the objects that are used in the business domain. Considering BPM, the *data objects* involved are the representation of business artefacts in the process specification. They typically reference a structure, for example a class or a schema definition, and should be automatically processable. During process execution these references are resolved and the actual data is gathered from the corresponding storage facility where the business artefacts are managed.

It is important to note that we postulate here the *closed world assumption*, i.e., that all the relevant knowledge indeed can be represented. While this certainly would not be justified, e.g., for robotics in real-world environments, Enterprise Resource Planning (ERP) systems actually define in their databases what is officially relevant for (real-world) businesses using them. What is not represented there can neither be officially booked nor handled.

We also employ models of object life cycles (as sometimes used for business process modeling). Their states typically correspond to achievements with regard to these objects in the course of the overall (business) process. A data object is characterized by its states and state transitions represented as an object life cycle, where each one describes the allowed behavior of a distinct class of data objects [68].

For formal verification making use of object life cycles, it is necessary to have a formal specification of these object life cycles. The concept of an object life cycle is independent of the domain that it is used in, and the concepts from the domain need to provide their own instantiation of an object life cycle. That is, each domain object has its own definition of its life cycle. We use ontologies written in Web Ontology Language (OWL) to specify object life cycles of business artefacts in our (business) domain declaratively. Figure 2.7 shows an object life cycle for an *Invoice* of the customer payment process of our running example as an FSM.

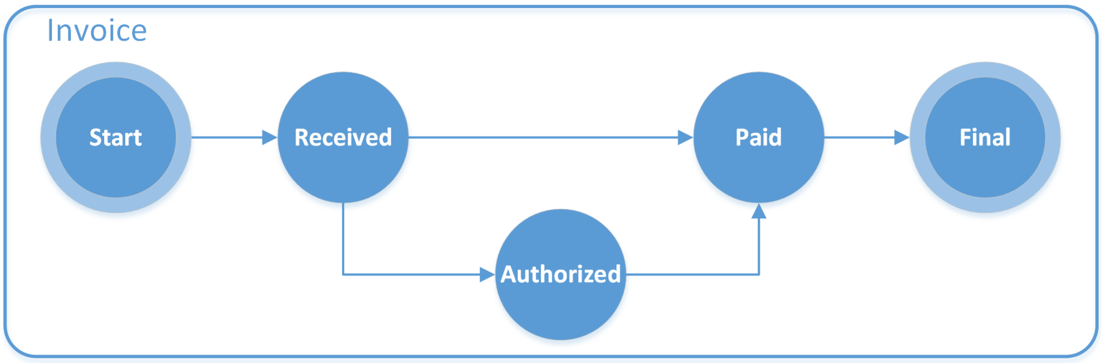


Figure 2.7: Invoice Object Life Cycle FSM

This FSM does not contain any signals among states yet. In our approach, such an object does not change its states by itself, but the transitions are triggered from the BPM using an action making changes to the object. In essence, the object life cycle specifies how a business artefact may evolve over time.

Ryndina et al. [94] present an approach to check the consistency of BPMs and object life cycles of artefacts. Each artefact that a BPM operates on is represented by a corresponding object life cycle, both a given one and another one automatically generated from the BPM. Consistency between the BPM and the given object life cycle is checked indirectly by formally comparing the latter with the automatically generated object life cycle. In contrast to our approach, Ryndina et al. do not consider specific contexts in which a process is enacted. In particular, they do not connect a given process model with a given object life cycle using semantic task specification as we do.

2.3 Semantic Service Specification

Semantic service specification has been based upon the OWL [66], which is a knowledge representation language used to build and administer ontologies or a specific knowledge base.[1] Semantic Markup for Web Services (OWL-S) [40] is an ontology built upon OWL for semantic descriptions of Web services. OWL-S consists of three parts: *Service Profile*, *Process Model*, and *Service Grounding*. The latter provides the means for interoperability with a Web Service Description (given in WSDL) and relates the semantic specification of a Web service with its WSDL file. This involves the definition of the input and output parameters including their types. In addition, OWL-S provides pre-defined predicates for defining preconditions, result values and effects. Services can be modeled either as atomic or as more complex composite services, where the latter consist of several (orchestrated) atomic services.

To illustrate how semantic specification works, let us consider the task “Pay Invoice” of our running example. Within this task, an *Invoice* is to be paid, and after that passed along according to the control flow in the BPMN model. The task itself is specified through its input and output, but it lacks a semantic specification that describes what the task accomplishes. The output alone is insufficient, as it only specifies the result of the task in the form of a type (in this case, an Invoice). However, it is not specified what kind of changes occur during task execution and how the domain in which the BP is enacted, is affected. This additional specification can be provided using the OWL-S formalism.

The *hasResult* predicate of OWL-S specifies the result of a service, where it couples both outputs and effects. Outputs are passed along from the service and correspond to an output variable from, for example, a WSDL file specifying a Web service. In addition, effects specify how the domain changes. To be more precise, they specify the changes that are caused by the service execution. Effects are specified with the *hasEffect* predicate. A semi-formal specification for the “Pay Invoice” Task (as inspired by [47]) is presented in Listing 2.3.

Listing 2.3: Semi-formal *Pay Invoice* Specification

```
Pay Invoice:
  Input:          Invoice
  Output:         Invoice
  Precondition:   --
  Effect:         paid(Invoice)
```

This task operates on an Invoice, which is passed to it as an input. It produces an output, again an Invoice, and also the changes in the domain are modeled as effects and specified using the *hasEffect* predicate.

Most times we also need formal condition specifying if a service can be executed. Listing 2.4 shows this additional precondition in bold face.

[1]Web Ontology Language: http://www.w3.org/TR/owl2-overview/

Listing 2.4: Semi-formal *Pay Invoice* Specification with Precondition

```
Pay Invoice:
  Input:          Invoice
  Output:         Invoice
  Precondition:   authorized(Invoice)
  Effect:         paid(Invoice)
```

Here, there is a formal condition (precondition) is specified on the input, which has to be fulfilled before the service can be executed. The precondition states, that the Invoice has to be authorized before a payment.

Semi-formally, a simple example of the task "Create Invoice", where no precondition and no input is needed, is shown in Listing 2.5.

Listing 2.5: CreateInvoice Task

```
CreateInvoice:
   Input:          --
   Output:         Invoice
   Precondition:   --
   Effect:         created(Invoice)
```

In our approach, semantic service specification plays a major role, since it provides a *declarative* representation of each service. It is the basis for *semantic task specification*, also partly inspired by the annotations of BPMs by Weber et al. [111]. They addressed the problem that control flow does not capture what the process activities actually do when they are executed. So, they annotated individual activities with logical preconditions and effects, specified relative to an ontology with axioms of the underlying business domain. This allowed them to verify the overall process behavior, but they did not utilize semantic task specification in the context of model checking as our approach does.

In this thesis, we formalize such a specification in two ways, using the Fluent Calculus, and predicates grounded in an object life cycle, respectively.

2.4 Business Rules

The concept of conditional (business) rules is extensive and to provide a clear, definitive definition proves to be difficult. According to Huang [49], business rules can be seen as operational rules that describe how an organization performs miscellaneous tasks. A similar specification has been given by Ross [93], where business rules are defined as the basic knowledge of a business including terms, facts and rules. The Business Rule Group defines business rules in the report [104] "Defining Business Rules – What Are They Really?" as statements that define or constrain the business of an organization and classify them into four categories. Their purpose is to influence the business process in a certain way.

Essentially (business) rule state that certain conditions yield a specific consequence. Such rules can be expressed in a variety of notations, e.g., Semantics of Business Vocabulary and Business Rules (SBVR) [80]. We chose Semantic Web Rule Language (SWRL) as a representation language for our rules, since it is also compatible with specifications of objects in our domain ontology given in OWL.

Each SWRL-Rule consists of a head and a body. The head is deduced if the conditions in the body evaluate to true. So, SWRL-Rules provide the means to deduce knowledge from existing facts. A typical example of an SWRL-Rule is the following relationship in families: if a parent of a child has a brother, than it can be deduced that this child has an uncle. Listing 2.6 shows how such a rule can be defined.

Listing 2.6: Example of an SWRL-Rule

```
hasParent(?child, ?parent) && hasBrother(?parent, ?brother) => hasUncle(?
    child, ?brother)
```

This notation is used since all the basic knowledge in OWL is specified in the form of triples. They consist of a *Subject*, a *Predicate* and an *Object*, where a predicate relates a subject to an object. The notation *hasParent(?child, ?parent)* states that the *?child* is in a relationship with a *?parent* through the predicate *hasParent*. In this example, the *hasParent* predicate is used to check if two individuals *?child* and *?parent* are related. They are only related if a triple of the form *?child :hasParent ?parent* exists in the knowledge base. The rule is checked for all available individuals.

There are several possibilities available to specify such SWRL-Rules. For example, they can operate on classes or on individuals (instances). One possible OWL representation of the rule above is shown in Listing 2.7. In this case, the rule operates on concrete individuals. These are identified via the *child* and *parent* variables, and all available instances in the domain are used.

Listing 2.7: Excerpt from SWRL-Rule Example

```
<swrlx:individualPropertyAtom swrlx:property="hasParent">
  <ruleml:var>child</ruleml:var>
  <ruleml:var>parent</ruleml:var>
</swrlx:individualPropertyAtom>
```

Such rules are often used to describe effects in OWL-S, as they show how the state of the domain changes. More precisely, the effects specify how the previous state of the domain is transferred to the new state after task execution. In essence, they allow the deduction of new knowledge based on the task execution.

2.5 Fluent Calculus

The original idea was introduced by McCarthy and Hayes [65] long time ago. Their Situation Calculus consists of three elements:

1. Situations
   represent the evolving states of the domain, where certain conditions hold in each state.

2. Actions
   represent the changes between situations. A special predicate *poss* determines whether a specific *action* can be performed or not.

3. Fluents
   represent the elements of the domain that can change over time. Typically, predicates are used for this representation, which take a situation as an argument. An example is the fluent *carrying(o,s)*, which states if an object o is carried, e.g., by a robot, in situation s.

Based on previous work on the Situation Calculus such as [91], Thielscher [105] developed the Fluent Calculus. It differs from the Situation Calculus in how situations are treated and how fluents are used. The Fluent Calculus defines that a new state after the execution of an action is equal to the previous state with exceptions to the effects of the action. In addition, fluents are treated as functional terms. The fluents from the Situation Calculus are stripped off the situation parameter, and special predicates, e.g., *holds*, are introduced. These special predicates take a functional term and a state as an argument. They are used to check whether specific conditions hold in a specific state or not. For example, the fluent *carrying(o,s)* from the Situation Calculus translates to a functional term *carrying(o)* in the Fluent Calculus. Hence, this term is not depended on the current state anymore. To check whether this term holds in a specific state *s*, the *holds* predicate is used, e.g., *holds(carrying(o), s)*.

Hence, the Fluent Calculus provides a formalism to model specific *actions* that lead from one situation to another. This is specified using the *poss* and *state_update* predicates. These predefined predicates model the preconditions (*poss* statement) and effects (*state_update*) of an action. Together, they provide a formal specification of an action.

To illustrate how such an action in Fluent Calculus is applied in *FLUX* [106], we use a simple example. Let us use our semi-formally given semantic specification in Listing 2.5 for illustration and explain its formalization in the Fluent Calculus in Listing 2.8, more precisely formulated in the language of the tool FLUX. In the Fluent Calculus each Action is specified via an *poss* and *state_update* predicate. The *poss* predicate checks if a specific action can be executed, i.e. that its preconditions are fulfilled. Since the "CreateInvoice" Action of Listing 2.5 does not have any preconditions, the body of the corresponding poss predicate in Fluent Calculus is also empty.

Listing 2.8: Semantic Specification of CreateInvoice Action in FLUX

```
poss(createInvoiceAction, Z). % Precondition %

state_update(Z1, createInvoiceAction, Z2,[]) :-
  update(Z1, [invoice(invoice), attributeSet(invoice, created)], [], Z2).
      % Postcondition %
```

The postcondition of the action is that the invoice has been created. Thus, the *state_update* predicate inserts these facts via the *invoice(invoice)* and *attributeSet(invoice, created)* predicates into the knowledge base. The first part is the *head* of the predicate *state_update* predicate. This *head* is separated from the *body* through the :– delimiter. The signature *createInvoiceAction* is used to identify the action and link it to the corresponding poss statement.

An *update* statement models state transfer and, in general, takes several arguments, where the second argument specifies the statements to be added to the new state and the third argument the statements to be removed from the previous state. These two arguments specify the add and delete list of predicates that are applied to the current state Z1 and form the new state Z2. This mechanism allows new predicates to be introduced as well as existing predicates to be removed from a state, in contrast to, e.g., predicate calculus. In effect, predicates or facts can change over time or more precisely after invocation of an action. Furthermore, this enables the calculus to negate existing facts or set negated facts to true. In this example, the second argument specifies that *attributeSet(invoice, created)* is to be added to the new state Z2, while the third argument is empty since nothing is to be removed.

As another more complex example, Listing 2.9 shows the action "TransmitInvoice" semi-formally.

Listing 2.9: TransmitInvoice Action

```
TransmitInvoice:
   Pre:  created(Invoice)
   Eff:  transmitted(Invoice)
```

Its formalization is analogous, but we show it in Listing 2.10, since TransmitInvoice has a non-empty precondition. This non-empty precondition is illustrated by the non-empty body of the poss predicate. The statement *holds(attributeSet(invoice,created), Z)* checks if, in the current state of the knowledge base, the fact *attributeSet(invoice,created)* holds.

Listing 2.10: Semantic Specification of TransmitInvoice Action in FLUX

```
poss(transmitInvoiceAction(Invoice), Z) :-
  holds(attributeSet(Invoice, created), Z),       % Precondition %
  knows_val([Invoice], invoice(Invoice), Z).              % Input %

state_update(Z1, transmitInvoiceAction(Invoice), Z2, []) :-
  update(Z1, [attributeSet(Invoice, transmitted)], [], Z2). % Postcondition %
```

The *body* of the poss statement consists of two constraints modeling input and precondition, which are logically connected with *and* each. Checking input is specified with the predicate *knows_val* and checking preconditions with the predicate *holds*. *knows_val* specifies a check whether for a given variable a value can be found in the current state. The predicate *holds* is for specifying whether a given predicate or literate holds, i.e., is known to be true, in the current state Z.

One additional predicate of the Fluent Calculus is *knows_not*. It specifies a check whether a given value is *not* known in the current state.

Such Fluent Calculus formulations provide the basis for formal *verification* of the semantic specification of a composed action against the semantic specifications of the single actions involved. For example, in a very simple action composition, first an invoice is created and then transmitted:

$$< CreateInvoice, TransmitInvoice >$$

The semantic action specification of this simple composed action is given semi-formally in Listing 2.11.

Listing 2.11: Composed Action

```
Sequence CreateInvoice, TransmitInvoice:
   Pre: --
   Eff: Invoice(transmitted)
```

The simple formulization in FLUX is shown in Listing 2.12. All steps are executed sequential and are and-connected. This means, that the result will only be available in state Z3 if all statements can be executed.

Listing 2.12: Composed Action in FLUX

```
  poss(createInvoiceAction, Z1),
  state_update(Z1, createInvoiceAction, Z2, []),
  poss(transmitInvoiceAction(Invoice), Z3),
  state_update(Z3, transmitInvoiceAction(Invoice), Z3, []).
```

In fact, we previously proposed a verification approach based on the Fluent Calculus already in [45], where all possible sequences of actions are exhaustively tried out by a planner and checked against a goal condition that is actually to be avoided. In this thesis, however, we use Fluent Calculus in combination with semantic action specifications for formal verification.

The FLUX implementation of the Fluent Calculus is available for the constraint logic programming system ECLiPSe.[2]

2.6 Model Checking

Model checking (or property checking) is a formal verification technique based on models of system behavior and properties, specified unambiguously in formal languages (see, e.g., [8]). The behavioral model of the system under verification is often specified using a FSM, in our case using synchronized FSMs. The properties to be checked on the behavioral model are formulated in a specific property specification language. Several tools (such as SPIN [100] or NuSMV [77]) exist for performing these checks by systematically exploring the state-space of the system. When such a tool finds a property violation, it reports it in the form of a counterexample.

In this work, we make also use of the branching-time logic *Computation Tree Logic (CTL)* for property specification [62].

Since a rough understanding of some of the CTL operators is needed for understanding our formalization approach, let us briefly sketch these here. CTL provides expressions of relations between states (path formulas) using operators referring to behavior over time. It allows modeling properties on the computation tree of a FSM. In CTL the set of traditional propositional logic operators is extended by operators such as:

- *AG* (Always Globally): an expression p is true in state s_0 if p is true in all states for all possible state transitions $s_0 \geq s_1, s_1 \geq s_2, \ldots$. [78, p.37]
- *EF* (Eventually Future): an expression p is true in state s_0 if there exists a series of transitions $s_0 \geq s_1, s_1 \geq s_2, \ldots, s_{n-1} \geq s_n$ such that p is true in s_n. [78, p.37]
- *AF* (Always Future): an expression p is true in state s_0 if for all series of transitions $s_0 \geq s_1, s_1 \geq s_2, \ldots, s_{n-1} \geq s_n$ p is true in s_n. [78, p.37]

2.7 Related Work

Salomie et al. [95] studied Web service composition using the Fluent Calculus, viewing automatically composing Services as an Artificial Intelligence *planning* problem. This service composition technique has been further discussed by Bhuvaneswari et al. [16]. While the techniques for automatically composing services and for verifying a given composition are closely related and both supported by FLUX, none of this previous work addresses our main topic — V&V of service composition and (business) processes.

[2]ECLiPSe Constraint Programming System:
`http://www.eclipseclp.org`

Another approach for semantic specification of services has been developed by Baryannis et al. [12], where an intermediate language (WSSL) is formulated. WSSL enriches the basic and standard service specification in WSDL with pre- and postconditions. These are actually based on the Fluent Calculus and could, in principle, be used for our verification purposes to V&V as well. In fact, our formulations of pre- and postconditions in FLUX as given above were informed by WSSL.

There have already been approaches for automatic *planning* algorithms based upon the OWL-S specification of services which try to automatically generate composite services out of atomic services, e.g., Klusch et al. [54, 53] and Ziaka et al. [118]. However, the approach based on the Fluent Calculus and FLUX seems to be preferable because of its well-defined semantics.

With respect to the real ends of our own approach built on FLUX, none of these approaches deals with additional knowledge required for the service composition that should *not* be embedded into the service specification itself.

Related research on implicit business knowledge (tacit knowledge) is still in its infancy. Chesbrough and Spohrer [22] emphasized in their research manifesto for services science that the nature of tacit knowledge complicates the services exchange — and service exchange is a corner stone for service composition. More precisely, tacit knowledge limits the ability of each service-party to fully comprehend the needs and abilities of each other. The authors point out that a multidisciplinary perspective toward services becomes increasingly important to gradually codify tacit knowledge. We actually found very simple cases where tacit knowledge needs to be made explicit for V&V of service composition and business processes, and we show that this is business knowledge that should not be encoded in the service specifications.

Montali et al. [71] introduced an alternative way of specifying service choreographies by directly defining them through a set of policies referred to as constraints. These are embedded in their flow language, explicitly connecting services, e.g., with their time flow. Their approach uses Linear Temporal Logic (LTL) to verify conformance checking, conflicts and dead activities, interoperability between global and local models, etc. In contrast, we focus less on the choreography aspect but explicitly represent (business) rules declaratively and use the Fluent Calculus and Model Checking for verification of composed services against the specifications of the single services including (business) rules.

Feng and Kirchberg [31] proposed an approach for verifying properties of the process model of an OWL-S service. Via mapping rules, this approach translates the process model into a process algebra model and uses a model checker to verify the properties of such a translated model. It handles the control flow as well as the binding-based data flow of the process model. In contrast to our approach, implicit knowledge is not separated from the service specification. Our work makes the (business) rules explicit that actually glue services together semantically in a BPM.

Li et al. [117] used Propositional Logic for requirements verification of service workflow, where requirements include business rules. Their work is capable of checking compliance and also of detecting conflicts of imposed requirements. However, its focus is mainly on compliance checking between a Service Workflow Net (SWN) and SWSpec formulas. This is different from our research on whether semantic service specification is sufficient for V&V of service composition and BPMs.

Ni and Fan [74] transformed models and formally verified semantic Web services composition. More precisely, their approach verifies the correctness of semantic Web services composition based on models of Colored Petri Nets that are transformed from OWL-S models. It is sound to use such Petri Nets in order to verify reachability and soundness of composed services (among others). This approach differs from ours as it does not address our main topic — V&V of service composition and BPMs.

In the dynamic field of late binding and runtime verification of business processes, Angelis et al. [4] introduced automatic test-case generation aiming at checks of the behavior of services participating in a given orchestration. Assuming that the business process is available as a runnable model, their approach applies model-checking techniques to derive test cases suitable to detect possible integration problems. Our work does not rely on tests at runtime to find flaws, but uses formal logic at design-time for automated verification.

In the context of verification of BPMs, Weber et al. [111] addressed the problem that control flow does not capture what the process activities actually do when they are executed. So, they annotated individual activities with logical preconditions and effects, specified relative to an ontology with axioms of the underlying business domain. This allowed them to verify the overall process behavior, but without making business rules explicit as such and dealing with them separately from the activities. In contrast, Deutsch et al. [26] studied automatic verification of data-centric business process specifications. Their results suggest that significant classes of data-centric business process specifications may be amenable to automatic verification.

Business rules do not necessarily need to specify how an organization performs tasks but can also describe technical aspects. Orriens et al. [82] describe in their approach how business rules can be categorized and how such rules can be used for Web Service orchestration. They show that it is not sufficient to embed business knowledge in an orchestration language (like BPEL) directly, but that business rules need to be specified explicitly. This issue has also been discussed by Rosenberg and Dustdar [92], who show the need for integrating business rules into BPEL, since they are often changed and thus a rule-based system should be used. Eijndhoven et al. [108] provide a similar motivation with the focus of business rules on action rules. Wu et al. [114] propose a rule-based scheduling engine that can be embedded into frameworks. The idea is to use rules as role-assignments, which are used to influence process execution, message exchange, and flow constraints. None of these approaches deals with formal verification based on logic, however, like our work.

Lovrencic et al. [61] describe business rules as essential parts in today's business system model and the need for their formalization. Two approaches based on UML and ontology-based modeling are discussed in this work.

Earlier work on the interplay of business process models, service/task ontologies, and domain ontologies was carried out in the project SUPER (Semantics Utilised for Process

Management within and between Enterprises) [101], where a tool named “Maestro for BPMN” was developed. Born et al. [17] describe how user-friendly semantic annotation of process artifacts with tags/markups can be achieved in BPMs via Maestro. These annotations refer to semantics in domain ontologies, and based on them, this tool allows one to automatically compose activities within business processes [19]. Maestro also supports certain consistency checks of the control flow against semantic annotations of such annotated processes [19]. Born et al. [18] describe how “adequate” services can be identified for specific tasks through match-making by use of the semantic annotations.

Burkhart et al. [20] define in more recent work a *structural description of business models*. Their synthesis of eight existing ontologies and extend this knowledge with state-of-the-art research progress on business models. This work proposes transformation of such structural descriptions to business process models, which is a different but possibly complementary approach to ours.

Marzullo et al. [64] proposed another integration effort, with the purpose of supporting domain-driven software development. So, it centers around a shared domain specification to be used as a reference point for software applications. The central domain repository allows exchanging information in a standardized way between different projects or companies. So, the focus is clearly on efficient software development, even though Marzullo et al. [63] describe possibilities to include business process modeling as well. In contrast to our approach, the domain specification is not based on a formal specification language or ontology. Saiyd et al. [1] describe a similar approach to Marzullo et al., but propose an ontological foundation for domain-driven design. However, their work is more focused on the specification of the ontological concepts than their actual use. So, neither of these approaches has such a comprehensive integration and the scenarios of its use in mind that we propose in our work.

While BPMN 2.0 has, in contrast to the previous version BPMN 1.0, a defined meta-model, it is not based on a logic foundation. Therefore, Natschlaeger et al. [73] propose an OWL-based upper ontology for BPMN 2.0 to allow a formalized specification of BPMN 2.0 processes. Using it in our integration approach would certainly be possible and interesting, since it would make it completely based on ontologies. As it stands, however, our early feasibility prototypes indicate that using the meta-model of BPMN 2.0 should be sufficient for our currently envisaged scenarios of use.

Cabral et al. [21] show in their work how business process modeling can benefit from semantic information. They describe the ontology BPMO (Business Process Modeling Ontology), which includes semantic knowledge about organizational context, workflow activities and other business process parts. Using this ontology, it is possible to refer to semantically annotated data and services for working in a coherent way. In contrast, our approach uses BPMN 2.0 as modeling and orchestration language. In addition, we focus on combining BPMN with OWL semantics rather than representing business processes in an ontology.

Semantic Business Process Management (SBPM) helps handling the life cycle of business process management through ontologies and Web services, as proposed by Filipowska et al. [36]. They illustrate with various scenarios how SBPM can be used in the business process management area. They describe a set of ontologies for SBPM, which target the spheres of enterprise structures and operations. This work is based on *Web Service Modeling Ontology (WSMO)* [110] and its closely related representation language *Web Service Modeling Language (WSML)* [109] for combining semantic Web services with business process management [43]. In contrast, we use OWL-S for semantic specification of services, and BPMN 2.0 for BPMs, but we do not strive for representing (business) processes in an ontology.

Previous related work made it absolutely clear that some representation with defined semantics is a prerequisite for formal verification, also of (business) processes. Given such a representation, checking correctness properties inherent in the (business) process itself is possible. Wynn et al. [115] verify business processes against four defined properties (soundness, weak soundness, irreducible cancellation regions and immutable OR-joins). Sbai et al. [97] show how a model checker can be used to identify problems with a specification of a business process to be automated as a workflow, and how a verification of certain correctness properties can be accomplished. Kherbouche et al. [52] propose an approach for using model checking as a mechanism to detect errors such as deadlocks or lifelocks.

Some previous work addressed the question of what to verify a BPM against, to determine possible violations of certain properties given in addition to the process model itself. Fisteus et al. [5] propose a framework for integrating BPEL4WS and the SPIN and SMV verification tools. This framework can verify a process specification against properties such as invariants and goals through model checking. Armando and Ponta [6] show how model checking can be used for automatic analysis of security-sensitive business processes. They propose a system that allows the separate specification of the business process workflow and of corresponding security requirements. In more recent work [7], they show how model checking can be specifically used to check authorization requirements that are implemented in parts of business processes. Barros and Song [11] propose to check business processes against execution rules incorporated in workflows with model checking techniques.

Mrasek et al. [72] point out that formalizing properties in CTL is a difficult task and strive for making it easier through so-called patterns based on textual fragments in natural language. This approach can work in a given context for entering properties, and it helped in a case study. In general, however, the interpretation of these textual patterns is subtle and error-prone. So, they have to be prepared specifically for a given problem by CTL specialists, anyway. In particular, for our case study in [90] with given legal text, such an approach would most likely require a variety of different patterns and still be hard to validate.

Still, no previous work in the context of model checking of BPMs addressed systematically formalizing business rules and enriching tasks to our best knowledge, including model-based business process compliance-checking approaches [13]. Apart from [72], which addresses formalizing properties (but not formalizing the process), all the publications on model checking of (business) processes already assume the availability of formal representations.

Ligeza et al. [57, 58] consider a specification of business processes and business rules to a certain degree complementary. They tried to reconstruct BPMN in the logic programming language PROLOG to provide formal requirements on model correctness.

Lohmann [59, 60] presents an approach based on compliance rules, which are used to automatically create artefact-centric business processes that are compliant by design. The building blocks are life cycles of the involved artefacts.

Meyer et al. [68] define a "weak conformance" between process models and synchronized object life cycles. Their algorithm for soundness checking verifies whether each time an activity needs to access a data object in a particular state, it is guaranteed that the data object is in or can reach the expected state. They show that it is possible to transform process-centric BPMs into artefact-centric BPMs. They present algorithms on how this can be accomplished with synchronized object life-cycles. In contrast to our approach, they do not verify against additionally specified (business) rules.

Estãnol et al. [30] propose a verification approach based on artefact life cycles modeled in Unified Modeling Language (UML). It checks certain intrinsic properties such as liveliness of a class or an association.

In summary, we are not aware of any previous work that studied semantic service specification for V&V of service composition and (business) processes as we present it in this thesis, especially not by including (business) rules with semantic specifications into formal verification.

CHAPTER 3

Semantic Service Specification for V&V of Service Composition

Service composition is the process of creating a more complex (Web) service from other (Web) services. If such a service does not rely on any other services for its functionality, it is considered a simple or atomic service. Otherwise, we refer to it as a *composite service* [3, 67]. Examples could be services for receiving, authorizing and paying invoices, which are standalone/atomic. They form the new composite service "Processing Invoice" through service composition.

Much as any software, composite services and (business) software composed from services needs to be tested. This can (hopefully) find bugs and design problems but not provide any real guarantees. In contrast, semantic service specification based on formal logic allows for *formal verification of composed services* against the specifications of the single services. More precisely, our verification approach checks whether a defined sequence of service invocations is consistent with the semantic specifications of the services involved. In this sense, our verification means to formally and automatically check whether the composition is built right based on the single services.

In this context, we pose the question whether semantic service specification is sufficient for such a verification of composed services for implementing business processes. Verification based on logic (involving formally specified pre- and postconditions) may reveal loop-holes in the knowledge represented in the specification of a composed service with regard to the business process. Even though the verification of a composed service may be successful, its *validation* as a (fragment of) a (business) process may not. That is, the composition is not right as a (business) process. In such a situation, adding missing knowledge to the service specification may create a mismatch between this specification and the service implementation. So, we added specific *business rules* and achieved consistent results from V&V of service composition and (business) processes.

Our methodological approach to answering this question is to formulate the hypothesis that semantic service specification alone is sufficient for such a verification of composed services for implementing business processes. By providing an example where this is not the case, we can reject this hypothesis. For providing such an example, we use a simplified variant of the customer payment process of our running example for a small and a large company each (both just hypothetical for the purposes of this thesis), including the tasks Receive Invoice, Authorize Invoice and Pay Invoice, as well as their respective dependencies. For each of these tasks we assume implementations as Web services, which are to be composed accordingly. We assume that the authorization of the payment is unconditional, i.e., independent of the amount of the invoice, as this is sufficient for showing a V&V mismatch. This example is very simple on purpose. We claim that the issue shown with it will most likely be relevant for any real-world processes as well, since it even occurs in this simple example.

As a means to this ends, we employ given theories and their supporting technology. When using such knowledge represented in the Fluent Calculus [105], verification of a composed service can be done in the related tool FLUX.[1] It has fully defined semantics also of its reasoner, which can be employed for this verification. This entails the necessity to specify pre- and postconditions in the FLUX tool.

Still, we had to extend this approach for our purposes of V&V. We did this by additionally representing and including a certain kind of *(business) rules* here.

Figure 3.1 gives an overview of the technologies involved in the context of this V&V approach. We utilize an OWL-S *Semantic Service Specification Repository* that contains semantic descriptions of the (Web) services. Each entry in this repository describes one WSDL service. In addition, we may reference some sort of *Domain/Reference Ontology* where additional information, e.g., on artefacts, is stored. Using this information, we can automatically generate a *Fluent Calculus Knowledge Base* in FLUX containing a specification of each service. For composed services, we use BPMN and their *BPM Tasks*. Although service composition is strictly speaking not a (business) process, we can utilize BPMN for their illustration. Each *BPM Task* has a reference to an entry of the *Semantic Service Specification Repository* and, implicitly, to an implementing WSDL service. This allows us to systematically create a *Service Composition* to be verified in FLUX.

We consider V&V of services an integral part of service *design*. Our approach may serve as a holistic approach, since it integrates V&V of services and (business) processes [47].

3.1 Specifying Semantic Knowledge for Service Composition

Let us assume a very small (hypothetical) company for our customer payment process of our running example, where an *Invoice* is simply received first and then paid. There are two services involved for that, *ReceiveInvoice* and *PayInvoice*. Their semantic specifications are shown (somewhat simplifying, of course) in Listings 3.1 and 3.5.

[1]FLUX agent: http://www.fluxagent.org/

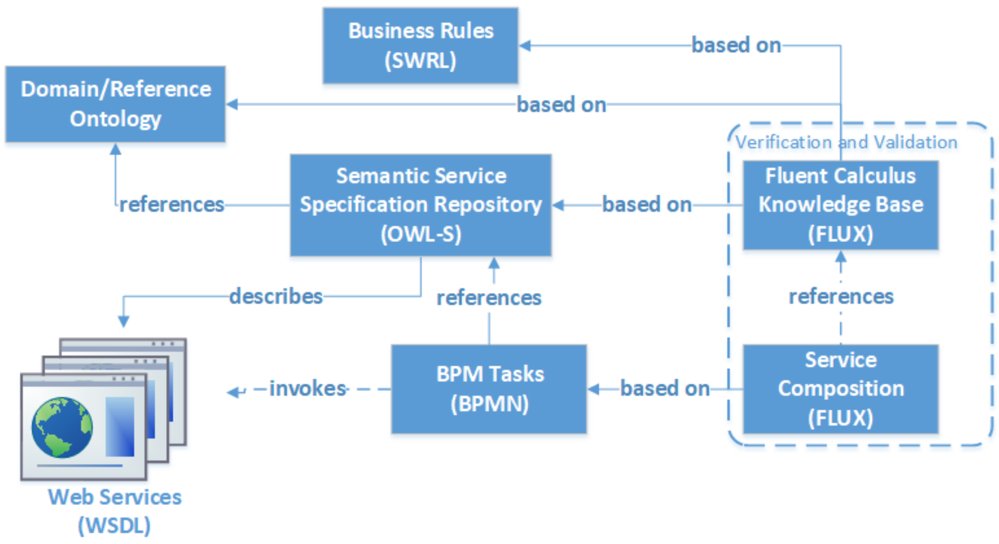

Business Rules (SWRL)
based on
Domain/Reference Ontology
based on
Verification and Validation
Semantic Service Specification Repository (OWL-S)
based on
Fluent Calculus Knowledge Base (FLUX)
references
describes
references
references
invokes
BPM Tasks (BPMN)
based on
Service Composition (FLUX)
Web Services (WSDL)

Figure 3.1: Technology overview for Verification and Validation

Listing 3.1: ReceiveInvoice Semantic Specification

```
ReceiveInvoice:
  Input:            Invoice
  Output:           Invoice
  Precondition:   exists(received(Invoice), false)
  Postcondition:    received(Invoice)
```

Listing 3.1 shows the semantic specification of the service that receives an invoice. In this case, the service has one input *Invoice*, which may be defined in a reference ontology, and outputs an *Invoice*. Additionally, a precondition states that there must not exist an already received invoice. The postcondition states that the service ensures that after its execution an invoice has been *received*.

The formulations of the *ReceiveInvoice* service in Fluent Calculus, more specifically in FLUX, is implemented using the *poss* and *state_update* predicates. The respective parts are shown in Listings 3.2, 3.3 and 3.4.

In Listing 3.2 the inputs and preconditions of the service represented in FLUX are shown. The poss predicate contains the name of the action *receiveInvoiceAction* and its input parameter *Invoice*. Inputs are checked via the *knows_val* predicate, which checks if for a given variable a value can be found in the current state. The precondition that no received invoice exists is checked through the *knows_not* predicate. The service may only be executed if all inputs and preconditions of the action given in FLUX are fulfilled.

Listing 3.2: ReceiveInvoice-Action preconditions encoded in FLUX

```
poss(receiveInvoiceAction(Invoice), Z) :-
  knows_val([Invoice], invoice(Invoice), Z),  % Input %
  knows_not(received(invoice(_)), Z). % Precondition %
```

If the preconditions are fulfilled, then the corresponding action can be performed. Listing 3.3 shows the update operation along with its outputs and postconditions. The action is specified by values that are added to and removed from the knowledge base. Hence, the action moves the knowledge base from one state to another by adding and removing information from it. In this case, a new invoice is added and the fact that this invoice has been *received*. This is formalized in the second parameter of the *update* predicate. The third parameter is a empty list since nothing is to be removed from the knowledge base.

Listing 3.3: ReceiveInvoice-Action postconditions encoded in FLUX

```
state_update(Z1, receiveInvoiceAction(Invoice), Z2,[]) :-
  update(Z1,
    [invoice(Invoice), received(invoice(Invoice))], %Output,Postcondition%
    [],
    Z2).
```

Finally, Listing 3.4 shows how the check if an action can be performed and its actual update statement are connected. The action of the poss and state_update statement is identified by its name and parameters.

Listing 3.4: Signature of ReceiveInvoice Action in FLUX

```
poss(receiveInvoiceAction(Invoice), Z),
state_update(Z, receiveInvoiceAction(Invoice), Z2, []).
```

In Listing 3.5, the semantic specification of the *PayInvoice* service is given. It demands, through its precondition, a *received Invoice* as its input. After completion of the service it is ensured that the invoice has been *paid*.

Listing 3.5: PayInvoice Semantic Specification

```
PayInvoice:
  Input:        Invoice
  Output:       none
  Precondition:  received(Invoice)
  Postcondition: paid(Invoice)
```

The FLUX specification is created analogously to the one for *ReceiveInvoice*. In order to show such a specification in one piece, we include here Listing 3.6. The precondition *received(Invoice)* is checked via the *holds* predicate, which checks if a certain value holds in a specific state Z. The state_update statement adds the *paid(Invoice)* information to the knowledge base.

Listing 3.6: PayInvoice encoded in FLUX

```
poss(payInvoiceAction(Invoice), Z) :-
  knows_val([Invoice], invoice(Invoice), Z),        % Input %
  holds(received(invoice(Invoice)), Z).             % Precondition %

state_update(Z1, payInvoiceAction(Invoice), Z2, []) :-
  update(Z1, [paid(invoice(Invoice))], [], Z2).     % Postcondition %
```

Algorithm 1 shows the translation of semantic service specifications to FLUX in pseudo-code.

Algorithm 1 Algorithm for translating Semantic Service Specifications to FLUX

```
1: procedure TranslateOWLS2FLUX($OWLSRepo$)
2:     $FLUXServices \leftarrow List < FluxService > ()$
3:     for ServiceSpec $service \in OWLSRepo.Services$ do
4:         $FLUXServices \leftarrow$ TranslateSemanticSpec2FLUX($service$)
5:     end forreturn $FLUXServices$
6: end procedure
```

The algorithm iterates through all service specifications given in OWL-S and calls a function that generates the corresponding FLUX counterpart using Algorithm 2.

It uses two templates, as shown in Listings 3.7 and 3.8 during the translation. The values in curly brackets are replaced by the values of the semantic service specification. Listing 3.7 shows the template for the preconditions and inputs.

Listing 3.7: Poss-Statement Template

```
poss({ActionName}({InputNames}), Z) :-
  {knows_val([{InputName}], {InputCondition}, Z)},  % Inputs: if available
     %
  {knows_not|holds}({precondition}, Z). % Preconditions: if available%
```

Listing 3.8 shows the template for the postcondition and outputs.

Listing 3.8: State_update-Statement Template

```
state_update(Z1, {ActionName}({Inputs}), Z2,[]) :-
  update(Z1,
    [{Outputs}, {Postconditions}],  % Output, Postcondition %
    [],
    Z2).
```

Algorithm 2 Algorithm for translating Semantic Service Specification to FLUX

```
 1: procedure TranslateSemanticSpec2FLUX($ServSpec$)
 2:     $possTemplate \leftarrow$ LoadPossTemplate($ServSpec.Name$)
 3:     for Input $input \in ServSpec.Inputs$ do$possTemplate.addInput(input)$;
 4:     end for
 5:     for Precondition $pre \in ServSpec.Preconditions$ do
 6:         $possTemplate.addPrecondition(pre)$;
 7:     end for
 8:     $updateTemplate \leftarrow$ LoadUpdateTemplate($ServSpec.Name$)
 9:     for Input $input \in ServSpec.Inputs$ do
10:         $updateTemplate.addInput(input)$;
11:     end for
12:     for Output $output \in ServSpec.Outputs$ do
13:         $updateTemplate.addOutput(output)$;
14:     end for
15:     for Postcondition $post \in ServSpec.Postconditions$ do
16:         $updateTemplate.addPostcondition(post)$;
17:     end for
18:     $FLUXService \leftarrow$ possTemplate.process();
19:     $FLUXService \leftarrow$ updateTemplate.process();
         return $FLUXService$
20: end procedure
```

For the small company, the obvious service composition resulting in a sequential business process is shown in Figure 3.2.

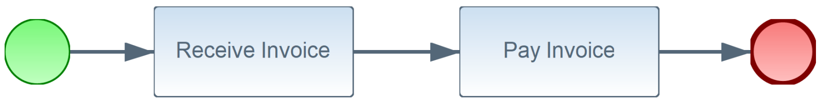


Figure 3.2: “Receive and Pay” Business Process of a small Company

In FLUX this service composition is simply expressed by executing the services in order as shown in Listing 3.9. Each service is expressed via its *poss* and *state_update* predicate and all statements are executed in succession.

Listing 3.9: “Receive and Pay” Business Process encoded in FLUX

```
% first check if the action receiveInvoice is applicable %
poss(receiveInvoiceAction(Invoice), Z),
% then perform the action through state_update  %
state_update(Z, receiveInvoiceAction(Invoice), Z2, []),

% check if payInvoice is applicable %
poss(payInvoiceAction(Invoice), Z2),
% and then perform its update %
state_update(Z2, payInvoiceAction(Invoice), Z3, []).
```

Algorithm 3 shows the translation of service compositions to FLUX in pseudo-code.

Algorithm 3 Algorithm for translating Service Compositions to FLUX

```
1: procedure TranslateServiceCompositonToFLUX($serviceComp$)
2:
3: $startService \leftarrow$ serviceComp.inital
4:     while $startService.hasNext$ do
5:         $service \leftarrow$ startService.next
6:         $FLUXServiceComp \leftarrow$ service.createPossUpdateStatement
7:     end while
8:     return $FLUXServiceComp$
9: end procedure
```

Listing 3.10 shows the templates used for service composition in FLUX.

Listing 3.10: Service Composition Template

```
poss({actionName}({inputs}), Z),
state_update(Z, {actionName}({inputs}), Z2, []).
```

3.2 Verification using Fluent Calculus

With these two specifications of actions in place, we can perform a formal verification of, e.g., sequences of actions using the FLUX tool (for our purposes of composed services or sequential business processes). For example, in the simple business process shown in Figure 3.2, first an invoice is received and then paid.

This abstract definition of such a sequential business process can be specified in FLUX as shown in Listing 3.9. However, for formal verification we also have to assign an initial state with some facts to our knowledge base. This is expressed by the first statement *Z = [invoice(“ExampleInvoice”)]* in Listing 3.11. In this case, just one fact of the form *invoice(“ExampleInvoice”)* is introduced for use in the action *ReceiveInvoice*. After this initialization, the *poss* statement specifies a check whether the first action *ReceiveInvoice* can be invoked. If *poss* evaluates to true, then the *state_update* is performed. The third step specifies a check whether the action *PayInvoice* can be applied and, if yes, the *state_update* is performed. The FLUX tool interprets all these specifications based on well-defined semantics and tells that this verification succeeds.

Listing 3.11: “Receive and Pay” Business Process encoded in FLUX for Verification

```
Z = [invoice("ExampleInvoice")],

% first check if the action receiveInvoice is applicable %
poss(receiveInvoiceAction(Invoice), Z),
% then perform the action through state_update  %
state_update(Z, receiveInvoiceAction(Invoice), Z2, []),

% check if payInvoice is applicable %
poss(payInvoiceAction(Invoice), Z2),
% and then perform its update %
state_update(Z2, payInvoiceAction(Invoice), Z3, []).
```

As shown in this example, this sequence of *poss* statements along with their corresponding *state_update* statements has to be provided to the FLUX tool for specifying the verification of the sequence defined in the business process shown in Figure 3.2. If all the given statements can be performed in this order, the verification succeeds, otherwise it fails. So, this is a verification of whether a defined sequence of service invocations is consistent with the semantic specifications of the services involved.

In FLUX, service specifications are always evaluated on the current state. That is, all information that is present is also available to the services. This enables the possibility for more complex links between services. For example, once ReceiveInvoice has received an invoice, PayInvoice may use it even though other services are invoked in between, as long as any of them are not explicitly removing the invoice, more precisely the fact that it has been received, from the knowledge base.

For example, a service might have a precondition *not paid(Invoice)* and its effect or postcondition is *paid(Invoice)*. This might seem conflicting first, but can be processed by FLUX since the check if a service is applicable is separated from the update statement of the service. In other words, first FLUX checks whether the precondition can be satisfied and if so, it performs an update that introduces or removes statements to or from the current state. Of course, the human user of FLUX is responsible for the *state_update* predicate and has to take care of conflicting statements. For this example, the *state_update* predicate must not only introduce the new statement *paid(Invoice)* but also remove the statement *not paid(Invoice)* from the new state.

A verification of this simple business process corresponding to a service composition was done against the specification of the atomic services using FLUX as shown above, and it succeeded. So far, everything looks fine, and the semantic knowledge for straightforward service composition appears to be the same as for a business process [47].

3.3 Verification and Validation Mismatch

Now let us assume a larger (hypothetical) company, where an *Invoice* needs authorization before making a payment. For modeling the services, both *ReceiveInvoice* and *PayInvoice* can be reused, but an additional service needs to be specified. The additional service with its semantic specification is presented in Listing 3.12 and specifies the authorization of a given *Invoice*.

Listing 3.12: AuthorizeInvoice Specification

```
AuthorizeInvoice:
  Input:          Invoice
  Output:         Invoice
  Precondition:   received(Invoice)
  Postcondition:  authorized(Invoice)
```

The formulation in the Fluent Calculus is shown in Listing 3.13.

Listing 3.13: AuthorizeInvoice encoded in FLUX

```
AuthorizeInvoice:
  poss(authorizeInvoiceAction( Invoice), Z) :-
    knows_val([Invoice], invoice(Invoice), Z),      % Input %
    holds(received(invoice(Invoice)), Z).     % Precondition %

  state_update(Z1, authorizeInvoiceAction( Invoice), Z2, []) :-
    update(Z1, [invoice(Invoice), authorized(invoice(Invoice))], [], Z2).
    % invoice(Invoice) is the output of the action and
       authorized(invoice(Invoice)) is the postcondition %
```

The obvious business process for this larger company integrates the *AuthorizeInvoice* process between *ReceiveInvoice* and *PayInvoice* as seen in Figure 3.3.

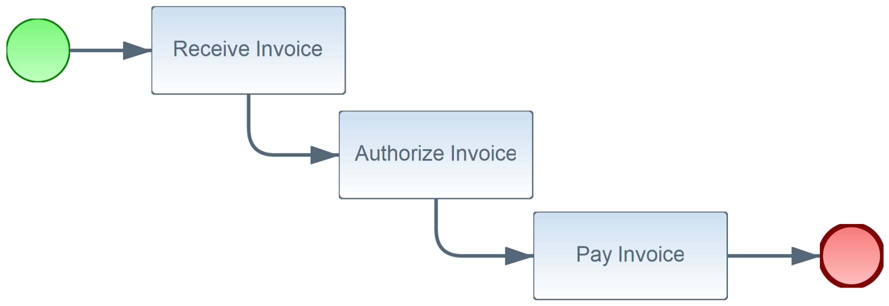


Figure 3.3: Large Company: "Receive, Authorize and Pay" Business Process

A verification of this business process corresponding to a service composition was done against the specification of the atomic services using the FLUX tool, and it succeeded.

In principle, other business processes may be erroneously defined (which is more plausible and likely for large and complex processes, of course), such as the invalid processes shown in Figures 3.4 and 3.5.

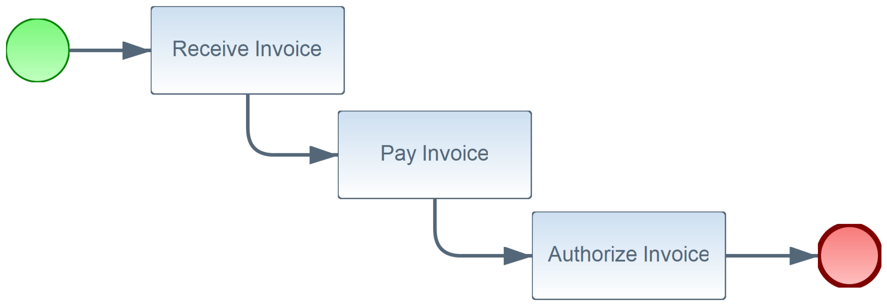


Figure 3.4: Invalid Business Process: “Receive, Pay and Authorize”

Somewhat surprisingly, also the verification of these service compositions for (invalid) business processes succeeded, although any reasonable validation would fail for them, of course. Note, that such a validation is not tool-supported and would in practice require expert business knowledge, while for our intentionally simple example it can be done using common sense. The reason for the process in Figure 3.4 is lack of explicitly represented knowledge, i.e., authorization is required before making a payment. The reason for the process in Figure 3.5 is also lack of explicitly represented knowledge, in this case it does not make sense in reality to authorize an already paid Invoice.

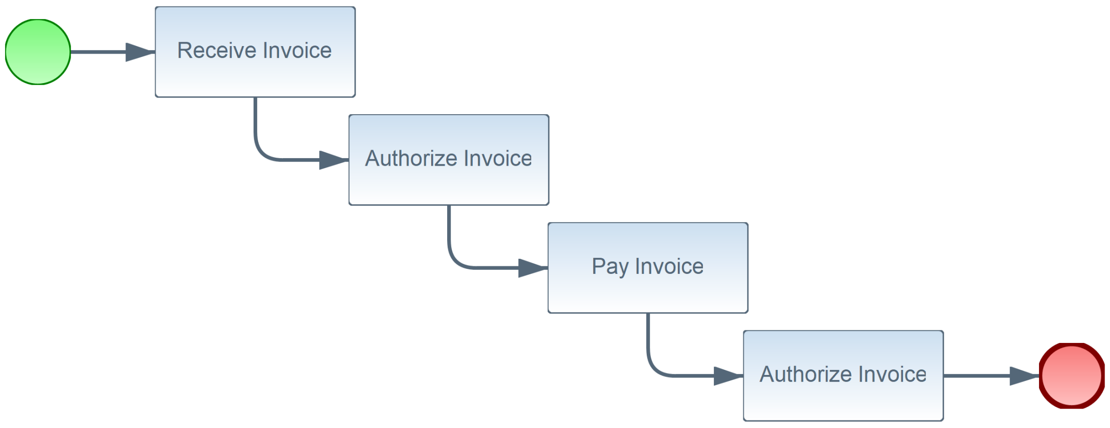


Figure 3.5: Invalid Business Process: “Receive, Authorize, Pay and Authorize”

Representing these pieces of knowledge (for the larger company) may be simply done by extending the service specifications of *AuthorizeInvoice* and *PayInvoice*. In this case a newly introduced precondition is sufficient and the specification (where the new parts are given in bold face), is presented in Listings 3.14 and 3.15.

Listing 3.14: PayInvoice Specification with explicit precondition

```
PayInvoice:
  Input:          Invoice
  Output:         none
  Precondition:   authorized(Invoice)
  Postcondition:  paid(Invoice)
```

The semantic specification of Listing 3.15 states that an invoice may only be authorized if it is not already *paid* (*not paid(Invoice)*).

Listing 3.15: AuthorizeInvoice Specification with explicit precondition

```
AuthorizeInvoice:
  Input:          Invoice
  Output:         Invoice
  Precondition:   not paid(Invoice)
  Postcondition:  authorized(Invoice)
```

These additional preconditions are represented in FLUX as well and the new specification encoded in Fluent Calculus is shown in Listings 3.16 and 3.17.

Listing 3.16: PayInvoice Specification with explicit precondition in FLUX

```
PayInvoice:
  poss(payInvoiceAction(Invoice), Z) :-
    knows_val([Invoice], invoice(Invoice), Z),      % Input %
    holds(authorized(invoice(Invoice)), Z).                     %
        Precondition %

  state_update(Z1, payInvoiceAction(Invoice), Z2, []) :-
    update(Z1, [paid(invoice(Invoice))], [], Z2).        % Postcondition %
```

For the AuthorizeInvoice service the *not paid(Invoice)* precondition of Listing 3.15 is expressed as *knows_not(paid(invoice(Invoice)), Z)* in Listing 3.17.

Listing 3.17: AuthorizeInvoice Specification with explicit precondition in FLUX

```
AuthorizeInvoice:
  poss(authorizeInvoiceAction(Invoice), Z) :-
    knows_val([Invoice], invoice(Invoice), Z),    % Input %
    knows_not(paid(invoice(Invoice)), Z).                          %
        Precondition %

  state_update(Z1, authorizeInvoiceAction(Invoice), Z2, []) :-
    update(Z1, [invoice(Invoice), authorized(invoice(Invoice))], [], Z2).
```

Strictly speaking, however, another service composition for an (invalid) business process as shown in Figure 3.6 was still verified with FLUX, but certainly not validated.

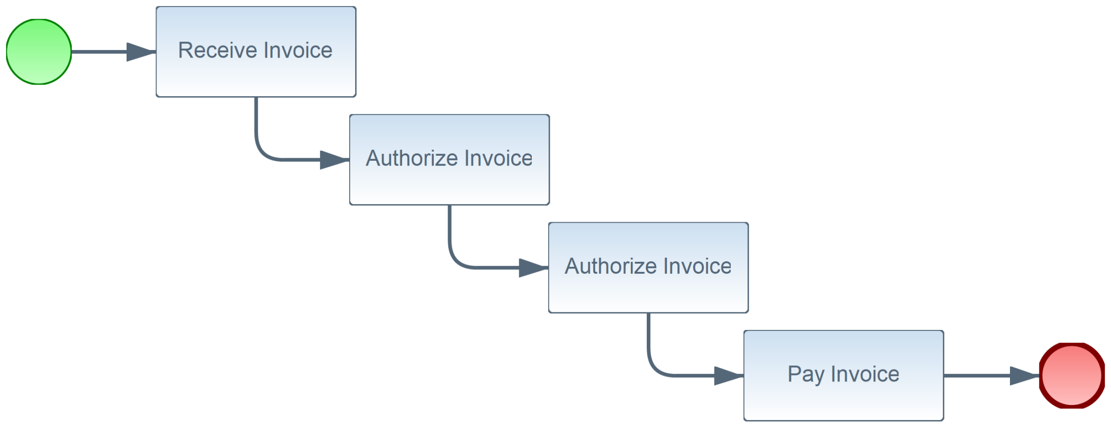


Figure 3.6: Invalid Business Process: “Receive, Authorize, Authorize and Pay”

Yet another extension of the precondition of *AuthorizeInvoice* would avoid this (see the specification in Listing 3.18).

Listing 3.18: Additional Precondition for AuthorizeInvoice

```
AuthorizeInvoice:
  Input:          Invoice
  Output:         Invoice
  Precondition:   not paid(Invoice) ∧ (not authorized(Invoice))
  Postcondition:  authorized(Invoice)
```

Note, that after all these extensions of the semantic specifications, the original business process shown in Figure 3.2 above for the small company *cannot* be verified anymore, although it is a valid process. While this problem is not inherently related to any software implementation, it is easy to understand when assuming that the basic services introduced above have implementations fitting the semantic service specifications. Even though the verification for the small company does not work with the semantic service specification of *PayInvoice* for the large company anymore, the service implementation still works, however.

Hence, there is actually a *mismatch of semantic specification and service implementation*, more precisely an *over-specification*. The additional conditions do not fit the implementation according to the original specification anymore. In fact, this additional knowledge encoded is not directly related to these services per se [47].

CHAPTER 4

Context-dependent Semantic Task Specification

Tasks of process models and their specifications, as indicated in Chapter 3, must not directly be replaced by their implementing services and their specifications. These tasks are enacted in a specific process and require customized specifications, i.e., they have their own *semantic task specification*. Hence, our approach does not involve enriching service specifications, but rather introduces semantic task specifications. The latter incorporate knowledge of the (business) context in which a task is enacted, given through business rules. This chapter explains how this context information can be expressed and how it can be used to construct semantic task specifications.

4.1 Semantic Task Specification

A semantic task specification is available only for tasks in (business) processes. They formally specify what this tasks requires and ensures in a (business) process. This is different from the specification of a service as a semantic task specification may also incorporate knowledge about the business (process). Considering our running example of the large and small companies again, we might want to specify that an invoice has to be authorized before its payment can be made. However, this might not necessarily be a precondition of the implementing service, maybe because all information is available, even without authorization, for paying an invoice, and thus the service does not specify it. In this case a separate specification, the semantic task specification, is required to formally specify this additional demand. These semantic task specifications can be used to express all additional specifications that are not part of the implementing service specification. The additional specifications are essentially and-connected pre- and postconditions. Hence, their pre- and postconditions are propositional logic formulas.

For example the adapted preconditions of Listing 3.14 could be considered part of a semantic task specification.

4.2 Making Services Reusable Through Semantic Task Specifications

Using semantic task specifications facilitates the reuse of services across different business processes in a formally justified manner (not just running their procedural implementations). In fact, each service can be used by multiple tasks.

For such reuse, however, a task specification and the specification of its implementing service need to be in a *subtyping* relationship where the service specification is a subtype of the task specification. A subtyping relationship imposes constraints on specifications [112]. The precondition of the supertype implies the precondition of the subtype, i.e., these conditions are either the same or the latter is weaker than the former. In contrast, the postcondition of the subtype implies the postcondition of the supertype, i.e., these conditions are either the same or the latter is weaker than the former. Therefore, in our approach the precondition of the task specification has to imply the precondition of the specification of the service that implements this task, and vice versa for the postconditions. Figure 4.1 visualizes the subtyping relationship between service and task specifications.

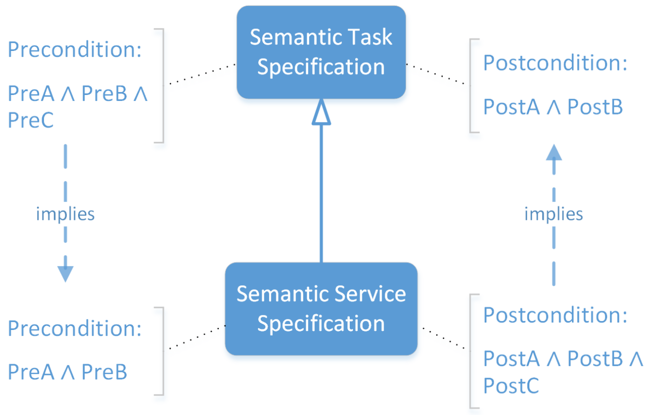


Figure 4.1: Subtyping relationship between service and task specifications

Through this approach we are able to decouple a service specification from the business context and, thus, to make the service reusable in different business contexts. Figure 4.2 illustrates such a reuse of *Pay Invoice* in more than one business process.

These two processes have different contexts. In contrast to the small company, the process of the large company requires authorization before payment of an invoices. This additional information is modeled as an extra precondition for the corresponding *Pay Invoice* task of the large company. Listing 4.1 shows the semantic task specification of *Pay Invoice* for the large company.

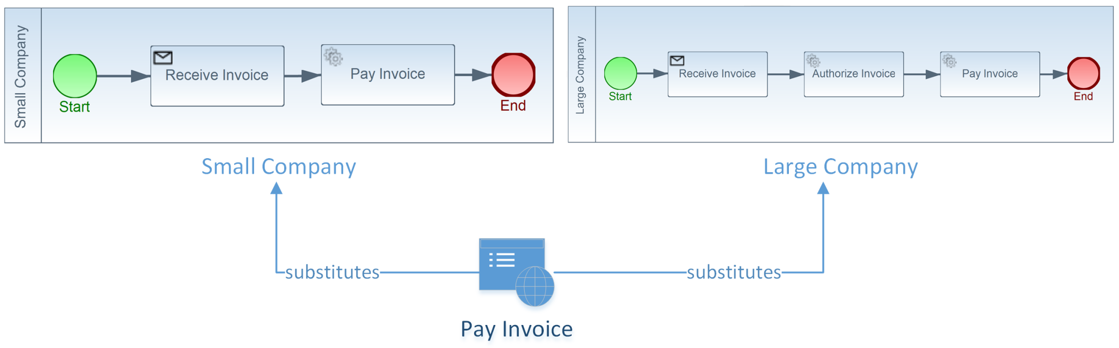


Figure 4.2: Making Services and their Specifications Reusable

Listing 4.1: Pay Invoice Semantic Task Specification

```
Task:  Pay Invoice
  Pre:  received(invoice) ∧ authorized(invoice)
  Post: paid(invoice)
```

In effect, there are two *Pay Invoice* tasks in different business processes, with different semantic specifications, but they use the same service implementation. As shown in Figure 4.4, the *Pay Invoice* task in the context of a large company has an additional precondition, *authorized(Invoice)*, which specifies that the task can only be executed if an authorized invoice is available. The subtype relationship between this service and these tasks guarantees *substitutability* for facilitating reuse.

4.3 Specifying Context Information as Conditional Rules

Until now we have constructed all our semantic task specifications manually, i.e., we did not have a formal specification of the context in which a process is enacted. Rules are one option that is commonly used to specify conditions that a process has to adhere to. Actually, this is business knowledge in addition to these services, more precisely these are a kind of business rules. So, we propose to make such knowledge explicit in an additional specification separate from the service and task specification. Essentially, (business) rules can be considered as formulas in propositional logic.

Let us consider the conditional rule shown in (Business) Rule 4.1.

(Business) Rule 4.1. *Authorization Required before making a Payment*
"An invoice has to be authorized to make its payment."

This rule simply states that a task that makes a payment of an invoice can only be executed if the invoice has been authorized. Essentially, this rule has a condition, invoice has to be authorized, and a (possible) consequence, invoice can be paid. A semi-formal representation is given in Listing 4.2.

Listing 4.2: Business rule stating that an invoice has to be authorized to be paid

```
Rule Authorize-Before-Payment-Invoice:
  condition:  paid(invoice)
  consequence: authorized(invoice)
```

This rule can be formally stated as an implication relationship between paid(invoice) and authorize(invoice). If an invoice is paid, *paid(invoice)*, then it has to be authorized, *authorized(invoice)*, as well.

$$paid(Invoice) \rightarrow authorized(Invoice) \tag{4.1}$$

As a more complex example, let us consider (Business) Rule 1.1 of our running example. This rule states that an invoice with an amount greater or equal than a threshold has to be authorized before its payment can be made. Again, we can formalize this rule as an implication stating that if an invoice is paid and its amount is greater than a threshold, it has to be authorized as well.

$$(paid(Invoice) \wedge (Invoice.amount \geq threshold)) \rightarrow authorized(Invoice) \tag{4.2}$$

However, it would be more convenient to express this (business) rules as a condition on the payment as this is what we want to restrict. Actually, the rule given above can be rewritten as:

$$paid(Invoice) \rightarrow ((Invoice.amount \geq threshold) \rightarrow authorized(Invoice)) \tag{4.3}$$

This formula is equivalent to the previous one and expresses that if an invoice is paid then, if its amount is greater or equal than a threshold, has also to be authorized.

Now, let us consider an extension of the (Business) Rule 1.1 in Section 1.3 since it involves a piece of *tacit knowledge*. This extension states that an invoice *must not* be authorized if its amount is less than a threshold. The rule is shown in (Business) Rule 4.2.

(Business) Rule 4.2. *Authorize Only if Amount Greater than a Threshold*
"If the amount of an invoice is greater or equal than a threshold level, its payment has to be authorized. If the amount of an invoice is less than a threshold, it must not be authorized before its payment"

The extension can be formalized similarly to the rule above as:

$$(paid(Invoice) \wedge \neg(Invoice.amount \geq threshold)) \rightarrow \neg authorized(Invoice) \tag{4.4}$$

Again, this can be converted to an implication on paid(Invoice):

$$paid(Invoice) \rightarrow (\neg(Invoice.amount \geq threshold) \rightarrow \neg authorized(Invoice)) \tag{4.5}$$

Since both parts of the (Business) Rule 4.2 `Authorize Only if Amount Greater than a Threshold` condition paid(invoice), we can combine them using a logic and-connector.

$$\begin{aligned} &paid(Invoice) \rightarrow \\ &\qquad (((Invoice.amount \geq threshold) \rightarrow authorized(Invoice)) \wedge \\ &\qquad\qquad (\neg(Invoice.amount \geq threshold) \rightarrow \neg authorized(Invoice))) \end{aligned} \tag{4.6}$$

Throughout this thesis, we formalize all context information with (business) rules using an implication.

4.4 Using Semantic Task Specifications and Conditional Rules in FLUX

Since the Fluent Calculus essentially works with actions, we model such a (business) rule as an action that sets a specific state, which is its postcondition, according to its input and precondition. This does not entail that some specific business actor would have to perform such an action. It just models the missing business knowledge in such a way that it fits the given formalism required for automated verification.

Listing 4.3 shows an example of such a representation of a (business) rule based on the examples introduced in Chapter 3. It sets for a specific invoice (its input) that it is ready for payment. The rule is only applied if an invoice is present and has already been authorized. Comparing this listing with the service specification in Listing 3.14 shows that the additional precondition has been moved to an explicit specification of our (business) rule.

Listing 4.3: Business rule that determines if an Invoice is ready for payment in FLUX

```
poss(ruleIsReadyForPayment(Invoice), Z) :-
    knows_val([Invoice], invoice(Invoice), Z),    % Input %
    holds(authorized(invoice(Invoice)), Z).

  state_update(Z1, ruleIsReadyForPayment(Invoice), Z2, []) :-
    update(Z1, [ruleIsReadyForPayment(invoice(Invoice))], [], Z2).
```

However, this explicit specification of the (business) rule requires some minor changes to the service specifications. The idea is that every service where an payment of an invoice is made, has an additional precondition that checks if a specific invoice is, in fact, ready for payment. Only if this condition is fulfilled, the action can be invoked. In our example, this condition is directly related to the postcondition of the (business) rule defined in Listing 4.3. Listing 4.4 shows the additional precondition *isReadyForPayment* encoded in FLUX. The results of the verification stays the same. That is why the *state_update* part has been omitted. This approach can be applied to all service specifications at once, while the service implementations do not have to be changed.

Listing 4.4: PayInvoice plus precondition for (business) rule encoded in FLUX

```
poss(payInvoiceAction(Invoice), Z) :-
    knows_val([Invoice], invoice(Invoice), Z),      % Input %
    holds(isReadyForPayment(invoice(Invoice)), Z).  % from Business Rule %
```

However, there is a problem with the specifications of the previous listings. Process 4.3 also verifies with this specification. The problem is that the precondition of the *payInvoiceAction* still holds after its invocation and thus it can be invoked again. To solve this problem, the fact has to be explicitly removed from the knowledge base that the invoice is ready for its payment. The resulting specification is shown in Listing 4.5, and with this specification Process 4.3 does not verify anymore.

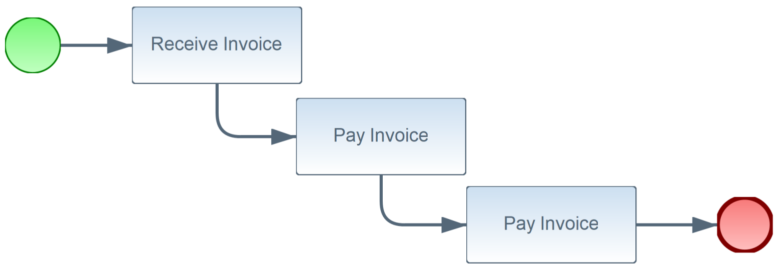


Figure 4.3: Invalid Business Process: “Receive, Pay and Pay”

Listing 4.5: Amended PayInvoice Specification encoded in FLUX

```
  state_update(Z1, payInvoiceAction(Invoice), Z2, []) :-
    update(Z1, [paid(Invoice)], [isReadyForPayment(invoice(Invoice))], Z2).
```

The same approach is also applied to the authorization action defined in Listing 3.18. Here we can introduce a new (business) rule that sets for a specific invoice if it is ready for authorization. The specification of the action must then include this fact as a precondition.

Listing 4.6 shows this additional (business) rule. In this case, the (business) rule is quite simple and just specifies that all available invoices are automatically ready for authorization. Similarly to the *payInvoice* action, also *authorizeInvoice* has to be changed and a new predicate, according to this (business) rule, has to be introduced.

Listing 4.6: Business rule that determines if an Invoice is ready for authorization

```
poss(ruleIsReadyForAuthorization(Invoice), Z) :-
  knows_val([Invoice], invoice(Invoice), Z).

state_update(Z1, ruleIsReadyForAuthorization(Invoice), Z2,[]) :-
  update(Z1, [isReadyForAuthorization(invoice(Invoice))], [], Z2).
```

Listing 4.7 shows the adjusted service specification. The additional precondition has been introduced and also the result of the action has been adjusted, so that the fact that the invoice is ready for authorization is removed after the invocation of the action.

Listing 4.7: Adjusted AuthorizeInvoice action

```
poss(authorizeInvoiceAction( Invoice), Z) :-
  knows_val([Invoice], invoice(Invoice), Z),
  holds( isReadyForAuthorization(invoice(Invoice)), Z).

state_update(Z1, authorizeInvoiceAction( Invoice), Z2, []) :-
  update(Z1,
    [authorized(invoice(Invoice))],
    [isReadyForAuthorization(invoice(Invoice))],
    Z2).
```

With these explicitly specified (business) rules, only the valid process in Figure 3.3 can be verified, while the verification correctly fails for all others. However, there is still an issue with our small company that does not have an authorization action in its (business) process. So, Figure 3.2 still cannot be verified. Since there is no authorization action, the (business) rule has to be adjusted. Similarly to the (business) rule in Listing 4.6, we can specify a (business) rule that states that all invoices are automatically ready for the pay action. This rule is shown in Listing 4.8.

Listing 4.8: Business rule that determines if an Invoice is ready for payment in a small company

```
poss(ruleIsReadyForPayment(Invoice), Z) :-
  knows_val([Invoice], invoice(Invoice), Z).

state_update(Z1, ruleIsReadyForPayment(Invoice), Z2,[]) :-
  update(Z1, [isReadyForPayment(invoice(Invoice))], [], Z2).
```

In effect, there is no need to change the service specification anymore. The specifications shown in Listings 4.4 and 4.5 can be directly reused [47].

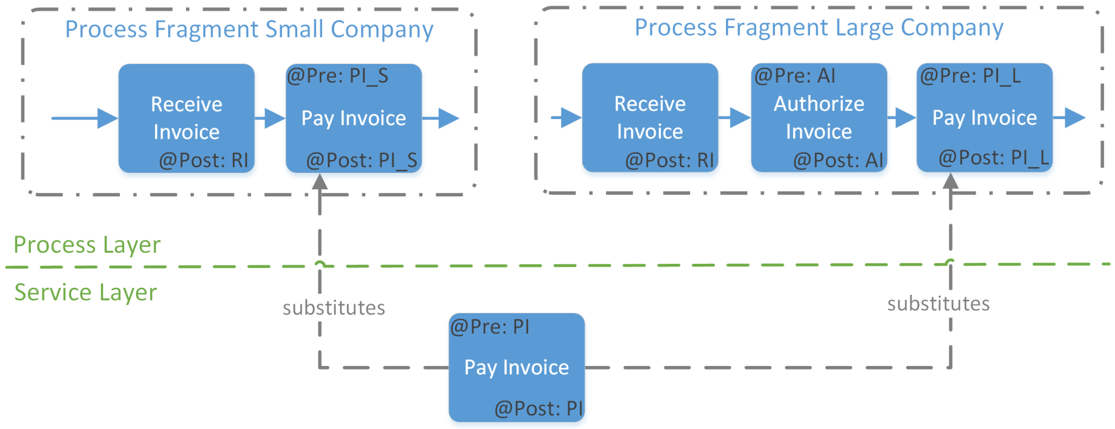
4.5 Context-dependent Semantic Task Specifications

As stated above, the context of the business process is important for a specification of tasks. Originally, without the information of the context, the task specification is the same as the one of the service. For example, initially the *Pay Invoice* task has the specification given in Listing 4.9.

Listing 4.9: Original *Pay Invoice* task specification

```
Task: Pay Invoice
      Pre:  received(Invoice)
      Post: paid(Invoice)
```

In Figure 4.4, this rule is taken into account by an additional precondition on the *Pay Invoice* task. The question is, how can we systematically enrich the task specification from an existing business rule?



Post: RI = received(Invoice); Pre: AI = received(Invoice); Post: AI = authorized(Invoice); Pre: PI_L = received(Invoice) ∧ authorized(Invoice); Post: PI_L = paid(Invoice); Pre: PI_S = received(Invoice); Post: PI_S = paid(Invoice); Pre: PI = received(Invoice); Post: PI = paid(Invoice);

Figure 4.4: Context-dependent Semantic Task Specification

This depends, of course, on the formalization of the (business) rule. For providing a systematic approach, we have to look at (business) rules in more detail. The (Business) Rule 4.1 specifies that "An invoice is only allowed to be paid, if it has been authorized before". Intuitively, this leads to the additional precondition *authorized(Invoice)* of the corresponding *Pay Invoice* task specification, but a systematic approach is preferable.

With a business rule formalization of *condition* → *consequence*, it is possible to systematically extract additional preconditions on tasks. To accomplish this, each postcondition of all tasks has to be examined if a matching business rule is available. A business rule matches if the consequence of the rule implies the postcondition of a task.

Considering our example, the condition *paid(Invoice)* of the (Business) Rule 4.1 `Authorization Required before making a Payment` in Listing 4.2 matches the postcondition of the *Pay Invoice* task from Listing 4.9. Hence, we can derive the additional precondition *authorized(Invoice)* for the task. The resulting precondition is *received(Invoice)* ∧ *authorized(Invoice)*. The precondition of this task specification is the same as *@Pre: TI_L* in Figure 4.4 and, in fact, is equivalent to the one in Listing 4.1, which we constructed manually without the use of business rules.

Essentially, (business) rules add additional preconditions to a semantic task specification. These additional preconditions are and-connected with the already specified preconditions of the semantic task specification.

Considering our (Business) Rule 1.1, we can systematically enrich the “Pay Invoice” task since it has a matching postcondition paid(invoice). The precondition to be added to the already specified precondition received(invoice) is shown in Equation 4.3 and is $(Invoice.amount \geq threshold) \rightarrow authorized(Invoice)$. The resulting precondition is shown in Listing 4.10.

Listing 4.10: Enriched *Pay Invoice* task specification

```
Task: Pay Invoice
      Pre:  received(Invoice) ∧ ((Invoice.amount ≥ threshold) → authorized(
          Invoice))
      Post: paid(Invoice)
```

Actually, the precondition can also be rewritten as:

$$received(Invoice) \wedge (authorized(Invoice) \vee \neg(Invoice.amount \geq threshold)) \tag{4.7}$$

We can use the same approach to automatically enrich the “Pay Invoice” task with our complex (Business) Rule 4.2. The enriched semantic task specification is shown in Listing 4.11.

Listing 4.11: Enriched *Pay Invoice* task specification for Authorize only if amount greater

```
Task: Pay Invoice
      Pre:  received(Invoice) ∧
          (((Invoice.amount ≥ threshold) → authorized(Invoice)) ∧
          (¬(Invoice.amount ≥ threshold) → ¬authorized(Invoice)))
      Post: paid(Invoice)
```

The precondition can be simplified to:

$$\begin{aligned} &\mathit{received}(\mathit{Invoice}) \wedge \\ &\qquad ((\mathit{authorized}(\mathit{Invoice}) \wedge (\mathit{Invoice.amount} \geq \mathit{threshold})) \vee \\ &\qquad\qquad (\neg \mathit{authorized}(\mathit{Invoice}) \wedge \neg(\mathit{Invoice.amount} \geq \mathit{threshold}))) \end{aligned} \tag{4.8}$$

Our approach uses the following steps to enrich preconditions:

1. Initially copy semantic service specification to the task, e.g., precondition *Pre1* and postcondition *Post1*.
2. Look for business rule with matching postcondition *Post1*.
3. Add the preconditions of the Rule to the preconditions of the task, resulting in a combination of both, e.g., $\mathit{PreRule1} \wedge \mathit{Pre1}$.

With this approach, preconditions can be systematically extracted from formalized (business) rules and attached to the corresponding tasks [50].

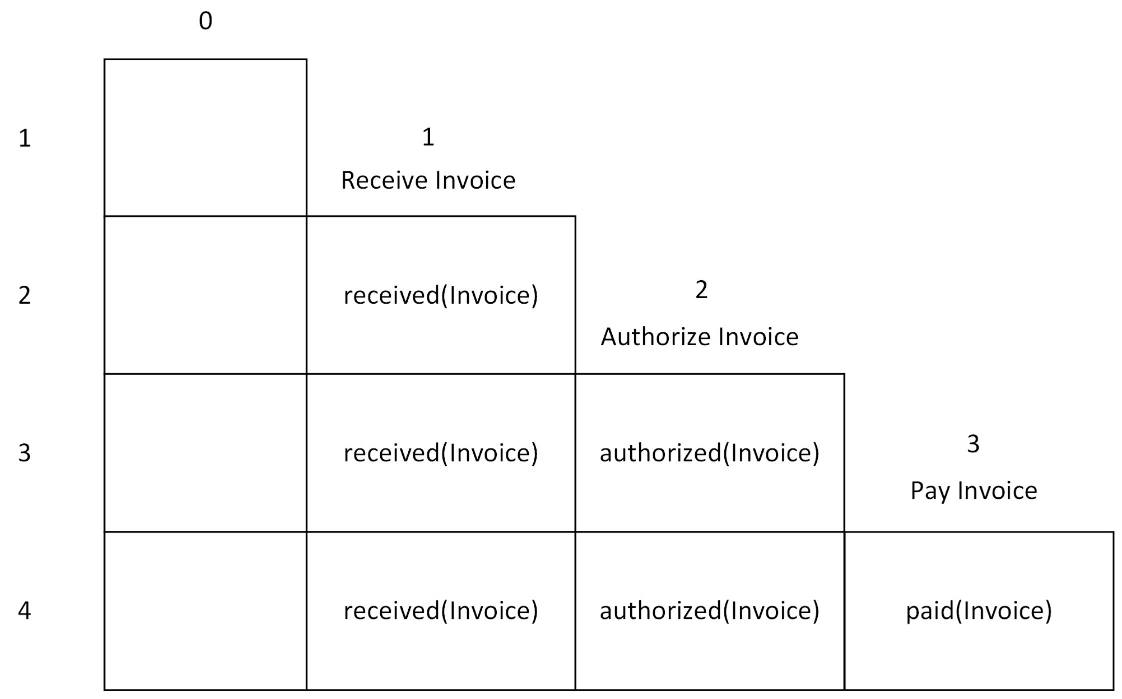
4.6 Semantic Specifications of Composite Tasks

Figure 4.4 shows (sequential) compositions of tasks. Such a composition as a whole has, in general, certain conditions for its execution and an overall effect. These are its precondition that has to be fulfilled so that the task composition as a whole can be executed, and its postcondition specifying the overall effect of an execution of the task composition. Hence, the precondition necessary for and the effect resulting from such a sequence (including business rules) can be taken as the semantic task specification of the composite task defined by this sequence. Still, the question remains how such pre- and postconditions can be systematically extracted from the task composition.

In fact, this kind of problem has been addressed already long time ago in the context of planning in Artificial Intelligence. Both for plan execution and for extracting macro-operators, a *Triangle Table* was devised [76, 35, 34]. Such a table shows explicitly which conditions have to be fulfilled for the execution of each action/operator (in our case each task) of the sequence (to its left), and which conditions result from it (below).

Let us use the composed task sequence for the large company as an example [50]. First an invoice is received, then it is authorized and only after that it is paid. Figure 4.5 presents the corresponding Triangle Table. It shows, for instance, the precondition of the task *Authorize Invoice*, received(Invoice), and its postcondition, authorized(Invoice). Both these conditions together (more precisely their conjunction) make up the precondition of the task *Pay Invoice*, given to its left.

Overall, the precondition of the whole sequence is the conjunction of all the conditions listed in the first column, and its postcondition is the conjunction of all conditions listed



	0			
1		1 Receive Invoice		
2		received(Invoice)	2 Authorize Invoice	
3		received(Invoice)	authorized(Invoice)	3 Pay Invoice
4		received(Invoice)	authorized(Invoice)	paid(Invoice)

Figure 4.5: Triangle Table of a Composite Task

in the last row. For our example, the resulting pre- and postconditions of the task composition are given in Listing 4.12.

Listing 4.12: Resulting pre- and postconditions derived from the Triangle Table

```
Composite Task: Large Company
  Pre: none
  Post: received(Invoice) ∧
        authorized(Invoice) ∧
        paid(Invoice)
```

We view this task composition as a new task consisting of several tasks. In the context of a business process, this task can be seen as a sub-process in which several tasks are executed. The semantic specification of this *composite task* is the same as the task composition specification.

In our example, the composite task has actually no precondition, since the first task does not have any precondition and the subsequent tasks all have preconditions fulfilled by predecessor tasks. However, this is not necessarily the case for all possible task compositions. If the *Receive Invoice* task, for example, had a precondition that an amount has to be set, then the task composition and, thus, the composite task would also have this precondition. In addition, subsequent tasks do not necessarily have only preconditions that are fulfilled by previous tasks. For example, if the *Pay Invoice* task additionally required an address for the invoice, then this would result in an extra precondition. This extra precondition is not fulfilled by previous tasks and, thus, would become an additional precondition of the composite task.

4.7 Recursive Application

Such a composite task, which specifies a sub-process, can be itself a building block for a higher-level business process. That is, each *composite task* can be used as a *single task* within another business process. In case such a composite task is used in another process, the corresponding task composition of the composite task is executed. This calls for a *recursive* application of our approach as described above.

A composite task can be used in the same way as shown for service tasks above. Its semantic specification as derived above is context-independent with regard to any higher-level business process it might be used in. This context-independent specification corresponds to a context-independent service specification as used above. That is, a higher-level process can use a composite task in the same way a business process uses a service. The higher-level business process may include an enriched semantic task specification based on its context, which again relates via subtyping to the context-independent composite task specification. Figure 4.6 illustrates the reuse of a composite task.

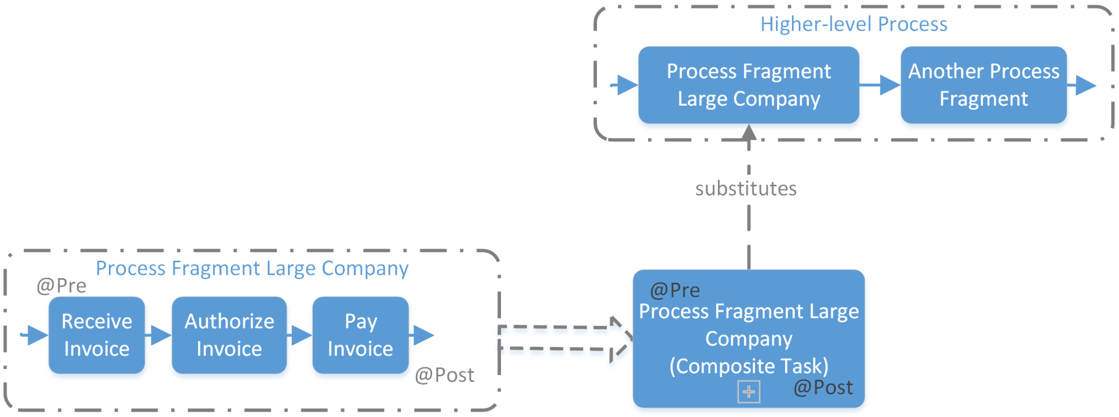


Figure 4.6: Recursive Application of Composite Tasks

Figure 4.7 illustrates how tasks can be embedded in higher-level business processes. It shows that the specifications of tasks are enriched depending on the context they are used in. This is shown in the upper part of the figure above the green broken line, labeled *In specific business context*. In the middle part of the picture, labeled *Without specific business context*, the corresponding context-independent specifications for the services and composite tasks are shown. It is important to note that the "Process Fragment Large Company" on the left side is a task composition, which is enacted in a specific context. The context is in this case that the large company has a business rule in place, that each

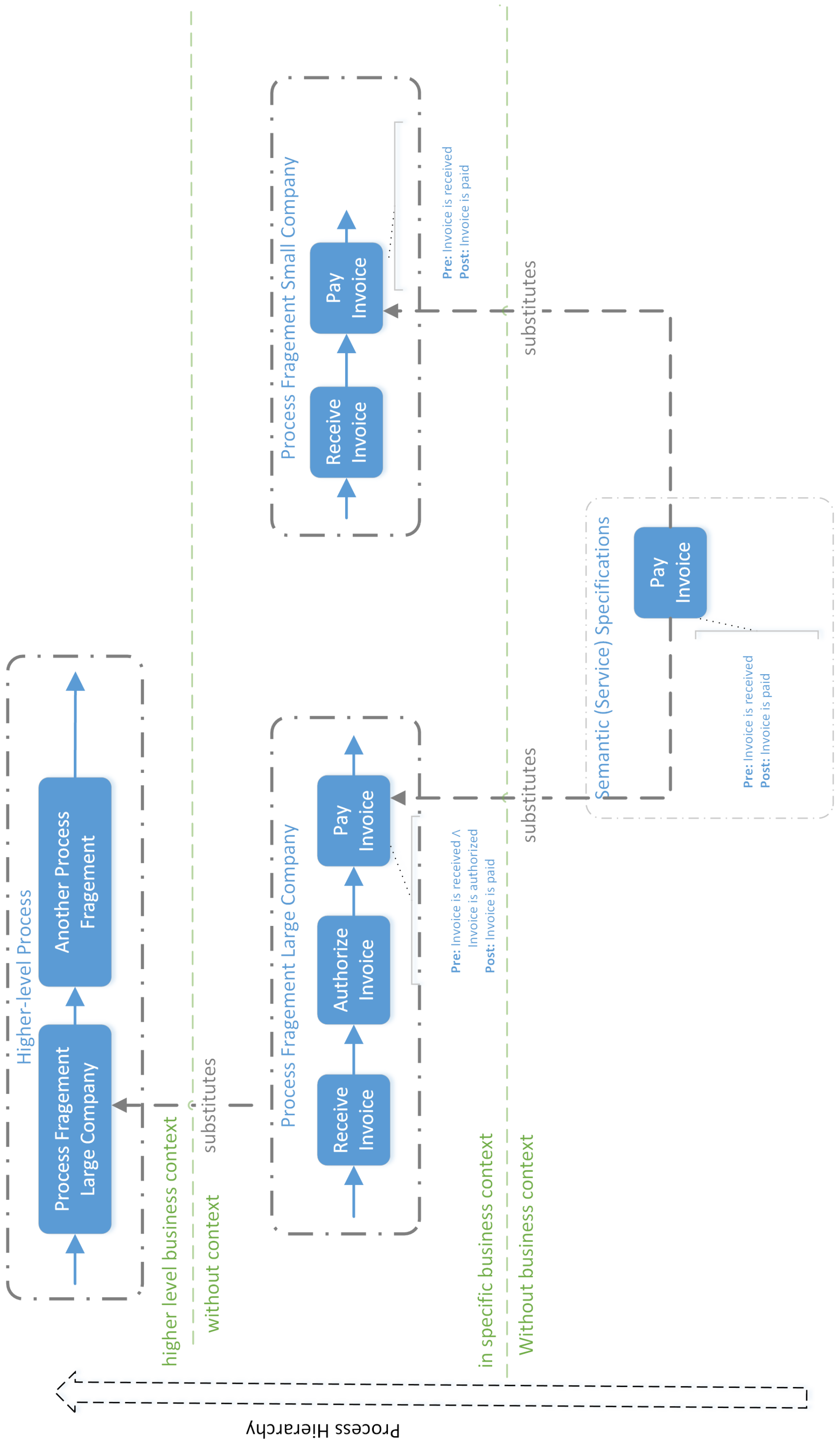


Figure 4.7: Recursive Application of Semantic Task Specifications

invoice has to be authorized before being paid. On this level, the task specification is context dependent. However, this task composition forms a new composite task shown on the right, whose semantic specification may have been systematically derived using a Triangle Table. This specification is context independent. Yet a higher-level process has its own context, which additionally can restrict the composite task via business rules. Handling this works in the same way as described above.

Hence, using the approach above recursively, we are also able to specify new tasks and use them in higher-level business processes [50].

CHAPTER 5

Using Object Life Cycles for Semantic Task Specifications

This chapter explains how we can use semantic task specifications in combination with artefact-centric models, as we formally ground them in life cycles of objects. We actually extend object life cycles to include attributes as well and ground pre- and postconditions on values of these attributes. Using the semantic task specifications in this way enables us to formalize them without knowledge of the process they are enacted in.

5.1 Specifying Semantic Task Specifications without Process Knowledge

Using *object life cycles* allows us to decouple the semantic task specifications from the process models as they only relate to artefacts and their life cycles. This means, that the person who specifies the tasks, does not not need to know in which process model they are used. If we consider the conditional rules of Section 4.3, then the advantage of decoupling both models becomes apparent. Since the conditional rules provide information about the business context and should be applicable to many process models, the tasks that are effected by it should not rely on information from the process model itself.

For example, if we consider our conditional (business) rule from 1.1 again, we could also imagine it to be specified as "Before executing pay invoice the task authorize invoice has to be executed". Although, both rules seem to express the same knowledge, they are fundamentally different. While the first rule states conditions on a business artefacts, which are not process specific and valid throughout the business, the second rule only limits the execution of a specific process model, i.e., the one with an Authorize Invoice and Pay Invoice task. Moreover, the second rule requires knowledge about concrete tasks in process models, which may change over time or could be re-factored to do something else. Hence, the conditional rule only applies to a limited number of process models and may not fit anymore after updating it. In contrast, business artefacts are typically available throughout the business and persist also after re-factoring parts of them.

In [48] we showed how object life cycles can be used to specify properties that the BPMs are verified against. Thus it is necessary to have a formal specification of these object life cycles. The concept of an object life cycle is independent from the domain that it is used in, and the concepts from the domain need to provide their own instantiation of an object life cycle. That is, each domain concept has its own definition of its life cycle by creating instances of the object life cycle concepts.

For this purpose, we introduce an ontology to model life cycles in OWL. This ontology defines that an object life cycle consists of several *States* and *Transitions* among them. The *Transitions* are defined with the *nextState* relationship. Furthermore, the *nextState* relationship also specifies which sequences of *States* are allowed in an object life cycle. For each domain object, a concrete new instance of the object life cycle ontology is created. The definition of the domain can be provided in OWL as well or be part of an Enterprise Architecture [87].

Figure 5.1 shows how this life cycle ontology is used for an ontology of domain objects and their concrete life cycles. More precisely, the figure shows how the domain object *Invoice* can be enriched with an object life cycle. The concept *Invoice* has a relationship to the *State* concept, which defines that each instance of an *Invoice* is in exactly one state in the object life cycle at any given time. The object life cycle of *Invoice* is modeled through instances of the *State* concept and its *Transitions.* In this example, an *Invoice* is *received* first and then either *authorized* or *paid.* The example also illustrates how more than one successor state can be defined [87].

When considering our running example of Section 1.3 again, then we realize that all tasks are performed on objects. In this case the object is an *Invoice.* This Invoice cannot be altered arbitrarily but only in a specific way that underlying conditions are not violated. These conditions are specified by an object life cycle. In fact, there are two object life cycles of Invoices in place for both participants of the process, the delivering company and the customer.

Figure 5.2 shows the object life cycle of the invoice of the delivering company as an FSM. It states that an Invoice first is created, then transmitted, then a payment may be received and finally it is booked. This object life cycle does not involve any variations in its path and all states are visited one after another.

Figure 5.3 shows the object life cycle of the invoice of the customer company. Here an Invoice is first received (from the delivering company) and then paid. However, there is a variation possible and an Invoice of this life cycle may also be authorized before its payment.

5.2 Extending Object Life Cycles with Attributes

We extend our object life cycles to include attributes as well. The rationale is that the FSM that represents an object life cycle as usual is in a particular state at any given point in time, but it does not have any memory of the previous states it came through.

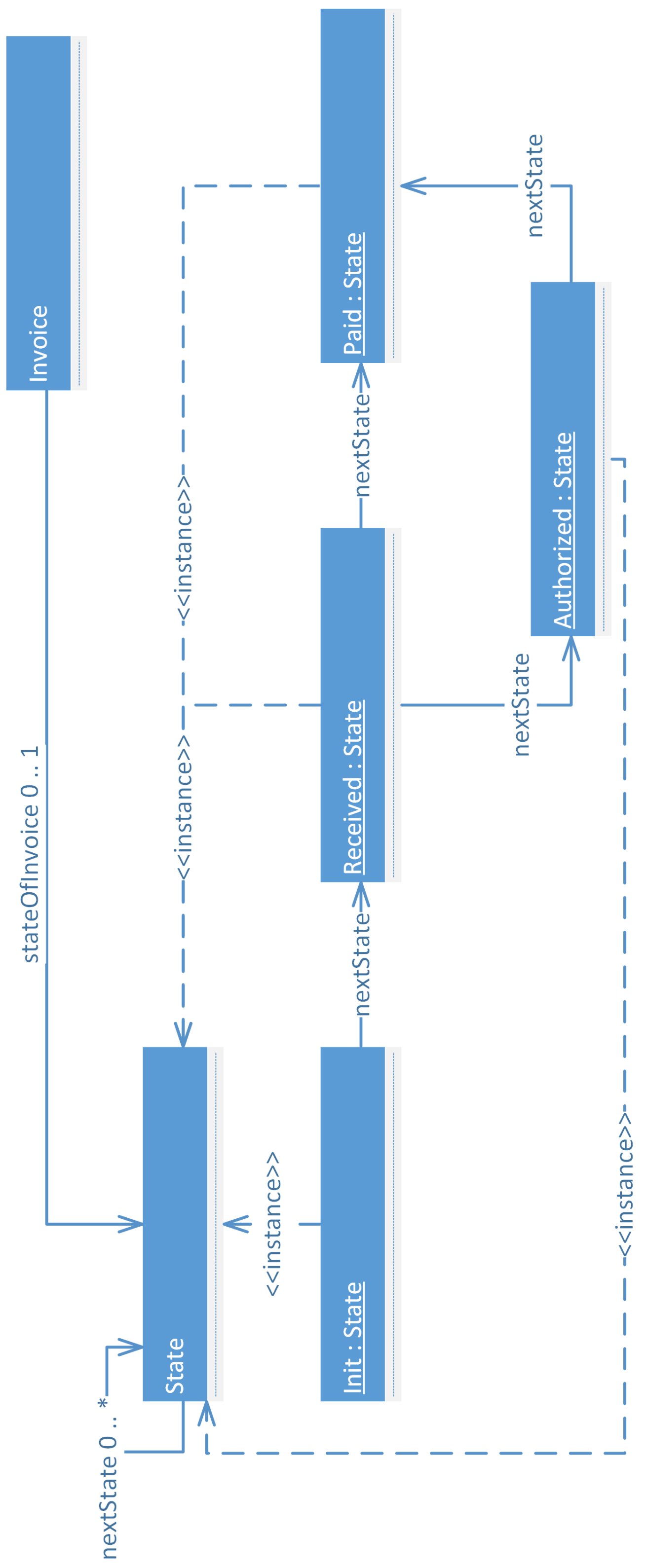


Figure 5.1: Part of an Ontology of Domain Objects and Their Life Cycles

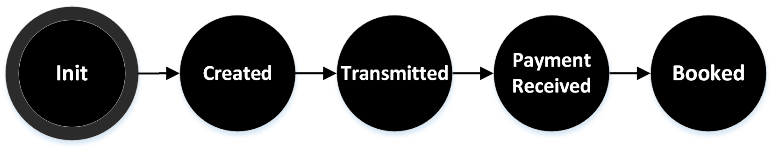


Figure 5.2: Object Life Cycle of the Delivering Company

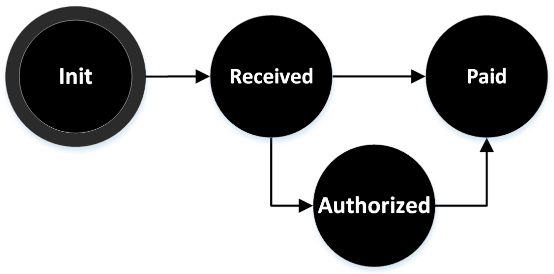


Figure 5.3: Object Life Cycle of the Customer Company

For instance, when the FSM of the traditional *Invoice* object life cycle in Figure 2.7 is in state Paid, it has no memory of whether it was in state Authorized before or not. Hence, we add an attribute to each state for memorizing that an object was in this particular state.

More precisely, each attribute of an extended object life cycle is represented by an FSM with two states added to the object life cycle. The first state represents that the value of the attribute is false, and it is true in the last state. In effect, the current state of this additional FSM of a certain attribute assigned to a state of the basic FSM of the object life cycle tells whether the latter state has been visited or not.

Figure 5.4 shows an example of an attribute as an FSM. It represents the state that an Invoice has been created or not. The attribute has only been set if the FSM has been in the state *Created True*.

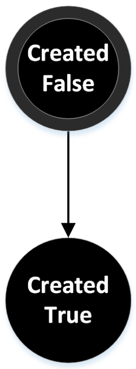


Figure 5.4: Created Object Life Cycle Attribute

There is no transition back to the state *Created False* on purpose. This is to prevent processes and object life cycles to remove information from the knowledge base. Thus, if an invoice has been created, this shall remain true. If all properties are modeled using this approach, a *monotonic* system is specified. For the purpose of verification we will only handle monotonic systems. However, it has to be noted that the FSMs of the attributes could include, in principle, transitions to previous states as well.

These attributes are extensions of the object life cycles of Section 5.1. In fact, the attributes may only be set by following along the transitions of the object life cycle. Hence, if a process moves the object life cycle of an artefact from the state Created to Transmitted, then setting of the attribute Transmitted is triggered, i.e., the attribute "reacts" to the state of the object life cycle.

Figure 5.5 illustrates how object life cycles and attributes are connected to each other. The FSMs of the attributes have a condition on their transition, which only triggers if the object life cycle reaches a specific state. For example, the attribute *Transmitted* is only set to true if the object life cycle reaches the state *Set Transmitted*, as expressed by the condition $State(InvoiceDCP) = SetTransmitted$.

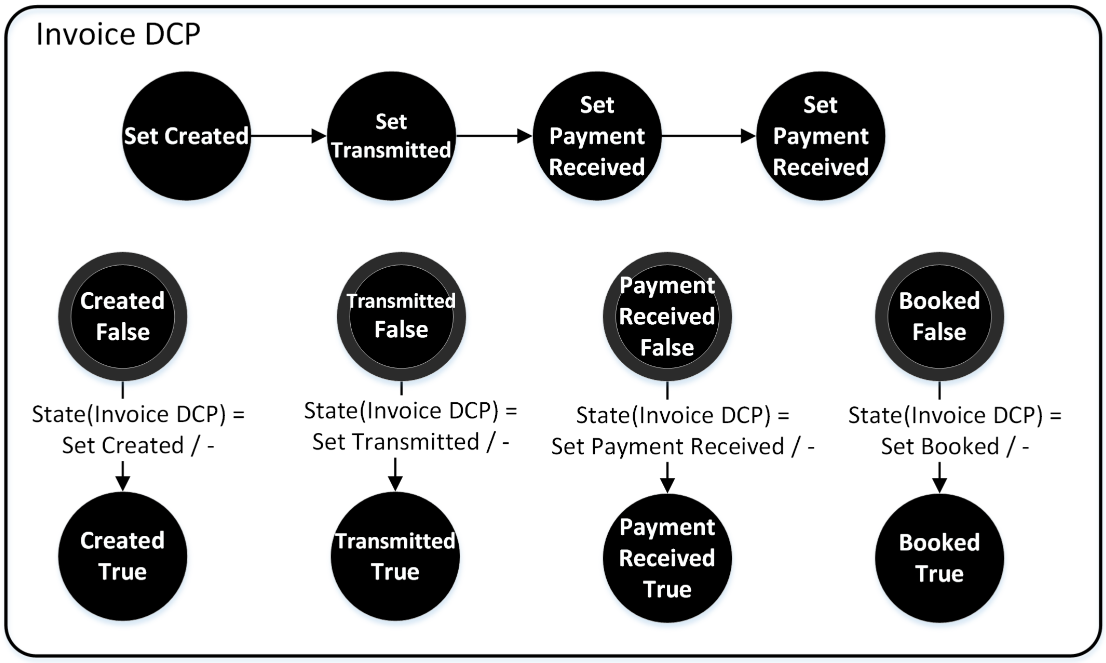


Figure 5.5: Object Life Cycle with its Attributes

In these extended object life cycles the overall state of a specific artefact is given by the current assignment of all its properties and its position in the object life cycle.

5.3 Grounding of Semantic Specifications in Extended Object Life Cycles

In order to define the real meaning of the semantic task specifications in the process models where the tasks are used, we specify their *grounding* in object life cycles, more precisely on our extended version of object life cycles.

Our approach utilizes object life cycles to formally verify process models against. Hence, formal representation of an object life cycle is a necessity. We use FSMs to directly specify object life cycles and relate the process models via semantic task specifications, given as *predicates grounded* in the *object life cycle*, to them.

Each semantic task specification may contain several statements related to the object life cycle and each statement is expressed as a predicate *attributeSet*. This predicate operates on a *subject* and an *object*. In essence, the predicate is in the form of *attributeSet([Subject],[Object])*. In our case, the subject is related to the artefact and the object to a specific attribute of the formal representation of the object life cycle. Thus, the predicate can be interpreted as *attributeSet([Artefact],[Attribute])* and denotes that for *[Artefact]* the *attribute [Attribute]* is *set*.

Let us consider our example again. In Listing 5.1, the semantic task specification of the *Transmit Invoice* task is specified. It contains one precondition, an invoice had to be created, and one postcondition, an invoice is transmitted. In this case, the "invoice" relates to the artefact and "created" or "transmitted" to a specific state of the FSM of the object life cycle. In this case, the attributeSet predicate is expressed as follows: for the precondition *attributeSet(invoice, created)* and for the postcondition *attributeSet(invoice, transmitted)*. By doing so, we formally ground our predicate on the attributes of the artefact. In this way, we also connect the *declarative* and *procedural* specification formally.

Listing 5.1: TransmitInvoice Task

```
TransmitInvoice:
   Pre: created(Invoice)
   Eff: transmitted(Invoice)
```

Additionally, the semantic task specification also specifies an action that is performed. This action triggers a transition on the object life cycle. Formally we define this action as a predicate *triggerTransition([Subject], [Object])*, which is specified on a subject, i.e., the artefact, and an object, i.e., the state to be reached. For example, the *Book Invoice* task triggers the transition into *Set Booked* of the object life cycle.

CHAPTER 6

An Approach to Formal Verification

In the previous chapters, we described how semantic task specifications can be formulated using business artefacts with life cycles. In this chapter, we want to show how we can use semantic task specifications in combination with context-dependent conditions, for formally verifying the consistency of process models with their tasks and life cycles of artefacts.

The semantic task specifications are the central entity for connecting the other models together. They formally specify in a declarative way, via preconditions and postconditions, the actions that a task of a process model performs and relate them to artefacts in the (business) domain. Thus, they establish a connection between artefact-centric and process-centric models. In addition, they provide the means to express context-dependent information, which is only to be considered in certain situations. This context-dependent information is modeled via conditional rules and can be systematically applied to semantic task specifications. The executing services of tasks are invoked during enactment of the process model. Figure 6.1 illustrates how all these concepts are connected and work together.

As illustrated in the figure, there is no need for references from a task to its implementing services. The services may be chosen and substituted based on their specifications. However, in some cases an explicit selection of a service might be preferable, e.g., because of non-functional properties such as execution time. In this case, an explicit reference may be needed. However, the subtyping relationship with the semantic task specification has to be ensured.

A task and its effects on the (business) domain is fully established by describing the state before and after its execution. This state is expressed by the values of all artefacts in the domain. Essentially, a task performs some action that alters the values of artefacts. Hence, there exists an assignment of artefact properties before and after its execution. These changes in property assignments are defined by the semantic task specifications.

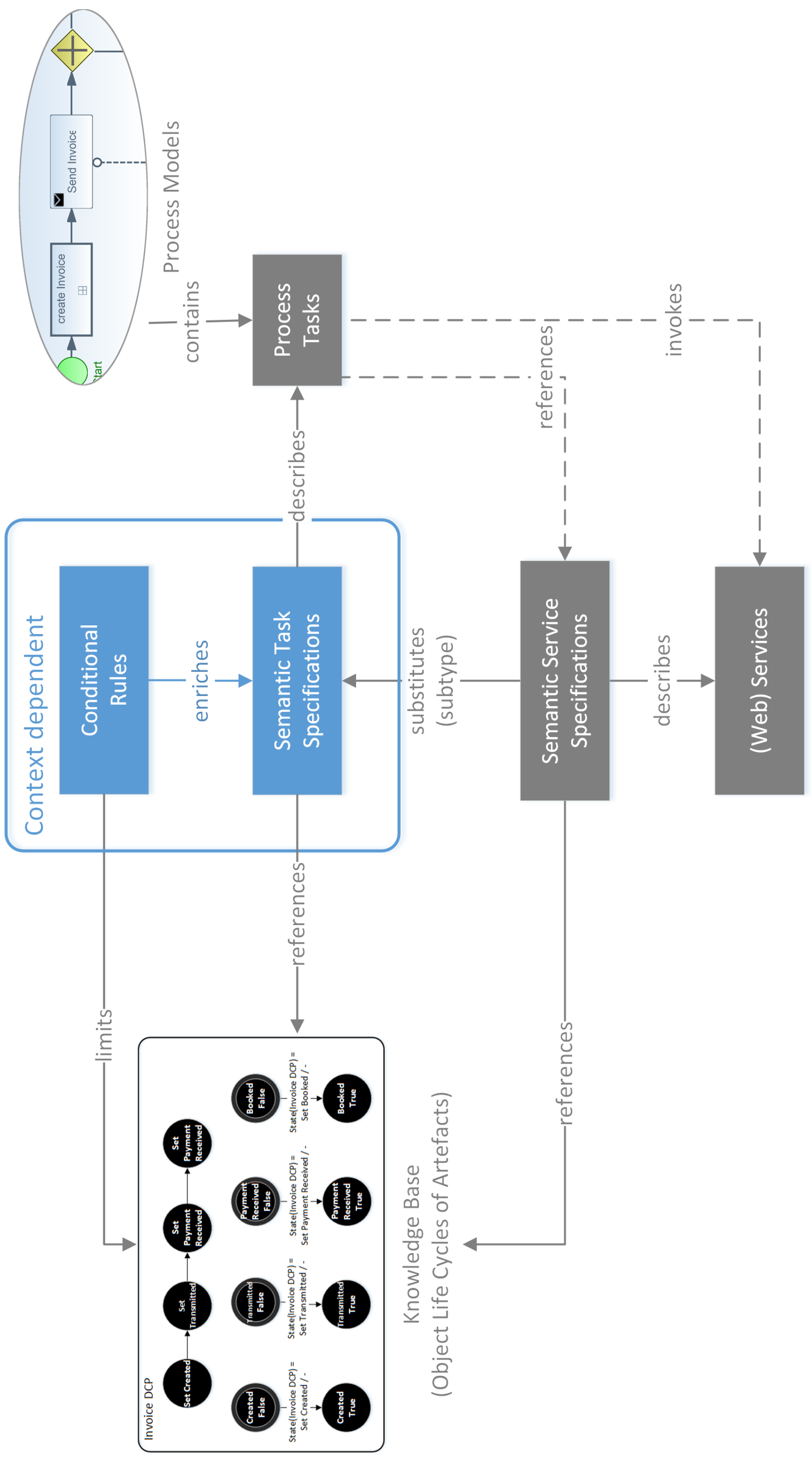


Figure 6.1: Approach for Formal Consistency Verification of Process Models and Artefacts

Since not all assignments are admissible at all times, we also need an object life cycle that constrains these assignments. By combining these parts, we are able to perform formal consistency verification.

For our formal verification approach, we decided to represent all models necessary in one modeling notation. As object life cycles are typically defined using FSMs, we opted to use FSMs as well to represent process models as illustrated in Figure 6.2. More precisely, we transform these models to several synchronized FSMs. This allows us to use one notation for our verification approach.

6.1 Connecting Process Models and Object Life Cycles Through Semantic Task Specifications

Until now, the process models and their tasks have only been connected to attributes of object life cycles. Basically, a task may only be executed if preconditions are fulfilled, i.e., certain attributes have specific values, and after execution postconditions are fulfilled, i.e., certain attributes are set to specific values. This does not yet account for the actual life cycle of the object and when attributes may be set or not. This becomes apparent when we consider, for example, Figure 5.2 where no conditions are specified on the transitions in the object life cycle. Thus, the tasks have to relate to the object life cycle, i.e., they should *trigger* transitions in it.

This allows us, not only implicitly, through the changes of attributes given by the postcondition, to define the action a task performs, but also explicitly by triggering specific transitions in the object life cycle. To be more precise, this decouples the action a task performs from its postcondition. The actual changes in the domain, i.e., the attribute of the artefact, are not performed by the task itself via its postcondition, but rather by the corresponding artefact-centric model, i.e., the extended object life cycle. By doing so the postcondition can be used to *ensure* that the execution of the task has yielded the expected changes. This is in contrast to performing the changes itself. Figure 6.3 illustrates this behavior.

Decoupling the postcondition from actually performing actions in the knowledge base is the foundation for our formal verification approach.

6.2 Transforming Extended Object Life Cycles to FSMs

The model of the object life cycle of a knowledge base, as shown in Figure 5.1, can be systematically translated into an FSM. Each state of the figure is translated to a state in the FSM. An exception is the first state, which is modeled as a separate state and used as an entry point for the FSM. Figure 5.3 shows the result of the translation.

The FSM does not contain any signals. This is due to the fact that there are no signals yet available, and that the transitions are triggered from the process and not from the object life cycle itself. However, to synchronize the FSMs of the process model, these signals have to be constructed systematically.

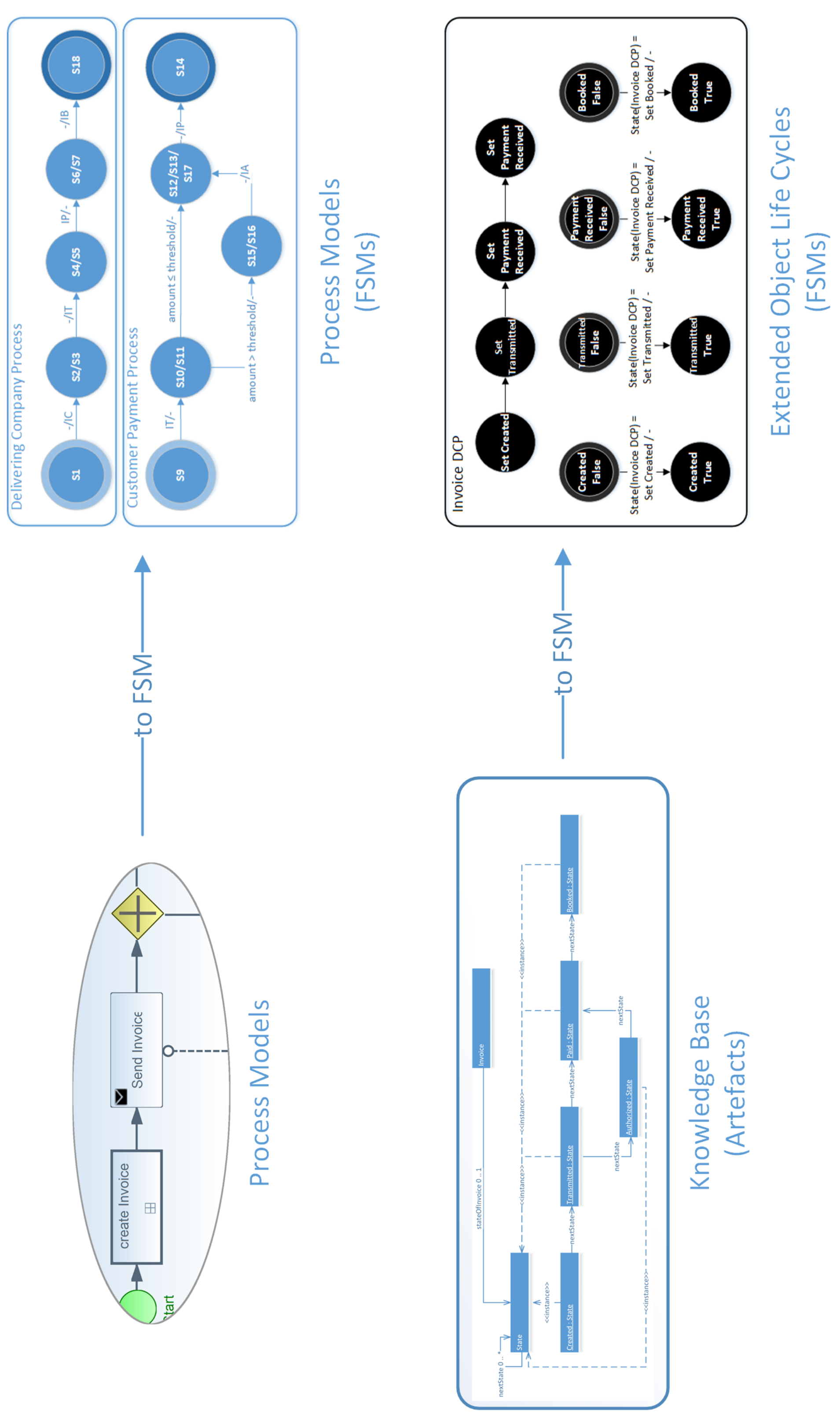

Process Models
to FSM
Process Models (FSMs)
Delivering Company Process
Customer Payment Process
Knowledge Base (Artefacts)
to FSM
Invoice DCP
Extended Object Life Cycles (FSMs)

Figure 6.2: Translating models to FSMs

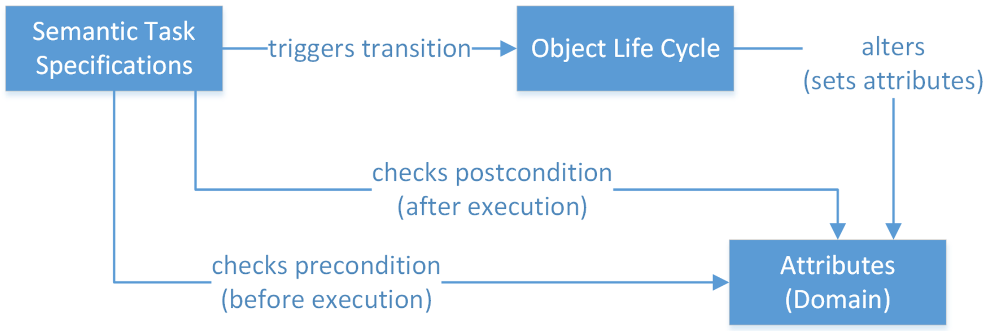
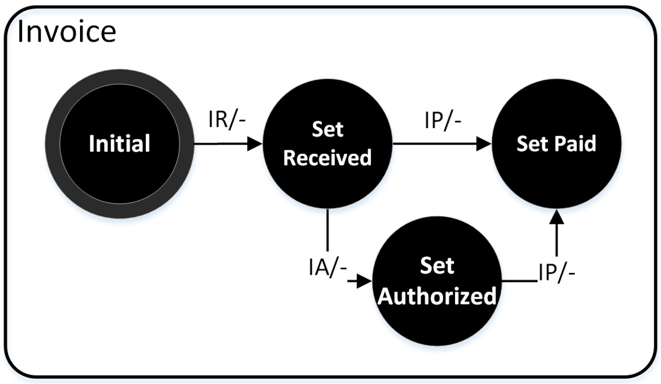

Semantic Task Specifications
triggers transition
Object Life Cycle
alters (sets attributes)
checks postcondition (after execution)
checks precondition (before execution)
Attributes (Domain)

Figure 6.3: Triggering changes in artefact-centric models

Each state in the object life cycle FSM can have a number of successor states. These states are identified via the *nextState* predicate. Again, the predicate can be omitted, and only the subject and the object are of interest. To be more precise, the subject, which is in this case the previous state, is not even necessary. For example, the state transition from *Created* to *Transmitted* is specified via the *nextState* predicate. It is of the form *nextState(SetCreated, SetTransmitted)*. However, we need additional information to create the signal, since we have to identify the object that the transition is based upon. When *stateOfInvoice(Invoice, SetCreated)* and *nextState(SetCreated, SetTransmitted)* are known, according to a logical analysis of these two statements, the next state of the *Invoice* must be *Transmitted*. Thus we can derive *stateOfInvoice(Invoice, SetTransmitted)* and determine *InvoiceSetTransmitted* as the signal of the state *SetCreated*. If a state has more than one successor states, then the corresponding signal has to be derived for each transition. This results in the FSM shown in Figure 6.4.



IR = InvoiceSetReceived; IA = InvoiceSetAuthorized; IP = InvoiceSetPaid;

Figure 6.4: Object Life Cycle with systematically constructed Signals of the Customer Invoice

Additionally, each attribute of an extended life cycle is modeled as an FSM as well. It consists of two states. The first state represents that the value of the attribute is false, and it is true in the last state. The name of the state is constructed as *[AttributeName][Value]*, e.g., for the attribute *Received* it is *ReceivedFalse* and *ReceivedTrue*. The transition between these two states is mapped to the corresponding set-state of the object life cycle, i.e., it is only triggered if the corresponding set-state is reached. Figure 6.5 shows the extended object life cycle with its attributes [87].

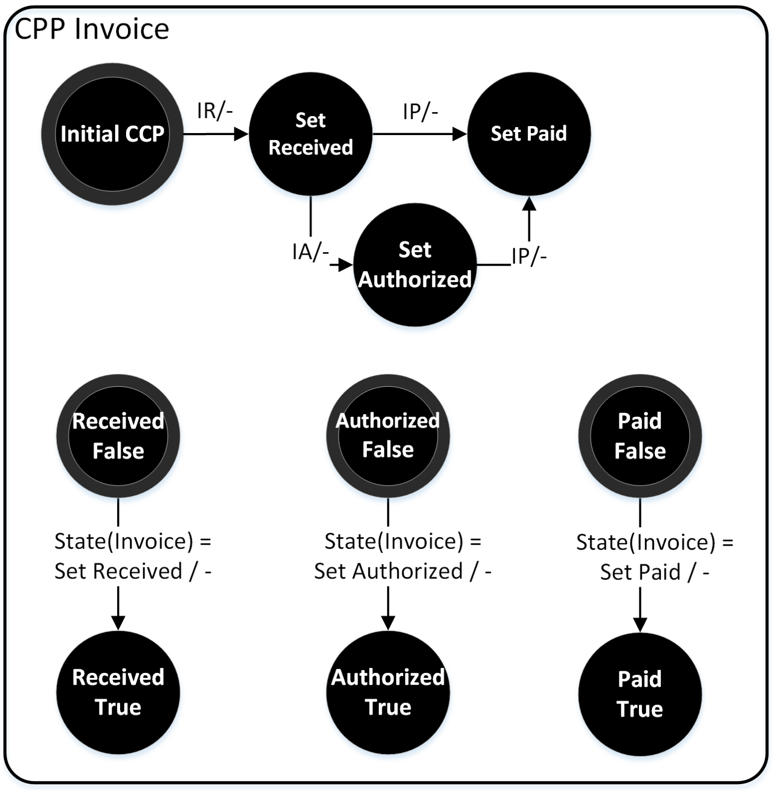


Figure 6.5: Extended Object Life Cycle with Attributes Customer Invoice

6.3 Transforming Enriched Process Models to FSMs

For a rigid approach to verification, it is necessary to have a systematic transformation in place that translates process models to synchronized FSMs. The transformation approach builds on the one defined in [87].

Each task in a process model is transformed to a corresponding FSM part. There are three states created for each task and a transition, without any condition, between them as illustrated in Figure 6.6.

The control flow between tasks in process models is directly translated to the FSM and connects the two adjacent tasks. For example, if the task *Transmit Invoice* is transformed into an FSM, then it is connected to the previously transformed task *Create Invoice* according to the control flow defined in the process model.

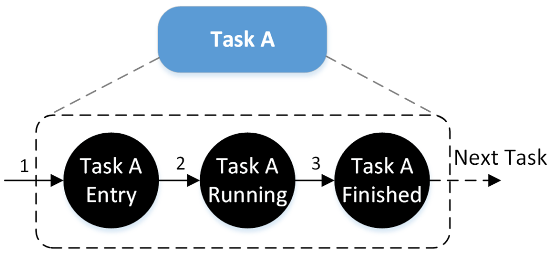


Figure 6.6: Translating Tasks of Process Models to FSMs

Each task is defined through a semantic task specification and this specification defines the transitions between the states. The first state *Task A Entry* represents the entry point of the task. This state may only be reached if the precondition of the corresponding semantic task specification is fulfilled. Thus, the preconditions impose a guard condition on the incoming transition of *Task A Entry.* The second state *Task A Running* represents a currently executing task. As mentioned in Section 6.1, each task may trigger the object life cycle. Hence, the incoming transition of *Task A Running* has a trigger. Finally, the third state *Task A Finished* represents an already completed tasks. This state may only be reached if all postconditions are fulfilled. Therefore, it includes a guard condition on its incoming transition. A translated task according to its semantic specification is shown in Figure 6.7.

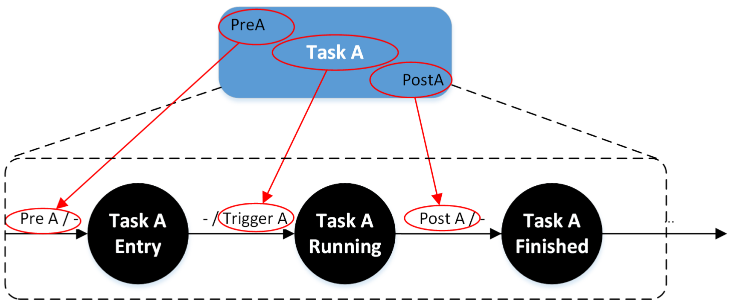


Figure 6.7: Translating Tasks with Semantic Specifications to FSMs

We differentiate three types of tasks in a process model: *Action Tasks*, *Send Tasks* and *Receive Tasks*. Action Tasks are used to perform some sort of action that alter the domain. These are the most common tasks in process models and they behave as described above. *Send*- and *Receive-Tasks* are special kinds of tasks and thus have to be treated separately. These tasks are implemented by a service, but are also specified using semantic task specifications.

In case of the *Receive-Task*, a waiting operation for an incoming message, or more generally an external trigger, is in place. An incoming message has to be received first, and only then the next transition is executed. Hence, an additional incoming signal is necessary in the FSM to model this waiting operation, see Figure 6.8.

The corresponding *Receive-Task*, for example *Receive Invoice*, has an incoming signal

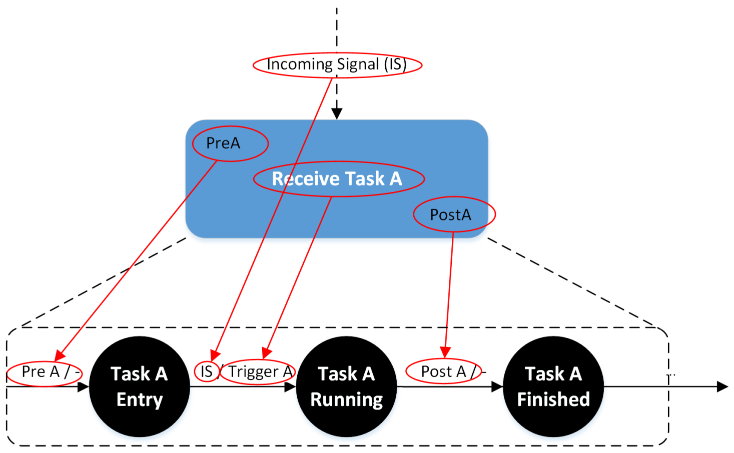


Figure 6.8: Translating Receive Tasks with Semantic Specifications to FSMs

and is only triggered when this signal is set by an external entity. This signal can be directly translated from the process model. If a task has an incoming message edge, then the signal is constructed based on this edge and the task name. For example, the *Receive Invoice* task of Figure 2.6 has one incoming message edge from the task *Transmit Invoice*. Hence, the trigger signal of the corresponding entry state in an FSM can be systematically constructed as *[IncomingTaskName]*, e.g., *TransmitInvoice*.

Send-Tasks are the counterpart to the Receive-Tasks as they send messages to other participants. These messages are sent concurrently to the execution of the task and thus are modeled as an additional trigger, see Figure 6.9. The signals of these tasks can be constructed systematically as *[SendTaskName]*.

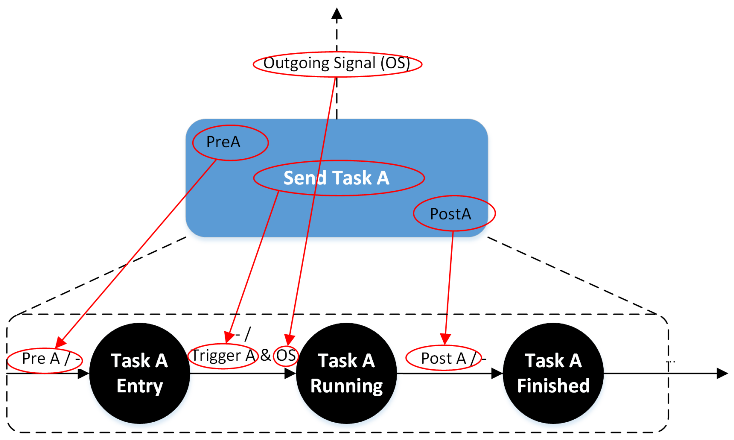


Figure 6.9: Translating Send Tasks with Semantic Specifications to FSMs

In essence, *Send-* and *Receive-Actions* are used to synchronize parallel processes. Their signals are not related to the extended object life cycle as they are not influencing the domain artefacts.

The transformation of all three types of tasks can be shown together as illustrated in Figure 6.10.

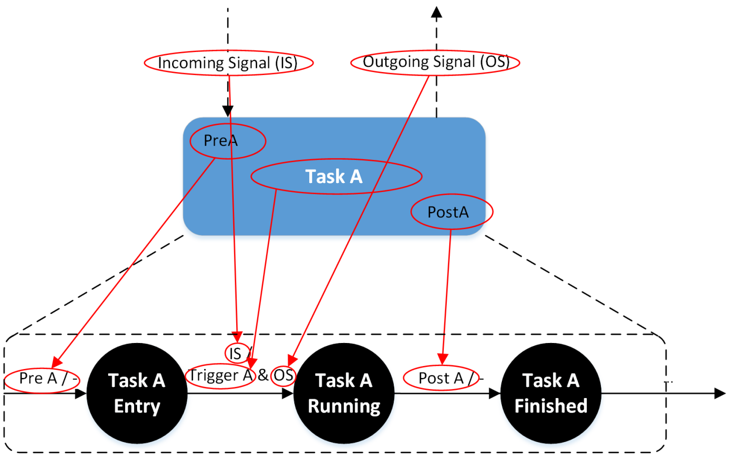


Figure 6.10: Generalized Transformation of Tasks with Semantic Specifications to FSMs

The guard conditions and the trigger signals of the tasks have to be constructed systematically to fit the existing extended object life cycles that the semantic task specifications reference. As mentioned in Section 5.3, the semantic specifications are formally grounded in extended object life cycles. Hence, the preconditions, the postconditions and the actions that they perform reference the object life cycle and its attributes.

First, let us consider the action a task executes. As mentioned above, the action is actually a trigger that produces a state transition in the object life cycle. This is formally defined by the predicate *triggerTransition*. For example, the *Authorize Invoice* task triggers the transition into the state *Set Authorized* of the object life cycle. We can use the subject and object to systematically construct the trigger signal in the FSM. The specification of the performed action of *Authorize Invoice* is *triggerTransition(Invoice, Set Authorized)*. We can use the subject, *Invoice*, and the object, *Set Authorized*, to systematically construct the signal *InvoiceSetAuthorized*. This resulting signal is the same as the systematically constructed signal in the object life cycle shown in 6.2, connecting the actions that a task performs via a triggering signal with the FSM of the object life cycle.

A similar systematic approach is needed for the pre- and postconditions. Here the semantic specification does not use the object life cycle, but rather references the attributes of our extended version of object life cycles. The conditions are specified using the *attributeSet* predicate, which is defined on a subject, an artefact, and an object, a state of the artefact. We can now systematically construct the conditions as *state([Subject][Object]True)*. This essentially states that the condition is only fulfilled if the subject is in state *[Subject][Object]True*. Let us consider the *Authorize Invoice* task and its semantic specification again. The precondition is defined by *attributeSet(Invoice, Received)*. The resulting guard condition in our FSM is *state(InvoiceReceivedTrue)*, which is the same as the systematically constructed name for states of attributes. Hence, we connect the resulting FSMs of the process model and the object life cycle through preconditions. The same approach is applied for postconditions as well.

In addition, *Decision Nodes* and *Merge Nodes* have to be treated separately. A corresponding state in the FSM is created for both nodes, however, their incoming and outgoing edges are treated differently. As *Decision Nodes* split the execution path of a process model it may only contain one incoming edge from the previous state. This state correlates to the last action, according to the control flow, of the BPM. The number of outgoing edges correlate to the number of outgoing control flows of the BPM. If one of these outgoing control flows also contains a condition, the condition is directly translated to the corresponding outgoing edge of the decision state. The counterpart of the *Decision Node* is the *Merge Node*. A *Merge Node* may have several incoming control flows and only one outgoing control flow. In the FSM, the *Merge Node* is represented as a single state and its incoming edges correspond with the incoming control flows of the node in the BPM. However, in contrast to the BPM, each incoming edge is directly connected to the previous state of the corresponding control flow. The outgoing edge is directly connected to the following state according to the control flow of the process model.

The resulting FSM of the customer payment process of our running example is shown in Figure 6.11.

As this representation is very verbose we opt to use a simplified version. We can combine the three states that are created for a task to just one state. In this version, we combine the precondition and the trigger on one incoming transition of the task's FSM. The outgoing transition contains the postcondition as a guard. This outgoing transition is also combined with the incoming transition of the subsequent task. More precisely, it is combined with the precondition of the next task. Figure 6.12 illustrates this simplification.

This simplified depiction, however, does prevent us from showing some of the more intricate characteristics of our verification approach. Hence, we use the simplified version to give an overview and only resort to the more verbose version if it is necessary to show specific details of the verification.

Using this transformation approach with the simplification, we can systematically construct an FSM representation of the process models of our running example. The resulting FSMs of these processes, together with their synchronization through their Receive- and Send-tasks, are shown in Figure 6.13.

Using the systematically constructed signals, the models can be connected to the FSMs of the extended object life cycles from Section 6.2. The resulting FSMs are shown all together in Figure 6.14.

We are aware, however, that certain more complex processes cannot be translated to FSMs due to their inherent limitation of expressiveness. Thus, we only translate a subset that is sufficient for our verification approach. For example, BPMN multi-instance tasks, which can create a potentially infinite number of parallel task instances, cannot be translated to FSMs, also not to a fixed number of FSMs that are synchronized with signals. In essence, we do not support multiple instances of tasks, parallelism and loops in BPM with our verification approach.

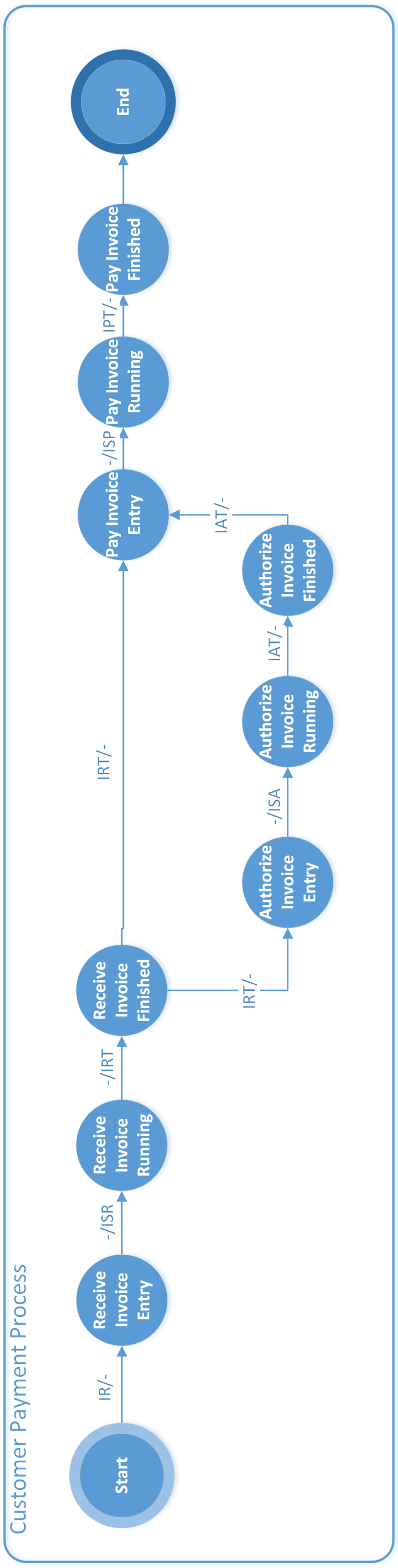


Figure 6.11: Customer Payment Process represented as an FSM

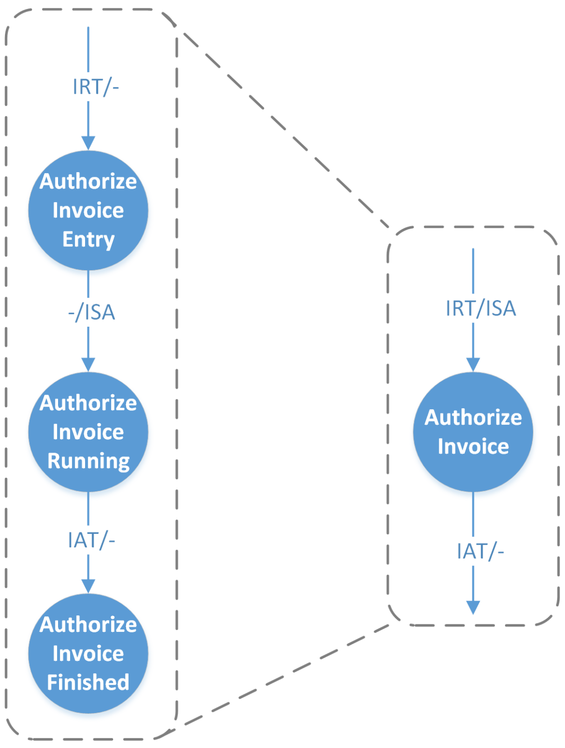


Figure 6.12: Simplifying Tasks in FSMs

6.4 Modeling Business Rules as FSM Properties

There are several options how a business rule can be formalized. For the purpose of model-checking, a formula in temporal logic would typically refer to the states of the process model, but this requires that the property modeler knows the process model he is supposed to check [90]. However, we use our extended life cycle models of business artefacts, so that the formulas can refer to their states instead.

Let us explain this approach with the the rule defined in (Business) Rule 1.1. Strictly speaking, this business rule defines a characteristic of the data object Invoice. So, relating its formalization to a model of the Invoice object life cycle seems to be straightforward. We use our life cycle models to specify the (business) rule, without any additional knowledge of the process itself and especially its control flow. Still, variables for specifying characteristics of the Invoice object may be relevant, in our example *amount*. In addition, the threshold value defined in the customer company is needed, which is defined as an additional constant.

Using our formalization of (business) rules using implications of Section 4.3, we can express them as CTL properties using *always globally*. The *AG* operator states that for all possible transitions its content has to hold in each state. Essentially, our (business) rule in propositional logic specifies a global constraint that has to be fulfilled. Thus, its translation to CTL is straight forward as we "only" have to add our propositional logic

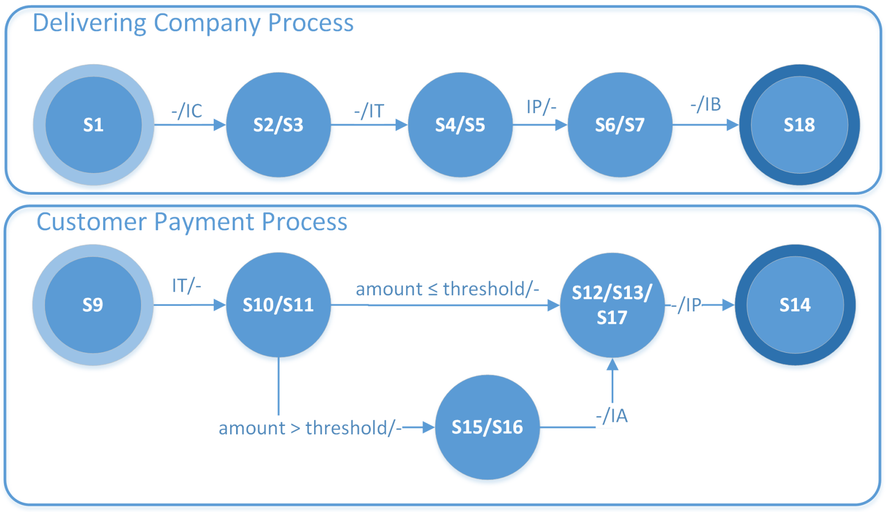


Figure 6.13: Process Models as simplified FSMs

formula to the AG operator, which checks if the condition is fulfilled after all transitions, i.e., at all times. The following CTL formula defines a property corresponding to the (Business) Rule 1.1 as

$$\begin{aligned}\mathbf{AG}(((Invoice.state\ =\ paid))\ \rightarrow\\ ((Invoice.amount\ \geq\ threshold)\ \rightarrow\ (Invoice.state\ =\ authorized)))\end{aligned} \tag{6.1}$$

The same formalization can be used for our more complex (Business) Rule 4.2.

$$\begin{aligned}\mathbf{AG}((Invoice.state\ =\ paid)\ \rightarrow\\ (((Invoice.amount\ \geq\ threshold)\ \rightarrow\ (Invoice.state\ =\ authorized))\ \wedge\\ (\neg(Invoice.amount\ \geq\ threshold)\ \rightarrow\ \neg(Invoice.state\ =\ authorized))))\end{aligned} \tag{6.2}$$

Using the simplified version of Equation 4.8 yields the property shown below:

$$\begin{aligned}\mathbf{AG}((Invoice.state\ =\ paid)\ \rightarrow\\ ((Invoice.state\ =\ received)\ \wedge\\ ((Invoice.amount\ \geq\ threshold)\ \wedge\ (Invoice.state = authorized))\ \vee\\ (\neg(Invoice.amount\ \geq\ threshold)\ \wedge\ \neg(Invoice.state = authorized))))\end{aligned} \tag{6.3}$$

So, while the object life cycle on its own does not represent the business rule, of course, these properties together with it do.

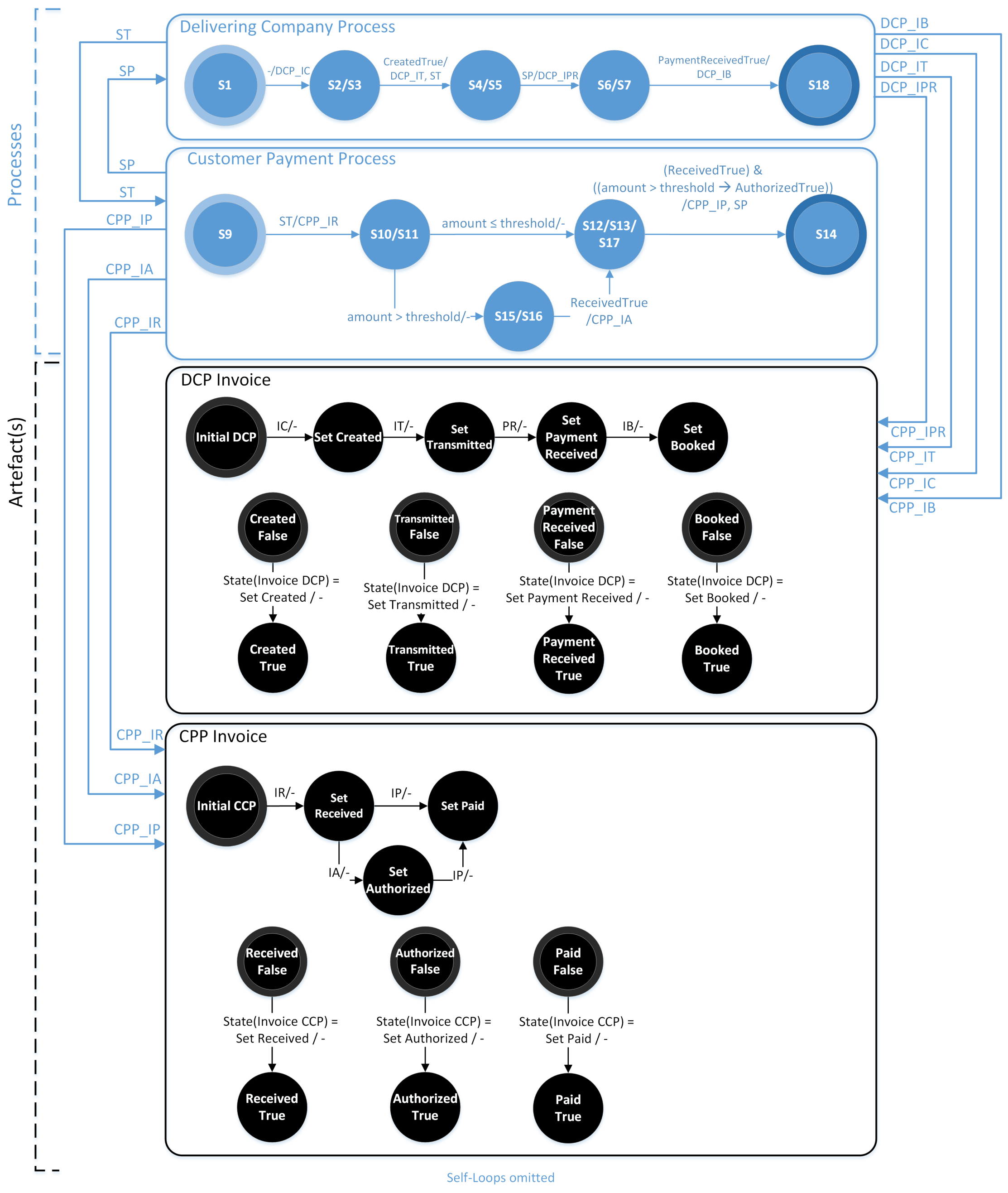


Figure 6.14: Simplified Synchronized FSMs for Running Example

CHAPTER 7

Evaluation and Results

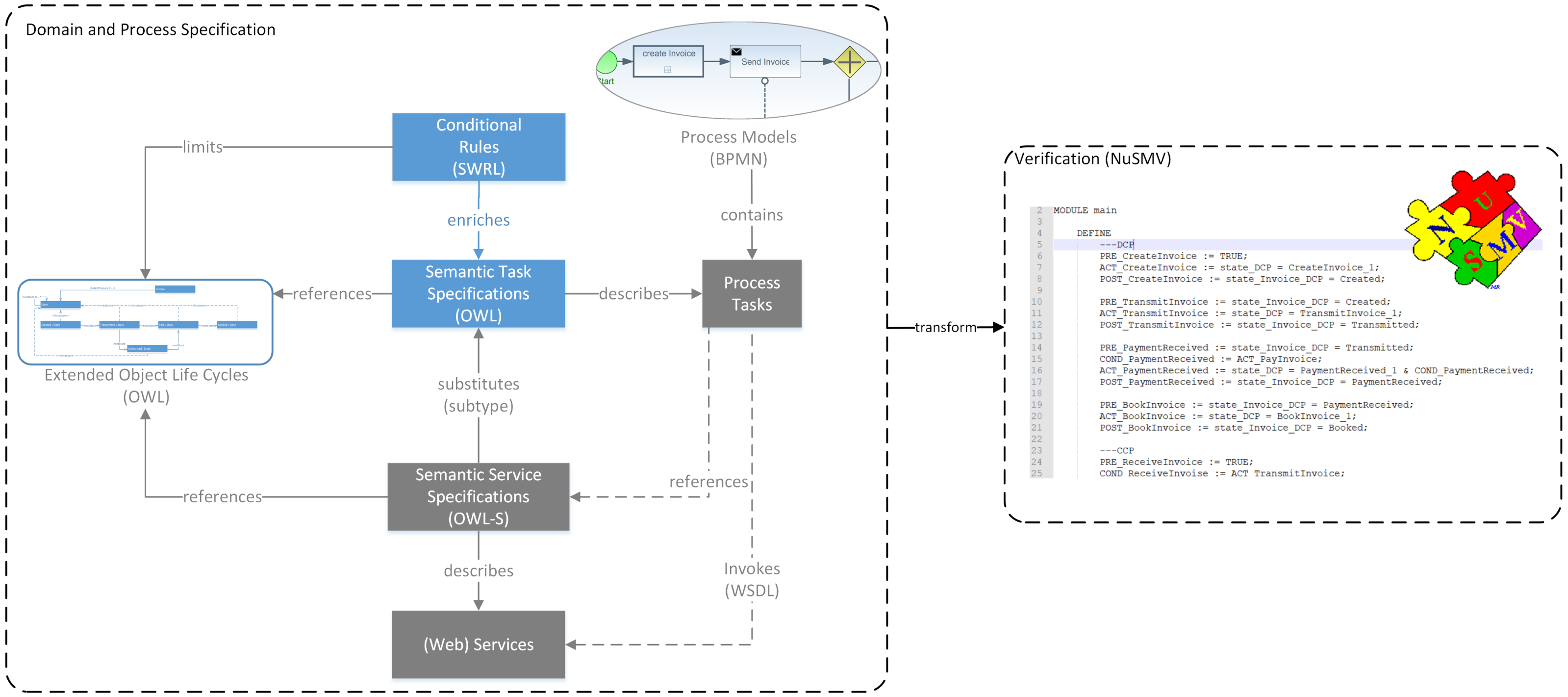
In the previous chapters, we explained how the conceptual models can be connected with each other through semantic task specifications and how their consistency can be formally verified. These models are not bound to one specific modeling notation, but rather represent higher conceptual ideas that can be expressed in a variety of notations. However, for formally verifying concrete instances of process models one notation has to be selected. This chapter presents an evaluation of our formal verification approach using BPMN, OWL and NuSMV.

We use ontologies in OWL to represent the domain knowledge, i.e., our knowledge base. The rationale for OWL is, that it already supports semantic specification of concepts via description logic. With OWL it is relatively simple to create semantic concepts of object life cycles and their attributes as illustrated in Section 5.1.

There are many pre-defined upper ontologies available for a multitude of application fields. One such upper-ontology is OWL-S, which provides the means to semantically specify services. We use these specifications to relate services to our object life cycle ontology defined above. In essence, the preconditions and effects of a service are related to states or attributes in an object life cycle.

Another benefit of using OWL is the availability of inference rules. There are many rule engines available and we decided to use SWRL and the tool Protege. In essence, SWRL models an implication between a body and a head. The head represents the consequence that must hold if the body evaluates to true. We use SWRL as a means to specify our context information as conditional rules (see Section 4.3). For our formal verification approach, we do not use the inference that rule engines provide, but rather use SWRL as a means to specify information that is further processed.

For process models we decided to use BPMN. BPMN is not only a well-established modeling notation for BPMs, but also provides the means to extend and customize its notation. We use this extension mechanism to connect tasks with their corresponding semantic task specifications. Further details on BPMN are given in Section 2.1.2.



```
MODULE main

    DEFINE
        ---DCP
        PRE_CreateInvoice := TRUE;
        ACT_CreateInvoice := state_DCP = CreateInvoice_1;
        POST_CreateInvoice := state_Invoice_DCP = Created;

        PRE_TransmitInvoice := state_Invoice_DCP = Created;
        ACT_TransmitInvoice := state_DCP = TransmitInvoice_1;
        POST_TransmitInvoice := state_Invoice_DCP = Transmitted;

        PRE_PaymentReceived := state_Invoice_DCP = Transmitted;
        COND_PaymentReceived := ACT_PayInvoice;
        ACT_PaymentReceived := state_DCP = PaymentReceived_1 & COND_PaymentReceived;
        POST_PaymentReceived := state_Invoice_DCP = PaymentReceived;

        PRE_BookInvoice := state_Invoice_DCP = PaymentReceived;
        ACT_BookInvoice := state_DCP = BookInvoice_1;
        POST_BookInvoice := state_Invoice_DCP = Booked;

        ---CCP
        PRE_ReceiveInvoice := TRUE;
        COND ReceiveInvoise := ACT TransmitInvoice;
```

Figure 7.1: Technologies for Evaluating Verification Approach

Figure 7.1 provides an overview of the technologies used for evaluating the verification approach. As this figure suggests, the verification is not performed directly on the models. We use *model checking* via the tool NuSMV for formally verifying the consistency of the models involved. To perform formal verification, we have to generate a representation of our models in NuSMV. This representation has to contain combined information of the models as well as the definition of the conditional rules as *properties*. The properties are actually what we verify our models against. NuSMV supports properties expressed in CTL (see Section 6.4).

Before we can define the transformation of our models to NuSMV, we first have to specify how we logically connect the models together. For logically connecting task- and artefact-centric models, we adopt parts of OWL-S specifications previously defined for services and employ them for semantic task specification. In essence, we utilize the possibility to specify conditions and effects, to provide a formal semantic specification of all tasks that are embodied in a process model. That is, each task in BPMN has a counter-part in OWL-S, which semantically specifies its behavior. The conditions and effects relate to the object life cycle defined in OWL and thus connect the two models together.

The predicate *hasResult* in OWL-S couples outputs and effects. Effects are closely related to tasks, since they specify how the domain objects change, and are specified via the *hasEffect* predicate. To be more precise, they specify the changes that are caused by the service execution. In our approach, we relate these changes to state transitions in an object life cycle. That is, each service execution may change the state of a domain object.

The complete semantic specification of a task in OWL-S is very verbose. We show only an excerpt containing the output and the effect in Listing 7.1. The task is identified via the *rdf:about="AuthorizeInvoice"* statement and is modeled as an *AtomicProcess*. An *AtomicProcess* executes an atomic operation, for example a WSDL operation. The next part, starting with the predicate *hasResult* in Listing 7.1, describes the result. It shows the output (*rdf:ID="InvoiceOutput"*) and its binding. The binding is actually omitted here, since it does not influence the specification of the effects. These are specified in the *hasEffect* predicate. In Listing 7.1, an SWRL-Rule is used to define the effect. The rule is identified by the rdf:ID *StateTransition*. Recalling that all statements in RDF are represented as triples of the form *Subject*, *Predicate* and *Object*, we can determine the meaning of the rule. The rule basically specifies that in our domain ontology the predicate *stateOfInvoice* is set for the subject *InvoiceOutput* to the object *Set Authorized*.

Listing 7.1: Excerpt of OWL-S Specification of Authorize Invoice Task with its Effect

```
<process:AtomicProcess rdf:about="AuthorizeInvoice">
  ...
  <process:hasResult>
     <process:Result>
      <process:hasResultVar>
         <process:ResultVar rdf:ID="InvoiceOutput">
          <process:parameterType rdf:resource="#Invoice"/>
         </process:ResultVar>
      </process:hasResultVar>
      <process:withOutput>
         <process:OutputBinding>
          ...
         </process:OutputBinding>
      </process:withOutput>
      <process:hasEffect>
        <expr:SWRL-Condition rdf:ID="StateTransition">
        <swrl:AtomList>
          <rdf:first>
            <swrl:IndividualPropertyAtom>
              <swrl:propertyPredicate rdf:resource="stateOfInvoice"/>
              <swrl:argument1 rdf:resource="#InvoiceOutput" />
              <swrl:argument2 rdf:resource="#SetAuthorized" />
            </swrl:IndividualPropertyAtom>
          </rdf:first>
        </swrl:AtomList>
        </expr:SWRL-Condition>
      </process:hasEffect>
     </process:Result>
  </process:hasResult>
  ...
</process:AtomicProcess>
```

With these specifications, the tasks are directly related to the object life cycles of the domain objects [87].Additionally, we use SWRL-Rules to specify context information as conditions on object life cycles. Let us consider our *Invoice* example again to illustrate this. A service *Authorize Invoice* is semantically specified and has the effect that an *Invoice* is authorized. In this case, the effect of the task execution is that the attribute *Authorized* of the *Invoice* is set to the state *AuthorizedTrue*. Such information basically expresses that “An Invoice has to be received before it can be authorized”. The naive implementation of this rule is shown in Listing 7.2.

Listing 7.2: Naive implementation of Received before Authorized Rule

```
  attributeSet(Invoice,ReceivedTrue) -> attributeSet(Invoice,AuthorizedTrue)
```

This rule, however, is not correct as it expresses an implication between *ReceivedTrue* and *AuthorizedTrue*. This implication states that if an Invoice has been received, and attributeSet(Invoice, ReceivedTrue) evaluates to true, an Invoice also has to be authorized, attributeSet(Invoice, AuthorizedTrue) evaluates to true. This is not what the rule above given in natural language expresses and it is also logically not correct as not all received invoices have to be authorized.

We actually have to switch the two statements in the implication or the direction of the implication as illustrated in Listing 7.3. This rule states, that in order for attributeSet(Invoice, AuthorizedTrue) to be evaluated to true also attributeSet(Invoice, ReceivedTrue) has to be evaluated to true. Otherwise, the implication would fail and the rule would be broken.

Listing 7.3: Implementation of Received before Authorized Rule

```
attributeSet(Invoice,AuthorizedTrue) -> attributeSet(Invoice,ReceivedTrue)
```

These rules can be used to systematically enrich semantic task specifications. In essence, they add additional preconditions to a semantic task specification with matching postcondition. For example, if there is a semantic task specification that has in a postcondition that the Authorized attribute should be set (attributeSet(Invoice,AuthorizedTrue)), then we can match the body of the rule with this postcondition. Since the rule specifies a condition that has to be fulfilled for it to evaluate to true, we can derive that the consequence of the rule should be added as precondition as well.

Let us express this with a more complex example. In our running example, there is a task called *Pay Invoice* in the *Customer Payment Process*. It can be reached through two paths: directly from *Receive Invoice* or diverted through *Authorize Invoice*. Which path is taken depends on the amount of the received invoice. If the amount is larger than a threshold, then an extra authorization step is necessary. The corresponding rule is shown in (Business) Rule 1.1. Let us assume the (business) rule is formalized as shown in Listing 7.4.

Listing 7.4: Implementation of Authorized before Paid Rule

```
attributeSet(Invoice,PaidTrue) → (attributeSet(Invoice,ReceivedTrue) ∧ (¬(
    Invoice.amount ≥ threshold) ∨ (attributeSet(Invoice,AuthorizedTrue) ∧
    (Invoice.amount ≥ threshold))))
```

Actually, the rule in Listing 7.4 can be simplified to the one in Listing 7.5.

Listing 7.5: Implementation of Authorized before Paid Rule

```
attributeSet(Invoice,PaidTrue) → (attributeSet(Invoice,ReceivedTrue) ∧ (¬(
    Invoice.amount ≥ threshold) ∨ attributeSet(Invoice,AuthorizedTrue)))
```

The task *Pay Invoice* has a semantic task specification that does not differ from its implementing semantic service specification, which states that a received invoice is necessary to make a payment. The semantic task specification is expressed in Listing 7.6.

Listing 7.6: Pay Invoice Semantic Task Specification

```
Pay Invoice:
   Pre: attributeSet(Invoice,ReceivedTrue)
   Eff: attributeSet(Invoice,PaidTrue)
```

As we can see, the postcondition *attributeSet(Invoice, PaidTrue)* of the Pay Invoice task matches the body of the (business) rule. Thus, we can add the conditions of the rule as an additional precondition to the semantic task specification. Listing 7.7 shows this enriched semantic task specification.

Listing 7.7: Enriched Pay Invoice Semantic Task Specification

```
Pay Invoice:
   Pre: attributeSet(Invoice,ReceivedTrue) ∧ ((attributeSet(Invoice,
      ReceivedTrue) ∧ ¬(Invoice.amount ≥ customer.threshold) ∨ (
      attributeSet(Invoice,AuthorizedTrue))))
   Eff: attributeSet(Invoice,PaidTrue)
```

The question is, does this enriched semantic task specification still fulfills the subtyping relationship with the implementing service? To answer this question we have to consider the precondition as a Boolean logic formula. The precondition of a subtype has to be equal or weaker than the one of the supertype. Hence, all interpretations in which the enriched semantic task specifications evaluate to true, have also to be true for the subtype. We can directly conclude that this has to be true as the original precondition is and-connected with the additional conditions, i.e., the enriched precondition can only evaluate to true if *attributeSet(Invoice, ReceivedTrue)* evaluates to true. The formula can be simplified to the one in Listing 7.8.

Listing 7.8: Simplified Enriched Pay Invoice Semantic Task Specification

```
Pay Invoice:
   Pre: (attributeSet(Invoice,ReceivedTrue) ∧ ¬(Invoice.amount ≥ customer.
      threshold)) ∨ attributeSet(Invoice,AuthorizedTrue)
   Eff: attributeSet(Invoice,PaidTrue)
```

In the same way we can systematically enrich the PayInvoice task with (Business) Rule 4.2. The result is shown in Listing 7.9 and corresponds with the property for NuSMV defined in Equation 4.8.

Listing 7.9: Simplified Enriched Pay Invoice Semantic Task Specification for Authorize Only

```
Pay Invoice:
   Pre: (attributeSet(Invoice,ReceivedTrue) ∧ ((¬(Invoice.amount ≥ customer.
       threshold) ∧ ¬(attributeSet(Invoice,AuthorizedTrue))) ∨ ((Invoice.
       amount ≥ customer.threshold) ∧ attributeSet(Invoice,AuthorizedTrue)))
   Eff: attributeSet(Invoice,PaidTrue)
```

In our approach, we use BPMN as the notation for process models. We connect each task of our running example in Figure 2.6 with a corresponding task specification in OWL-S. The result is shown in Figure 7.2, witch each task containing annotations illustrating its semantic task specification. This is similar to the annotations in [111]. In case of existing task specifications in OWL-S, they can be reused, otherwise they have to be created from scratch.

For our systematic approach, each and every task in BPMN needs to have a semantic task specification given in OWL-S. Also for tasks that are not implemented by a (Web) service, a corresponding OWL-S specification has to exist. An example of such a task is *Transmit Invoice*. This task is a BPMN *Send-Task*, which is used to send messages to other tasks or processes. Such tasks sometimes do not use a service implementation for execution, but are directly executed by the BPMN engine. We employ OWL-S specifications for such tasks as well. Another example is the BPMN *Receive-Task*, which is the counterpart of a BPMN *Send-Task*.

We utilize the extension mechanism of BPMN to connect tasks with OWL-S specifications. The custom tag *semanticTaskRef* defines the reference to the corresponding OWL-S specification for a task in BPMN. Listing 7.10 shows how the connection between BPMN and the semantic task specification is established.

Listing 7.10: Connect BPMN Task to Semantic Task Specification

```
<serviceTask id="TransmitInvoice" name="Transmit Invoice" ...>
    ... // INPUT/OUTPUT Binding
    <extensionElements>
      <semService:semanticTaskRef id="transmitInvoiceSemanticTaskRef">
        http://ict.tuwien.ac.at/ontologies/semanticTaskSpec#TransmitInvoice
      </semService:serviceRef>
    </extensionElements>
    ...
</serviceTask>
```

The BPMN engine does only need a reference to the semantic task specification, all additional information can be accessed from there. We use a Uniform Resource Identifier (URI) to reference a specific instance of an semantic task specification in our knowledge base given in OWL. To process this information during enactment of the process, the

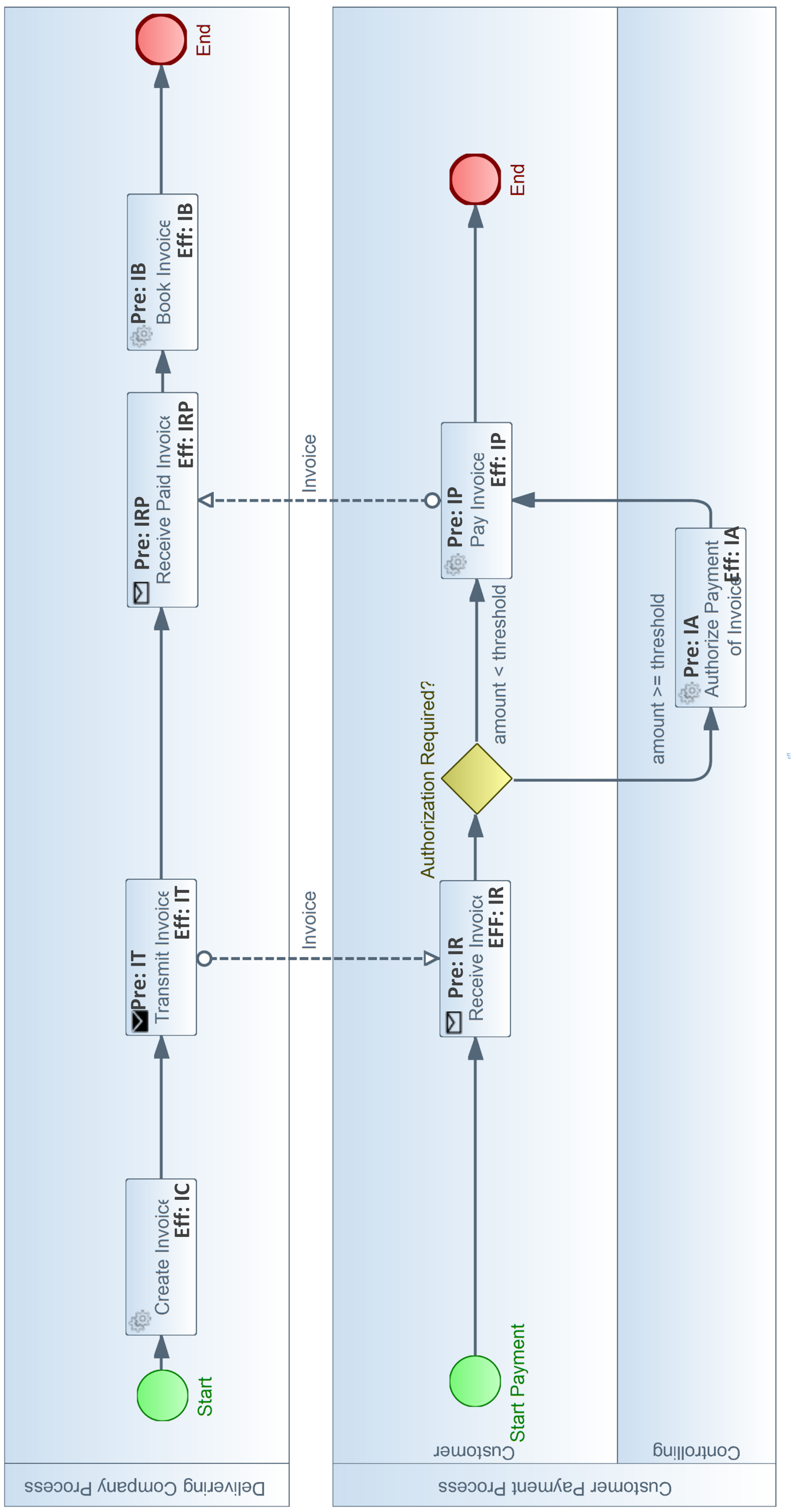


Figure 7.2: Running Example in BPMN with Annotations

tasks in BPMN has to be adapted, of course. Since our focal point is on verification and not execution we do not have to provide an adapted BPMN execution engine.

Using these techniques allows us to connect all our models together. However, the actual verification of their consistency still remains open. As stated above, formal verification using model checking in NuSMV requires a representation in NuSMV source-code. We decided against a direct translation of our models to NuSMV source-code as it would limit using our approach with other technologies. Instead, we opted to translate our models to an intermediate representation that is already closely resembling how models in NuSMV operate.

FSMs are our representation of choice, since NuSMV also operates on state machines. From a technical point of view we utilize *state machines* from the UML. Figure 7.3 illustrates the transformation of the higher-level models to NuSVM source-code.

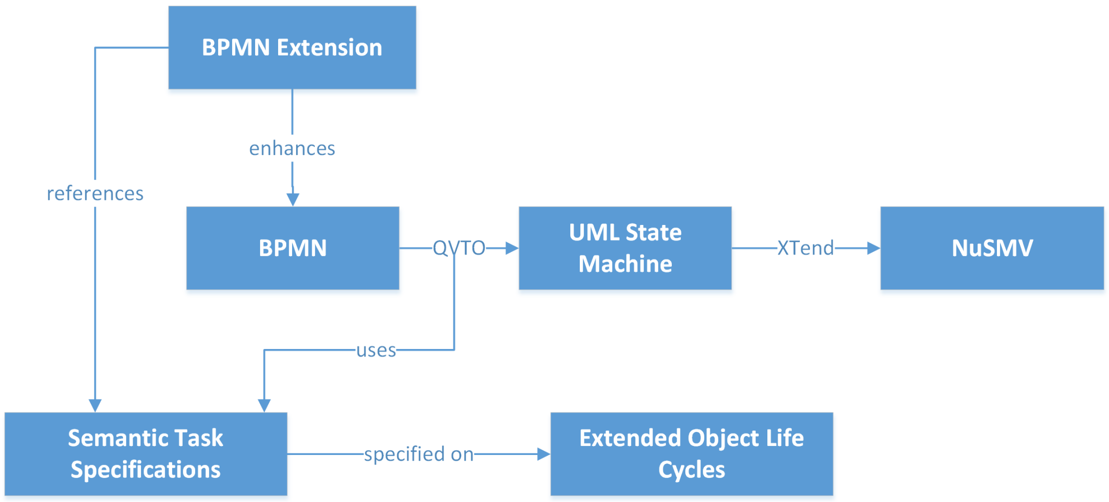


Figure 7.3: Transformation Approach for BPMN models

We use model-driven technologies to perform the transformation of the models. First, we transform our process models given in BPMN with Operational Query View Transformation (QVTO) to UML state machines. QVTO is a rule-based model-to-model transformation language. In the course of this transformation the references to the semantic task specification, provided by extended BPMN tasks, are used to create the state machines accordingly. The transformation of process models in BPMN to FSMs is based on the one described in Section 6.3.

Transforming of BPMN models to FSMs and their interleaved establishment of connections with the FSMs representing object life cycles works systematically according to the following steps:

1. For each Pool in BPMN, create an FSM.
2. For each Task or Gateway in this Pool, create states in the FSM according to Section 6.3.
3. If this Task has an outgoing message, create a corresponding signal setting on each outgoing transition of the resulting states.
4. If this Task has an incoming message, create a signal trigger on the corresponding outgoing transition of the states.
5. For each BPMN control flow element, create a corresponding transition.
6. For each annotated BPMN control flow element, (additionally) create a corresponding signal setting.
7. For each condition specified on a control flow element (from a Gateway), create a corresponding signal trigger on the corresponding outgoing transition of the state (of this Gateway).

The object life cycles and their attributes represented in OWL are transformed to FSMs according to Section 6.2. For this transformation, we use QVTO as well.

The synchronized FSMs at the intermediate level are self-contained, i.e., they contain all information necessary for creating an NuSMV representation. The transformation of the intermediate FSMs uses a template-based model-to-text approach. For our evaluation, we use Xtend to generate the NuSVM source-code. Listing 7.11 shows the resulting NuSMV source-code representation of the invoice used in the delivering company process. The transition from a state to another is encoded in the *next case* statement and only triggers if the corresponding signal is set.

Listing 7.11: Object Life Cycle represented in NuSMV source-code

```
    init(InvoiceDCPLifeCycle) := InvoiceDCPLifeCycleInitial;
    next(InvoiceDCPLifeCycle) :=
      case
        (InvoiceDCPLifeCycle = InvoiceDCPLifeCycleSetTransmitted) :
            InvoiceDCPLifeCycleSetPaymentReceived;
        (InvoiceDCPLifeCycle = InvoiceDCPLifeCycleInitial) &
            DCPCreateInvoiceSignalEvent : InvoiceDCPLifeCycleSetCreated;
        (InvoiceDCPLifeCycle = InvoiceDCPLifeCycleSetPaymentReceived) &
            DCPLifeCycleSetBookedSignalEvent : InvoiceDCPLifeCycleSetBooked;
        (InvoiceDCPLifeCycle = InvoiceDCPLifeCycleSetCreated) &
            DCPTransmitInvoiceSignalEvent :
            InvoiceDCPLifeCycleSetTransmitted;
        no_signal_InvoiceDCPLifeCycle : InvoiceDCPLifeCycle;
        TRUE : InvoiceDCPLifeCycleError;
      esac;
```

Attributes of our extended object life cycles are essentially represented in the same manner as they are also expressed as FSMs. The *created* attribute of the invoice used in the delivering company is shown in Listing 7.12.

Listing 7.12: Attribute of an Extended Object Life Cycle represented in NuSMV source-code

```
    init(InvoiceDCPCreated) := InvoiceDCPCreatedFalse;
    next(InvoiceDCPCreated) :=
      case
        (InvoiceDCPCreated = InvoiceDCPCreatedFalse) & (InvoiceDCPLifeCycle
            = InvoiceDCPLifeCycleSetCreated) : InvoiceDCPCreatedTrue;
        TRUE : InvoiceDCPCreated;
      esac;
```

Given all the resulting connected FSMs and the property formulas in NuSMV, the model checking tool can do formal and automatic verification against (business) rules.

The models shown in Figure 6.14 can be verified against the (business) rule defined in Equation 6.1 of Section 6.4 and no violations are identified. However, another process, shown in Figure 7.4, can also be verified. This process performs an unconditional authorization of the invoice after receiving it and the amount of the invoice has no influence on the execution of the *Authorize Invoice* task. This is not in conflict with the (business) rule as it does not imply any conditions on whether an authorization can be made if the amount is less than a threshold.

Using (Business) Rule 4.2 and its formalization in CTL shown in Equation 6.3, this process does not verify anymore, but the one given in Figure 6.14 still does.

Using our verification approach, we can identify if conditions in the process models are not corresponding to a (business) rule. For example, let us consider the process fragment

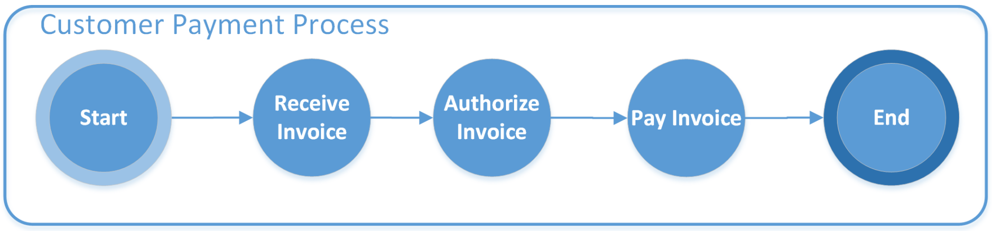


Figure 7.4: Customer Payment Process which Authorization at all times

in Figure 7.5, where the conditions in the customer payment process are swapped. In this process, an invoice is to be authorized only if its amount is *less* than a threshold.

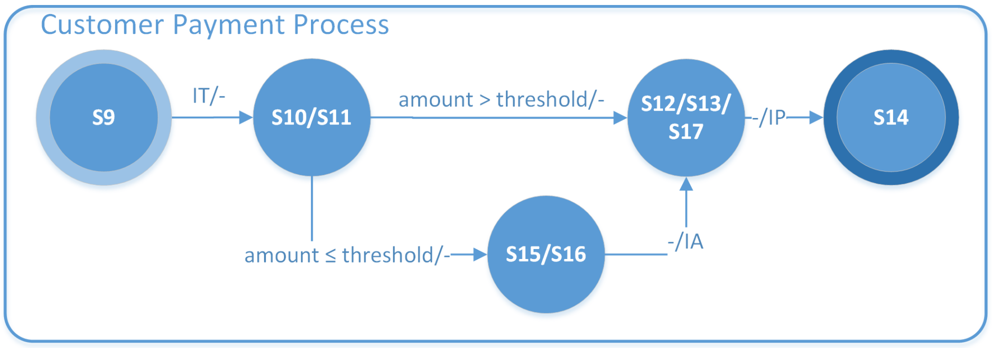


Figure 7.5: Customer Payment Process with Swapped Conditions

This does, of course, violate the (business) rule given above and the model checking tool provides a counter-example. Additionally, the corresponding input signals of the pay invoice states are not fulfilled and, thus, the task can not be performed.

As this evaluation illustrates, there are neither adaptions on the services and their semantic specifications nor the semantic task specifications necessary to change the properties a process is verified against. Actually, only changing the (business) rule is sufficient for verifying processes against different kinds of properties.

CHAPTER 8

Discussion and Future Work

The verification does not account for a process to actually reach a final state, i.e., to reach its end. Processes such as the one shown in Figure 8.1 can still be verified. Technically speaking, this behavior is not wrong as the process does not breach any conditions, but the object life cycle is not processed all the way through, i.e., the "booked" attribute is never set.

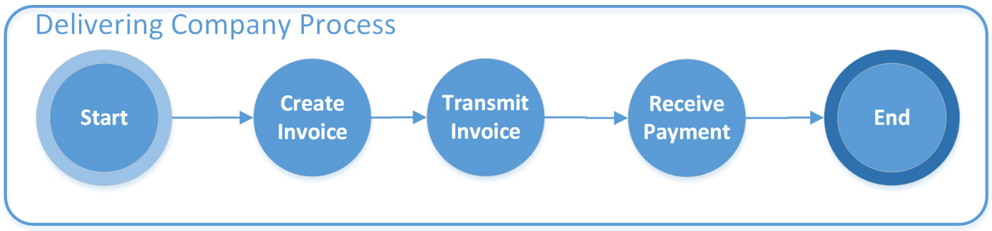


Figure 8.1: Delivering Company Process without Book Invoice Task

To account for processes that do not finish in a final state, we would have to introduce additional properties in NuSMV. Essentially, we would have to define that all involved state machines have to reach a final state. Technically, this could be accomplished by introducing properties, which check whether a state machine is in a defined state and if this state is not left anymore, i.e., the process stays in this final state.

Business rules can be implemented in WS-BPEL (Business Process Execution Language), a popular orchestration language for Web services, which includes procedural constructs such as loops and conditional selection of services [79]. However, in contrast to our approach, business rules are not identifiable as such in the procedural code. So, it is very difficult to extract what a business rule exactly states, and they are difficult to reuse. This approach is inherently different from ours as presented above and requires a different form of verification. In our approach, we actually bridge the gap between declarative and procedural specifications and connect them via semantic task specifications.

Business rules can also be implemented *within* (Web) services. However, this requires knowledge about *variability* for different processes in their specification (possibly with conditional preconditions) and of *all* tacit business rules as well. And it reduces the reusability of such services and contradicts the intention of the service approach. We showed above that simply adding knowledge of a business process to a semantic service specification makes it specific to this process and does not allow the reuse of this service in another business process any more.

While this thesis deals with verification of service composition, it is also possible to provide an automatic composition of services based on our combination of semantic Web-service specifications and business rules. Since FLUX also provides the possibility to automatically generate *plans* and even to develop an additional planner, it is possible to automatically generate service compositions for achieving a given *goal condition.* These would be verified by definition through this way of being generated. In addition, when a complete set of additional business rules is employed, the related business processes would be valid, too. Of course, the completeness of business rules cannot really be guaranteed, so that a validation of generated service compositions as business processes will still be required. In [45] we actually used a planner in FLUX for formal verification.

It is also possible for FLUX to have multiple operations to be triggered by a single operation. In such a situation, it creates all possible plans, from which a valid one in the sense of the business process may be chosen. FLUX can even deal with concurrency. For further information on the Fluent Calculus implemented by FLUX, we refer the reader to the FLUX manual at `http://www.fluxagent.org`. Overall, our verification approach using FLUX can handle everything that FLUX can.

This planning feature of FLUX can also be used for a different kind of verification approach. In this paper, we focus on the verification of given sequences of services, but FLUX also provides the means for a verification of whether a goal condition can be achieved by such a sequence. In [45] we used a negative goal condition as a condition to be avoided and used a planner to check whether this goal condition can be reached. If the goal condition can be reached the proposed models can not be verified.

Since our new approach connects task- and artefact-centric models systematically, it may also be useful for automatically verifying their consistency. Such consistency checks are complementary to the approach by Lohmann et al. [60], which employs compliance rules to automatically construct artefact-centric models from task-centric BPMs.

We tacitly assume that the OWL-S specification of the used Web services are defined using the same ontology as the one used for defining the object life cycles. In the context of an Enterprise Architecture, this assumption appears to be valid, because everything should be consistently defined within it.

FSMs (including synchronization through signals) are sufficient for our example, but certain more complex processes cannot be translated to FSMs at all, due to their inherent limitation of expressiveness. For example, BPMN multi-instance tasks, which can create a potentially infinite number of parallel task instances, cannot be translated to FSMs, also not to a fixed number of FSMs that are synchronized with signals.

Instead of FSMs, however, Petri nets may be used. BPMN models can be transformed (automatically) to Petri nets according to, e.g., Raedts et al. [88] and Dijkman et al. [27].

Still, our approach intrinsically needs additional object life cycle models. These can be represented in Petri nets as well (possibly even translated from FSMs). Meyer et al. [69] actually propose an approach that can integrate process models with object life cycles in Petri nets.

For using model checking tools, yet another transformation from Petri nets to their specific input languages is needed. For example, Raedts et al. [88] also propose automatic translation from Petri nets to mCRL2 to facilitate the use of model checking tools based on this language for the verification of process models.

Based on this related work, our new approach of formalizing properties using object life cycles can be easily generalized for unrestricted BPMN. Due to the change of the formalism, the formulation of properties representing a business rule has to be changed as well. Since Petri nets focus on *transitions* and *places* rather than states, small adaptations in these property formulas are necessary (i.e., references to places instead of states).

Procedural modeling languages allow for deliberately complex *orchestration* of Web services. We restrict service composition to sequences of services, where its semantic specification can be derived based on previous work in [76, 34], in order to achieve at least some systematic derivation.

Teschke in his contribution to [103] proposes to create a specification according to a business process requirement for retrieving components that are in a subtyping relationship to this specification. According to this idea, services may substitute components if they are in a behavioral subtyping relationship. This may involve pre- and postconditions as well. In fact, our approach could retrieve services for tasks based on their semantic specifications as well, since they adhere to subtyping. An automatic matching approach is conceivable.

Sanchez et al. [96] state that it is difficult to align business processes and the underlying information technology, i.e., implementing IT services. They identify incompatibilities between tasks in BPMN with their implementing services and formalize them mathematically. In contrast to our work, they focus on incompatibilities of service names or different input and output parameters and, in contrast to our approach, do not consider pre- and postconditions of services. Utilizing algorithms allows them to minimize incompatibilities and to find matching services more efficiently as they show in a case study. This approach could be combined with the approach of this thesis to identify possible matches of services for tasks that are not solely based on semantic task specifications.

Our systematic approach for deriving enriched task specifications from business rules was only studied yet for a specific kind of business rule formalized in a specific way. Future work should investigate this further.

Throughout this thesis we used a *business* process model to demonstrate the theoretical constructs of our approach. Our approach is however not limited to business processes or, more generally, a specific domain, but can rather be used in a variety of process models. Currently, we are applying our approach to a problem in the automotive domain. This involves a *cyber-phyiscal system* with a physic environment model.

CHAPTER 9

Conclusion

In artefact-centric modeling of business processes, artefacts and their life cycles are considered first-class citizens. This is in contrast to the more wide-spread modeling of business processes with the control flow between tasks in mind. Our approach bridges the gap between these two philosophies by annotating the tasks in BPMs with semantic task specification, which is defined using the object life cycles of artefacts. More generally, this approach implements a connection between task- and artefact-centric models. We show how this connection can be used for a seamless formal verification approach of BPMs through model checking.

The key research questions addressed in this thesis are:

1. How can task-centric and artefact-centric models be connected?

   We use *semantic task specifications* to relate tasks to artefact-centric models, i.e., we ground their logic representation in attributes of (extended) object life cycles.

2. Are semantic service specifications sufficient for task-centric process models?

   We found that *semantic service specifications* are not sufficient for task-centric models. Using only semantic service specification would limit the reusability of services in different process models. Moreover, adapting the semantic service specifications to fit more than one process model leads to over-specification of their implementing services.

3. How can a software service be reused in a different context without changing its implementation/specification?

   Providing a declarative specification of context information allows us to reuse services. We use (business) rules to express *context-dependent conditions* and enrich semantic task specifications with the information necessary. Neither the services nor their semantic specification have to be altered in our approach and thus over-specification does not occur. This enables reuse in different contexts.

4. Is it possible to decouple semantic specifications from process knowledge?

   We propose to relate the semantic specifications exclusively to attributes of (extended) object life cycles. Thus, no knowledge about the process that tasks are enacted in is required.

5. Is formal verification of process models against object life cycles possible?

   Formal verification of process models against object life cycles can be performed using semantic task specifications and (business) rules. We use NuSMv to formally verify the consistency of process models and the object life cycles they operate on.

Connecting conceptual models of processes and artefacts enables us to address all these questions. We use semantic task specifications to establish a formal connection between (extended) object life cycles and tasks of process models. The necessity of semantic task specifications is motivated by a verification and validation mismatch when using only semantic service specification. Actually, not only would services be over-specified, but also their reuse would be limited. In fact, they cannot be reused easily in different context without changing their implementation. Semantic task specifications decouple the service and its implementation from the context they are invoked in and, by doing so, enables their reuse. We used the Fluent Calculus implemented in FLUX to identify this V&V mismatch.

We use conditional (business) rules as a means to specify context information, i.e., they specify context-dependent conditions. These rules restrict process models, more precisely the paths taken in a process model, and are related to (extended) object life cycles. Using this approach allows us to decouple context information and semantic task specifications from process knowledge.

All these connected models together provide the means for formal verification through a model checker.

List of Figures

Listings

Bibliography

[1] N. Al Saiyd, I. Al Said, and A. Al Neaimi. Towards an ontological concepts for domain-driven software design. In *Networked Digital Technologies, 2009. NDT '09. First International Conference on*, pages 127–131, July 2009.

[2] T. Allweyer. *BPMN 2.0: Introduction to the Standard for Business Process Modeling.* Books on Demand, 2010.

[3] G. Alonso, F. Casati, H. A. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and Applications.* Data-Centric Systems and Applications. Springer, Berlin, 2004.

[4] F. D. Angelis, A. Polini, and G. D. Angelis. A Counter-Example Testing Approach for Orchestrated Services. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, pages 373–382. IEEE Computer Society, 2010.

[5] J. Arias-Fisteus, L. S. Fernández, and C. D. Kloos. Applying model checking to BPEL4WS business collaborations. In H. Haddad, L. M. Liebrock, A. Omicini, and R. L. Wainwright, editors, *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, pages 826–830. ACM, 2005.

[6] A. Armando and S. E. Ponta. Model Checking of Security-Sensitive Business Processes. In P. Degano and J. D. Guttman, editors, *Formal Aspects in Security and Trust, 6th International Workshop, FAST 2009, Eindhoven, The Netherlands, November 5-6, 2009, Revised Selected Papers*, volume 5983 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2009.

[7] A. Armando and S. E. Ponta. Model checking authorization requirements in business processes. *Computers & Security*, 40(0):1 – 22, 2014.

[8] C. Baier and J. Katoen. *Principles of Model Checking.* MIT Press, Cambridge, MA, USA, 2008.

[9] I. Bajwa, A. Samad, S. Mumtaz, R. Kazmi, and A. Choudhary. BPM Meeting with SOA: A Customized Solution for Small Business Enterprises. In *Information*

Management and Engineering, 2009. ICIME '09. International Conference on, pages 677–682, April 2009.

[10] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3):42–52, May 2016.

[11] C. Barros and M. A. J. Song. Automatized Checking of Business Rules for Activity Execution Sequence in Workflows. *Journal of Software (JSW)*, 7(2):374–381, 2012.

[12] G. Baryannis and D. Plexousakis. WSSL: A Fluent Calculus-Based Language for Web Service Specifications. In C. Salinesi, M. C. Norrie, and O. Pastor, editors, *Advanced Information Systems Engineering - 25th International Conference, CAiSE 2013, Valencia, Spain, June 17-21, 2013. Proceedings*, volume 7908 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2013.

[13] J. Becker, P. Delfmann, M. Eggert, and S. Schwittay. Generalizability and Applicability of Model-based Business Process Compliance-Checking Approaches — A State-of-the-Art Analysis and Research Roadmap. *BuR — Business Research*, 5(2):221–247, 2012.

[14] J. Becker, M. Rosemann, and C. von Uthmann. Guidelines of Business Process Modeling. In W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management, Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 30–49. Springer, 2000.

[15] K. H. Bennett and V. Rajlich. Software maintenance and evolution: a roadmap. In A. Finkelstein, editor, *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 73–87. ACM, 2000.

[16] A. Bhuvaneswari and G. R. Karpagam. Applying fluent calculus for automated and dynamic semantic web service composition. In A. Alnsour and S. Aljawarneh, editors, *Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications, ISWSA 2010, Amman, Jordan, June 14-16, 2010*, page 16. ACM, 2010.

[17] M. Born, F. Dörr, and I. Weber. User-Friendly Semantic Annotation in Business Process Modeling. In M. Weske, M. Hacid, and C. Godart, editors, *Web Information Systems Engineering - WISE 2007 Workshops, WISE 2007 International Workshops, Nancy, France, December 3, 2007, Proceedings*, volume 4832 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2007.

[18] M. Born, J. Hoffmann, T. Kaczmarek, M. Kowalkiewicz, I. Markovic, J. Scicluna, I. Weber, and X. Zhou. Semantic Annotation and Composition of Business Processes with Maestro. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *The Semantic Web: Research and Applications, 5th European Semantic*

Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings, volume 5021 of *Lecture Notes in Computer Science*, pages 772–776. Springer, 2008.

[19] M. Born, J. Hoffmann, T. Kaczmarek, M. Kowalkiewicz, I. Markovic, J. Scicluna, I. Weber, and X. Zhou. Supporting Execution-Level Business Process Modeling with Semantic Technologies. In X. Zhou, H. Yokota, K. Deng, and Q. Liu, editors, *Database Systems for Advanced Applications, 14th International Conference, DASFAA 2009, Brisbane, Australia, April 21-23, 2009. Proceedings*, volume 5463 of *Lecture Notes in Computer Science*, pages 759–763. Springer, 2009.

[20] T. Burkhart, S. Wolter, M. Schief, J. Krumeich, C. D. Valentin, D. Werth, P. Loos, and D. Vanderhaeghen. A comprehensive approach towards the structural description of business models. In J. Kacprzyk, D. Laurent, and R. Chbeir, editors, *International Conference on Management of Emergent Digital EcoSystems, MEDES '12, Addis Ababa, Ethiopia, October 28-31, 2012*, pages 88–102. ACM, 2012.

[21] L. Cabral, B. Norton, and J. Domingue. The Business Process Modelling Ontology. In *Proceedings of the 4th International Workshop on Semantic Business Process Management*, SBPM '09, pages 9–16, New York, NY, USA, 2009. ACM.

[22] H. Chesbrough and J. Spohrer. A research manifesto for services science. *Communications of the ACM*, 49(7):35–40, 2006.

[23] M. Chinosi and A. Trombetta. BPMN: An Introduction to the Standard. *Comput. Stand. Interfaces*, 34(1):124–134, Jan. 2012.

[24] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Service Definition Language (WSDL). Technical report, W3C, Mar. 2001.

[25] T. H. Davenport. *Process Innovation: Reengineering Work Through Information Technology.* Harvard Business School Press, Boston, MA, USA, 1993.

[26] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In R. Fagin, editor, *Database Theory - ICDT 2009, 12th International Conference, St. Petersburg, Russia, March 23-25, 2009, Proceedings*, volume 361 of *ACM International Conference Proceeding Series*, pages 252–267. ACM, 2009.

[27] R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Inf. Softw. Technol.*, 50(12):1281–1294, Nov. 2008.

[28] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, Today, and Tomorrow. In M. Mazzara and B. Meyer, editors, *Present and Ulterior Software Engineering.*, pages 195–216. Springer, 2017.

[29] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[30] M. Estañol, M.-R. Sancho, and E. Teniente. Verification and Validation of UML Artifact-Centric Business Process Models. In J. Zdravkovic, M. Kirikova, and P. Johannesson, editors, *Advanced Information Systems Engineering*, volume 9097 of *Lecture Notes in Computer Science*, pages 434–449. Springer International Publishing, 2015.

[31] Y. Feng and M. Kirchberg. Verifying OWL-S service process models. In *Proceedings of the 2011 IEEE International Conference on Web Services*, ICWS '11, pages 307–314, Washington, DC, USA, 2011. IEEE Computer Society.

[32] M. Fiammante. *Dynamic SOA and BPM: Best Practices for Business Process Management and SOA Agility.* IBM Press, 1st edition, 2009.

[33] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000. AAI9980887.

[34] R. E. Fikes, P. E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artif. Intell.*, 3(1):251–288, Jan. 1972.

[35] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[36] A. Filipowska, M. Kaczmarek, M. Kowalkiewicz, I. Markovic, and X. Zhou. Organizational Ontologies to Support Semantic Business Process Management. In *Proceedings of the 4th International Workshop on Semantic Business Process Management*, SBPM '09, pages 35–42, New York, NY, USA, 2009. ACM.

[37] M. Geiger, S. Harrer, J. Lenhard, M. Casar, A. Vorndran, and G. Wirtz. BPMN Conformance in Open Source Engines. In *2015 IEEE Symposium on Service-Oriented System Engineering, SOSE 2015, San Francisco Bay, CA, USA, March 30 - April 3, 2015*, pages 21–30. IEEE, 2015.

[38] M. Geiger, S. Harrer, J. Lenhard, and G. Wirtz. BPMN 2.0: The state of support and implementation. *Future Generation Computer Systems*, 80:250 – 262, 2018.

[39] D. Georgakopoulos and M. P. Papazoglou. *Service-Oriented Computing.* The MIT Press, 2008.

[40] W. W. W. Group. OWL-based Web Service Ontology Version 1.2. http://www.daml.org/services/owl-s/1.2/, March 2004.

[41] Group, W3c W. Web Services Glossary, Feb. 2004.

[42] M. Havey. *Essential business process modeling.* O'Reilly, 2005.

[43] M. Hepp, F. Leymann, J. Domingue, A. Wahler, and D. Fensel. Semantic Business Process Management: A Vision Towards Using Semantic Web Services for Business Process Management. In F. C. M. Lau, H. Lei, X. Meng, and M. Wang, editors, *2005 IEEE International Conference on e-Business Engineering (ICEBE 2005), 18-21 October 2005, Beijing, China*, pages 535–540. IEEE Computer Society, 2005.

[44] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez-Barahona. The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review. *ACM Comput. Surv.*, 46(2):28:1–28:28, Dec. 2013.

[45] R. Hoch and H. Kaindl. Verification of Feature Coordination using the Fluent Calculus. In E. Damiani, G. Spanoudakis, and L. A. Maciaszek, editors, *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2018, Funchal, Madeira, Portugal, March 23-24, 2018.*, pages 169–179. SciTePress, 2018.

[46] R. Hoch, H. Kaindl, R. Popp, and D. Ertl. Comprehensive Integration of Executable Business Process Models with Semantic Concept and Task Specifications. In *Proceedings of the Fifth International Conference on Business Intelligence and Technology (BUSTECH 2015)*, 2015. Vortrag: IARIA-Konferenz, Nice, France; 2015-03-22 – 2015-03-27.

[47] R. Hoch, H. Kaindl, R. Popp, D. Ertl, and H. Horacek. Semantic Service Specification for V&V of Service Composition and Business Processes. In T. X. Bui and R. H. S. Jr., editors, *48th Hawaii International Conference on System Sciences, HICSS 2015, Kauai, Hawaii, USA, January 5-8, 2015*, pages 1370–1379. IEEE Computer Society, 2015.

[48] R. Hoch, M. Rathmair, H. Kaindl, and R. Popp. Verification of Business Processes Against Business Rules Using Object Life Cycles. In Á. Rocha, A. M. R. Correia, H. Adeli, L. P. Reis, and M. M. Teixeira, editors, *New Advances in Information Systems and Technologies - Volume 1 [WorldCIST'16, Recife, Pernambuco, Brazil, March 22-24, 2016].*, volume 444 of *Advances in Intelligent Systems and Computing*, pages 589–598. Springer, 2016.

[49] H. Huang, W. Tsai, S. Bhattacharya, X. Chen, Y. Wang, and J. Sun. Business Rule Extraction from Legacy Code. In *COMPSAC '96 - 20th Computer Software and Applications Conference, August 19-23, 1996, Seoul, Korea*, pages 162–167. IEEE Computer Society, 1996.

[50] H. Kaindl, R. Hoch, and R. Popp. Semantic Task Specification in Business Process Context. In S. Assar, O. Pastor, and H. Mouratidis, editors, *11th International Conference on Research Challenges in Information Science, RCIS 2017, Brighton, United Kingdom, May 10-12, 2017*, pages 286–291. IEEE, 2017.

[51] F. Kamoun. A Roadmap Towards the Convergence of Business Process Management and Service Oriented Architecture. *Ubiquity*, 2007(April), Apr. 2007.

[52] O. M. Kherbouche, A. Ahmad, and H. Basson. Using model checking to control the structural errors in BPMN models. In R. J. Wieringa, S. Nurcan, C. Rolland, and J. Cavarero, editors, *IEEE 7th International Conference on Research Challenges in Information Science, RCIS 2013, Paris, France, May 29-31, 2013*, pages 1–12. IEEE, 2013.

[53] M. Klusch and A. Gerber. Evaluation of Service Composition Planning with OWLS-XPlan. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology - Workshops, Hong Kong, China, 18-22 December 2006*, pages 117–120. IEEE Computer Society, 2006.

[54] M. Klusch, A. Gerber, and M. Schmidt. Semantic Web Service Composition Planning with OWLS-Xplan. In T. R. Payne and V. A. M. Tamma, editors, *Agents and the Semantic Web, Papers from the 2005 AAAI Fall Symposium, Arlington, Virginia, USA, November 4-6, 2005.*, volume FS-05-01 of *AAAI Technical Report*, pages 55–62. AAAI Press, 2005.

[55] R. K. L. Ko, S. S. G. Lee, and E. W. Lee. Business Process Management (BPM) standards: A survey. *Business Process Management journal*, 15(5), 2009.

[56] M. M. Lehman. Laws of Software Evolution Revisited. In C. Montangero, editor, *Software Process Technology, 5th European Workshop, EWSPT '96, Nancy, France, October 9-11, 1996, Proceedings*, volume 1149 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.

[57] A. Ligeza, K. Kluza, G. J. Nalepa, and T. Potempa. AI Approach to Formal Analysis of BPMN Models. Towards a Logical Model for BPMN Diagrams. In M. Ganzha, L. A. Maciaszek, and M. Paprzycki, editors, *Federated Conference on Computer Science and Information Systems - FedCSIS 2012, Wroclaw, Poland, 9-12 September 2012, Proceedings*, pages 931–934, 2012.

[58] A. Ligeza and T. Potempa. AI Approach to Formal Analysis of BPMN Models: Towards a Logical Model for BPMN Diagrams. In M. Mach-Król and T. Pelech-Pilichowski, editors, *Advances in Business ICT, result of the 3rd International Workshop on Advances in Business ICT, ABICT 2012, Wroclaw, Poland, September 9-12, 2012*, volume 257 of *Advances in Intelligent Systems and Computing*, pages 69–88. Springer, 2012.

[59] N. Lohmann. Compliance by Design for Artifact-Centric Business Processes. In S. Rinderle-Ma, F. Toumani, and K. Wolf, editors, *Business Process Management - 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings*, volume 6896 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2011.

[60] N. Lohmann. Compliance by design for artifact-centric business processes. *Information Systems*, 38(4):606 – 618, 2013.

[61] S. Lovrencic, K. Rabuzin, and R. Picek. Formal modelling of business rules: What kind of tool to use? *Journal of Information and Organizational Sciences; Vol 30, No 2 (2006)*, 30, 12 2006.

[62] M. Maidl. The Common Fragment of CTL and LTL. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 643–652. IEEE Computer Society, 2000.

[63] F. Marzullo, J. Moreira de Souza, and G. Xexeo. A domain-driven approach for enterprise development, using BPM, MDA, SOA and Web services. In *Innovations in Information Technology, 2008. IIT 2008. International Conference on*, pages 150–154, Dec 2008.

[64] F. P. Marzullo, J. M. de Souza, and J. R. Blaschek. A Domain-Driven Development Approach for Enterprise Applications, Using MDA, SOA and Web Services. In *10th IEEE International Conference on E-Commerce Technology (CEC 2008) / 5th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (EEE 2008), July 21-14, 2008, Washington, DC, USA*, pages 432–437. IEEE Computer Society, 2008.

[65] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.

[66] D. L. McGuinness, F. Van Harmelen, et al. OWL Web Ontology Language Overview. *W3C recommendation*, 10(10):2004, 2004.

[67] B. Medjahed and A. Bouguettaya. *Service Composition for the Semantic Web*. Springer, Berlin, 2011.

[68] A. Meyer and M. Weske. Activity-centric and artifact-centric process model roundtrip. In N. Lohmann, M. Song, and P. Wohed, editors, *Business Process Management Workshops - BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers*, volume 171 of *Lecture Notes in Business Information Processing*, pages 167–181. Springer, 2013.

[69] A. Meyer and M. Weske. Weak Conformance between Process Models and Synchronized Object Life Cycles. In X. Franch, A. Ghose, G. Lewis, and S. Bhiri, editors, *Service-Oriented Computing*, volume 8831 of *Lecture Notes in Computer Science*, pages 359–367. Springer Berlin Heidelberg, 2014.

[70] H. Mili, G. Tremblay, G. B. Jaoude, E. Lefebvre, L. Elabed, and G. E. Boussaidi. Business Process Modeling Languages: Sorting Through the Alphabet Soup. *ACM Comput. Surv.*, 43(1):4:1–4:56, Dec. 2010.

[71] M. Montali, M. Pesic, W. M. P. v. d. Aalst, F. Chesani, P. Mello, and S. Storari. Declarative Specification and Verification of Service Choreographiess. *ACM Trans. Web TWEB*, 4(1):3:1–3:62, Jan. 2010.

[72] R. Mrasek, J. A. Mülle, K. Böhm, M. Becker, and C. Allmann. User-Friendly Property Specification and Process Verification - A Case Study with Vehicle-Commissioning Processes. In S. W. Sadiq, P. Soffer, and H. Völzer, editors, *Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings*, volume 8659 of *Lecture Notes in Computer Science*, pages 301–316. Springer, 2014.

[73] C. Natschläger. Towards a BPMN 2.0 Ontology. In R. M. Dijkman, J. Hofstetter, and J. Koehler, editors, *Business Process Model and Notation - Third International Workshop, BPMN 2011, Lucerne, Switzerland, November 21-22, 2011. Proceedings*, volume 95 of *Lecture Notes in Business Information Processing*, pages 1–15. Springer, 2011.

[74] Y. Ni and Y. Fan. Model transformation and formal verification for Semantic Web Services composition. *Advances in Engineering Software*, 41(6):879–885, 2010.

[75] A. Nigam and N. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428–445, 2003.

[76] N. J. Nilsson. *Principles of Artificial Intelligence.* Springer, Berlin, Heidelberg, Germany, 1982.

[77] NuSMV. NuSMV: A new symbolic model checker. `http://nusmv.fbk.eu/`. [Online; accessed Oct. 11, 2018].

[78] NuSMV. NuSMV: A new symbolic model checker manual. Technical report, NuSMV, Oct. 2018.

[79] OASIS. *Business Process Execution Language 2.0 (WS-BPEL 2.0)*, 2007.

[80] T. O. M. G. (OMG). Semantics of Business Vocabulary and Business Rules (SBVR), January 2008.

[81] T. O. M. G. (OMG). Business Process Model and Notation (BPMN). http://www.omg.org/spec/BPMN/2.0, Jan. 2011. [Online; accessed 03-July-2015].

[82] B. Orriëns, J. Yang, and M. P. Papazoglou. A Framework for Business Rule Driven Web Service Composition. In M. A. Jeusfeld and O. Pastor, editors, *Conceptual Modeling for Novel Application Domains, ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM, Chicago, IL, USA, October 13, 2003, Proceedings*, volume 2814 of *Lecture Notes in Computer Science*, pages 52–64. Springer, 2003.

[83] C. Ouyang, M. Dumas, W. M. P. V. D. Aalst, A. H. M. T. Hofstede, and J. Mendling. From Business Process Models to Process-oriented Software Systems. *ACM Trans. Softw. Eng. Methodol.*, 19(1):2:1–2:37, Aug. 2009.

[84] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, and W. M. P. van der Aalst. From BPMN Process Models to BPEL Web Services. In *2006 IEEE International Conference on Web Services (ICWS 2006), 18-22 September 2006, Chicago, Illinois, USA*, pages 285–292. IEEE Computer Society, 2006.

[85] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In J. Huai, R. Chen, H. Hon, Y. Liu, W. Ma, A. Tomkins, and X. Zhang, editors, *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, pages 805–814. ACM, 2008.

[86] C. Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, Oct. 2003.

[87] R. Popp, R. Hoch, and H. Kaindl. A Connection of Task-centric with Artefact-centric Models through Semantic Task Specification and its Use for Formal Verification. In T. Bui, editor, *50th Hawaii International Conference on System Sciences, HICSS 2017, Hilton Waikoloa Village, Hawaii, USA, January 4-7, 2017*, pages 1–10. ScholarSpace / AIS Electronic Library (AISeL), 2017.

[88] I. Raedts, M. Petkovic, Y. S. Usenko, J. M. E. M. van der Werf, J. F. Groote, and L. J. Somers. Transformation of BPMN Models for Behaviour Analysis. In J. C. Augusto, J. Barjis, and U. Ultes-Nitsche, editors, *Modelling, Simulation, Verification and Validation of Enterprise Information Systems, Proceedings of the 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS-2007, In conjunction with ICEIS 2007, Funchal, Madeira, Portugal, June 2007*, pages 126–137. INSTICC PRESS, 2007.

[89] V. Rajlich. Software evolution and maintenance. In J. D. Herbsleb and M. B. Dwyer, editors, *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 133–144. ACM, 2014.

[90] M. Rathmair, R. Hoch, H. Kaindl, and R. Popp. Consistently Formalizing a Business Process and its Properties for Verification: A Case Study. In J. Ralyté, S. España, and O. Pastor, editors, *The Practice of Enterprise Modeling - 8th IFIP WG 8.1. Working Conference, PoEM 2015, Valencia, Spain, November 10-12, 2015, Proceedings*, volume 235 of *Lecture Notes in Business Information Processing*, pages 126–140. Springer, 2015.

[91] R. Reiter. The Frame Problem in Situation the Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In V. Lifschitz, editor,

Artificial Intelligence and Mathematical Theory of Computation, pages 359–380. Academic Press Professional, Inc., San Diego, CA, USA, 1991.

[92] F. Rosenberg and S. Dustdar. Business Rules Integration in BPEL - A Service-Oriented Approach. In *7th IEEE International Conference on E-Commerce Technology (CEC 2005), 19-22 July 2005, München, Germany*, pages 476–479. IEEE Computer Society, 2005.

[93] R. G. Ross. *Principles of the Business Rule Approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[94] K. Ryndina, J. M. Küster, and H. C. Gall. Consistency of Business Process Models and Object Life Cycles. In T. Kühne, editor, *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *Lecture Notes in Computer Science*, pages 80–90. Springer, 2006.

[95] I. Salomie, V. R. Chifu, I. Harsa, and M. Gherga. Web service composition using fluent calculus. *IJMSO*, 5(3):238–250, 2010.

[96] E. S. Sánchez, P. J. Clemente, A. E. Prieto, and C. Ortiz-Caraballo. Aligning Business Processes With the Services Layer Using a Semantic Approach. *IEEE Access*, 7:2904–2927, 2019.

[97] Z. Sbai, A. Missaoui, K. Barkaoui, and R. Ben Ayed. On the verification of business processes by model checking techniques. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, volume 1, pages V1–97–V1–103, Oct 2010.

[98] J. Schumacher and M. Meyer. *Customer Relationship Management strukturiert dargestellt: Prozesse, Systeme, Technologien.* Springer Berlin Heidelberg, 2003.

[99] A. Sill. The Design and Architecture of Microservices. *IEEE Cloud Computing*, 3(5):76–80, Sep. 2016.

[100] SPIN. SPIN Verifying Multi-threaded Software with Spin. `http://spinroot.com/spin/whatispin.html`. [Online; accessed 01-December-2014].

[101] SUPER. Semantics Utilised for Process Management within and between Enterprises. `http://www.ip-super.org/`, 2009. [Online; accessed 21-January-2015].

[102] W. Tan and M. Zhou. *Business and Scientific Workflows: A Web Service-Oriented Approach.* IEEE Press Series on Systems Science and Engineering. Wiley, 2013.

[103] T. Teschke. 1 Business Process Oriented Component Retrieval. In V. Zsok and I. Juhasz, editors, *ECOOP'01 European Conference on Object-Oriented Programming, Proceedings, Workshop Reader (Poster Session)*, volume 2323 of *Lecture Notes in Computer Science*, pages 213–223. Springer, Springer, 2001.

[104] The Business Rules Group. Defining Business Rules – What Are They Really?, July 2000.

[105] M. Thielscher. Introduction to the Fluent Calculus. *Electron. Trans. Artif. Intell.*, 2:179–192, 1998.

[106] M. Thielscher. FLUX: A Logic Programming Method for Reasoning Agents. *Theory and Practice of Logic Programming*, 5(4-5):533–565, 2005.

[107] W. M. van der Aalst, M. Weske, and D. Grünbauer. Case handling: a new paradigm for business process support. *Data and Knowledge Engineering*, 53(2):129 – 162, 2005.

[108] T. van Eijndhoven, M. Iacob, and M. L. Ponisio. Achieving Business Process Flexibility with Business Rules. In *12th International IEEE Enterprise Distributed Object Computing Conference, ECOC 2008, 15-19 September 2008, Munich, Germany*, pages 95–104. IEEE Computer Society, 2008.

[109] W3C. Web Service Modeling Language (WSML), June 2005. [Online; accessed 08-January-2019].

[110] W3C. Web Service Modeling Ontology (WSMO) Primer, June 2005. [Online; accessed 07-March-2019].

[111] I. Weber, J. Hoffmann, and J. Mendling. Beyond soundness: on the verification of semantic business process models. *Distributed and Parallel Databases*, 27(3):271–343, Jun 2010.

[112] P. Wegner and S. B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In S. Gjessing and K. Nygaard, editors, *ECOOP'88 European Conference on Object-Oriented Programming, Oslo, Norway, August 15-17, 1988, Proceedings*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77. Springer, 1988.

[113] WFMC. XML Process Definition Language Version 2.2, August 2012.

[114] Q. Wu, C. Pu, A. Sahai, and R. S. Barga. Categorization and Optimization of Synchronization Dependencies in Business Processes. In R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 306–315. IEEE Computer Society, 2007.

[115] M. Wynn, H. Verbeek, W. van der Aalst, A. ter Hofstede, and D. Edmond. Business process verification – finally a reality! *Business Process Management Journal*, 15(1):74–92, 2009.

[116] Z. Xiao, I. Wijegunaratne, and X. Qiang. Reflections on SOA and Microservices. In G. Li and Y. Yu, editors, *4th International Conference on Enterprise Systems, ES 2016, Melbourne, Australia, November 2-3, 2016*, pages 60–67. IEEE Computer Society, 2016.

[117] L. D. Xu, W. Viriyasitavat, P. Ruchikachorn, and A. Martin. Using Propositional Logic for Requirements Verification of Service Workflow. *IEEE Trans. Industrial Informatics*, 8(3):639–646, 2012.

[118] E. Ziaka, D. Vrakas, and N. Bassiliades. Web Service Composition Plans in OWL-S. In J. Filipe and A. L. N. Fred, editors, *Agents and Artificial Intelligence - Third International Conference, ICAART 2011, Rome, Italy, January, 28-30, 2011. Revised Selected Papers*, volume 271 of *Communications in Computer and Information Science*, pages 240–254. Springer, 2011.

[119] O. Zimmermann, V. Doubrovski, J. Grundler, and K. Hogg. Service-oriented architecture and business process choreography in an order management scenario: rationale, concepts, lessons learned. In R. E. Johnson and R. P. Gabriel, editors, *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 301–312. ACM, 2005.

[120] O. Zimmermann, P. Krogdahl, and C. Gee. Elements of service-oriented analysis and design, 2004. [Online; accessed Oct. 11, 2018].

Acronyms

BP	Business Process.
BPEL	Business Process Execution Language.
BPM	Business Process Model.
BPMN	Business Process Model Notation.
BPMS	Business Process Management System.
CTL	Computation Tree Logic.
ERP	Enterprise Resource Planning.
FSM	Finite State Machine.
LTL	Linear Temporal Logic.
OWL	Web Ontology Language.
OWL-S	Semantic Markup for Web Services.
QVTO	Operational Query View Transformation.
RDF	Resource Description Framework.
REST	Representational State Transfer.
SBVR	Semantics of Business Vocabulary and Business Rules.
SOA	Service Oriented Architecture.
SPARQL	SPARQL Protocol and RDF Query Language.
SWRL	Semantic Web Rule Language.

UML	Unified Modeling Language.
URI	Uniform Resource Identifier.
WSDL	Web Service Definition Language.
XML	Extensible Markup Language.
XPDL	XML Process Definition Language.